République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Constantine
Faculté des Sciences
Département d'Informatique

Mémoire de Doctorat en Science en Informatique

Thème :

Développement des Outils Basés Maude pour les ECATNets. Domaine d'Application : Analyse des Programmes Ada

Présenté par : Mlle Noura Boudiaf

Date de soutenance : 10/04/2007

Composition du Jury

Mr. Zaidi Sahnoun	Professeur, Univ. Mentouri Constantine	Président
Mr. Allaoua Chaoui	Maître de Conférence, Univ. Mentouri Constantine	Rapporteur
Mr. Benmohamed Mohamed	Professeur, Univ. Mentouri Constantine	Rapporteur
Mme. Belala Faiza	Maître de Conférence, Univ. Mentouri Constantine	Examinateur
Mr. Meslati Djamel	Maître de Conférence, Univ. Badji Mokhtar Annaba	Examinateur
Mr. Kimour Mohamed Tahar	Maître de Conférence, Univ. Badji Mokhtar Annaba	Examinateur
Mr. Amar Aissani	Professeur, USTHB Alger	Examinateur

Dédicaces

Je dédie ce modeste travail à ma chère grande famille qui fait mon bonheur :

A mes parents qui m'ont soutenu pendant toutes mes longues années d'étude

A mes chères sœurs : Laila et Dhabia

A mes chers frères : Torki, Hocine, Messaoud, Mekki, Ahmed et Ibrahim,

A mes chères belles-sœurs,

A mes chers neveux,

A mes chères nièces,

A ma meilleur amie : Nacira

Remerciements

Je tiens à remercier le Docteur Allaoua Chaoui et le Professeur Mohamed Benmohamed de l'université de Constantine, de m'avoir encadré dans la réalisation de cette thèse de doctorat. Je remercie le Professeur Kamel Barkaoui du CNAM-Paris pour son aide précieuse. Je remercie les membres de jury qui ont accepté l'évaluation de ce travail. Je remercie également tous les membres de ma famille et mes collègues de travail à l'université d'Oum El Bouaghi, Mr. Lakhdar Sahbi et Mr. Messaoud Djebbar de m'avoir aidé et soutenu. Enfin, je remercie sincèrement tous ce qui ont participé de près et de loin dans la réalisation de ce travail.

Résumé

Les ECATNets [Bet93] sont une catégorie des réseaux de Petri algébriques (RPAs) basés sur une combinaison saine des réseaux de Petri de haut niveau et des types abstraits algébriques. La sémantique des ECATNets est définie en termes de la logique de réécriture [Mes96]. De telle sémantique offre une base solide et rigoureuse pour toute démarche de vérification des propriétés des systèmes décrits. L'intégration des ECATNets dans la logique de réécriture et dans son langage Maude est très prometteuse en termes de spécification et de vérification de leurs propriétés. Maude fournit une plate-forme permettant un développement facile et une exécution efficace des outils des ECATNets. Le langage Ada est un langage modulaire, puissant (orienté objet, multitâche, normalisé). Le langage Ada est souvent utilisé dans des systèmes temps réel et embarqués nécessitant un haut niveau de fiabilité. Ada est le premier langage objet standardisé de manière internationale. Cependant, l'analyse des erreurs de la concurrence comme le deadlock dans les programmes Ada ainsi que dans d'autres systèmes concurrents présente une difficulté qui défie les développeurs de tels systèmes. Cette difficulté rend les tests classiques inutilisables et la vérification de la correction peut être obtenue seulement en utilisant les méthodes formelles. L'objectif de cette thèse de doctorat est de développer certaines des outils d'analyse pour les ECATNets. Nous avons utilisé le langage de la logique de réécriture Maude pour le développement de toutes ces applications. Un deuxième objectif de cette thèse est d'étudier la translation des programmes Ada aux ECATNets afin de vérifier certaines propriétés concernant la concurrence. Nous avons divisé ce travail en deux étapes principales. En première étape, nous nous sommes concentrés sur le développement des méthodes d'analyse pour les ECATNets et leurs implémentations en utilisant Maude. Le manque des outils d'analyse pour les ECATNets nous a poussé à effectuer ce travail. Les méthodes développées pour les ECATNets dans le cadre de cette thèse incluent le graphe de couverture [Bou04b], [Bou04c], la simulation [Bou05] et l'implémentation des règles de réduction [Bou06a], [Bou06b]. En deuxième étape, nous avons montré comment utiliser certains de ces outils d'analyse proposés et développés en première étape à la vérification des programmes Ada (propriétés concernant la concurrence). D'abord, nous avons proposé une méthode systématique de translation d'un programme Ada vers les ECATNets [Bou04a]. Nous nous sommes concentrés sur la translation des éléments syntaxiques du langage Ada relatifs en particulier au concept de 'tâche'. En deuxième point, nous avons développé un compilateur Ada-ECATNet en se basant sur les directions de translation que nous avons proposées. Finalement, nous avons proposé une représentation très compactée des Ada-ECATNet obtenus [Bou06c], [Bou06d]. En utilisant les réseaux de Petri (ordinaires ou colorés) pour représenter des programmes Ada, les représentations compactées Ada-nets proposées dans la littérature sont dues à l'application des règles de réduction des réseaux de Petri après une translation complète d'un programme Ada vers le formalisme de réseaux de Petri. Dans notre approche, nous faisons une réduction au cours de translation des programmes Ada vers les ECATNets. De cette façon, la représentation Ada-ECATNet obtenue peut subir une double réduction en appliquant les règles de réduction des réseaux de Petri algébriques que nous avons déjà adaptées et implémentées pour les ECATNets. Le gain obtenu en termes d'efficacité a été montré à travers des exemples.

Abstract

ECATNets [Bet93] are a category of algebraic Petri nets (APNs) based on a safe combination of algebraic abstract types and high level Petri nets. The semantic of ECATNets is defined in terms of rewriting logic [Mes96], allowing us to built models by formal reasoning. The integration of ECATNets in rewriting logic is very promising in terms of specification and verification of their properties. Rewriting logic provides to ECATNets a simple, intuitive, and practical textual version to analyze systems, without loosing the formal semantic. Maude system provide a plate-form allowing an easy development and efficient execution for ECATNets tools. The language Ada is a modular, powerful language (object-oriented, multitask, normalized). The language Ada is often used in real-time and embeded systems requiring a high level of reliability. Ada is the first language object-oriented standardized in a international way. However, the analysis of the concurrency errors as the deadlock in the programs Ada as well as in the other concurrent systems presents a difficulty which challenges the developers of such systems. This difficulty makes the classic tests unusable and the checking of the correction can be only obtained by using the formal methods. The aim of this doctoral thesis is to develop some of analysis tools for the ECATNets. We used the language of the rewriting logic Maude for the development of all these applications. The second objective of this thesis is to study the translation of the Ada programs to ECATNets in order to verify certain properties concerning the concurrency. We divided this work into two main stages. In first stage, we focused on the development of the analysis methods for the ECATNets and their implementations by using Maude. The lack of analysis tools of the ECATNets pushed us to do this work. The methods developed for the ECATNets within the framework of this thesis include the coverability graph [Bou04b], [Bou04c], the simulation [Bou05] and the implementation of the rules of reduction [Bou06a], [Bou06b]. In the second stage, we showed how to use some of these analysis tools proposed and developed in first stage to check Ada programs (properties concerning the concurrency). At first, we proposed a systematic method of translation of an Ada program towards the ECATNets [Bou04a]. We focused on the syntactical elements' translation of the language Ada relative in particular to the concept of 'task'. Secondly, we developed an Ada-ECATNet compiler based on the translation's directions we proposed before. Finally, we proposed a representation very compact of obtained Ada-ECATNet [Bou06c], [Bou06d]. In all existing approaches concerning the utilization of Petri nets (simple or high level) in the description and the verification of the Ada's programs, we notice that these works first aim to translate completely Ada-programs to Petri nets and then apply reduction rules on the obtained Ada-nets. In our approach, the proposed reduction rules may be done during translation step. Therefore, the obtained reduced Ada-ECATNets may be submitted to another reduction such that proposed for APNs. This is possible because we adapted and implemented reduction rules defined by Schmidt [Sch97] to ECATNets in [Bou06a, Bou06b]. The obtained advantage in terms of efficiency of the execution of analysis methods was shown through examples.

Sommaire

Introduction

Les ECATNets [Bet93] sont une catégorie des réseaux de Petri algébriques (RPAs) basés sur une combinaison saine des réseaux de Petri de haut niveau et des types abstraits algébriques. La sémantique des ECATNets est définie en termes de la logique de réécriture [Mes96]. De telle sémantique offre une base solide et rigoureuse pour toute démarche de vérification des propriétés des systèmes décrits. Autant que des réseaux de Petri, les ECATNets fournissent un formalisme rapidement compris dû à leur description graphique. En outre, les ECATNets ont une théorie puissante et des outils de développement basés sur une logique ayant une sémantique saine et complète. La logique de réécriture fournit aux ECATNets une version textuelle simple, intuitive et pratique pour leur analyse sans perdre leur sémantique formelle (rigueur mathématique, raisonnement formel).

Le langage de la logique de réécriture Maude [Mes00], [Cla03] est simple, très expressif et efficace. Autant que langage de programmation, Maude est facile à comprendre et de programmer avec. Il offre peu de constructions syntaxiques et une sémantique claire. Maude est un langage compétitif en termes d'exécution avec d'autres langages de la programmation impérative. Maude offre la possibilité de décrire naturellement différents types d'applications, des petites applications déterministes aux systèmes avec un degré élevé de concurrence. En plus de la description des applications, Maude supporte la modélisation des langages, des formalismes et des logiques.

Dans ce cas, Maude fournit une plate-forme permettant un développement facile et une exécution efficace des outils des ECATNets. L'intégration des ECATNets dans la logique de réécriture et dans Maude est très prometteuse en termes de spécification et de vérification de leurs propriétés.

Le langage Ada [Iso95], [Iso01] est un langage modulaire, puissant (orienté objet, multitâche, normalisé). Le langage Ada est souvent utilisé dans la programmation des systèmes temps réel et embarqués nécessitant un haut niveau de fiabilité. Ada est le premier langage objet standardisé de manière internationale. Cependant, l'analyse des erreurs de la concurrence comme le deadlock dans les programmes Ada comme dans d'autres systèmes concurrents est souvent difficile et représente un défi aux développeurs de tels systèmes. Cette difficulté rend les tests classiques inutilisables et la vérification de la correction peut être obtenue seulement en utilisant les méthodes formelles.

L'objectif de cette thèse de doctorat est de développer certaines des outils d'analyse pour les ECATNets. Nous avons utilisé le langage de la logique de réécriture pour le développement de toutes ces applications. Un deuxième objectif de cette thèse est d'étudier la translation des programmes Ada aux ECATNets afin de vérifier certaines propriétés concernant la concurrence. Nous avons divisé ce travail en deux étapes principales. En première étape, nous nous sommes concentrés sur le développement des méthodes d'analyse pour les ECATNets et leurs implémentations en utilisant Maude. Les méthodes développées pour les ECATNets dans ce travail incluent l'analyse de l'accessibilité [Bou04b], [Bou04c], la simulation [Bou05] et l'implémentation de règles de réduction [Bou06a], [Bou06b].

Dans le cadre de l'analyse de l'accessibilité, nous avons proposé un algorithme d'analyse dynamique des ECATNets permettant de calculer le graphe de couverture et la vérification des propriétés de vivacité pour un système à états infinis ou un système à états finis. Cet algorithme permet d'étendre la possibilité de vérification de propriétés de vivacité pour les systèmes à états infinis qui n'est pas offerte pour l'instant par le Model Checking en général et celui supporté par Maude et donc le Model Checking des ECATNets, parce que les ECATNets sont intégrés dans Maude. Un outil pratique implantant cet algorithme a été crée à l'aide de l'environnement du langage Maude. Dans la logique de réécriture, un module peut être une donnée manipulée par un autre module. Ce concept est dit la réflectivité ou l'auto-interprétation. Le module qui est une entrée pour un autre module, est décrit dans un méta-niveau. Ce concept permet de décrire un ECATNet autant qu'un module dans le méta-niveau. Ce module devient une entrée pour notre outil implémentant le système d'analyse dynamique des ECATNets.

Notons que le système Maude offre une manière textuelle aux utilisateurs de créer et de manipuler leurs systèmes ECATNets. L'exécution sous le système Maude est faite en utilisant le style 'command prompt'. Dans ce cas, nous perdons l'aspect graphique du formalisme ECATNet qui est important pour la clarté, la simplicité et la lisibilité d'une description d'un système. Dans cette direction, nous avons proposé un simulateur graphique pour les ECATNets. Il permet à un utilisateur de créer un ECATNet graphiquement, et d'exécuter cet ECATNet pour un marquage initial et un nombre d'étapes de simulation quelconques.

Pour enrichir encore l'ensemble des outils destinés aux ECATNets, nous avons adapté certaines règles de réduction définies dans la littérature pour les RPAs. Les techniques d'analyse supportées par les réseaux de Petri en général, et les ECATNets en particulier sont habituellement très coûteuses, et toute règle de réduction proposée pour diminuer une telle explosion combinatoire est intéressante. Dans cette direction, nous avons adapté les règles de réduction des RPAs proposées dans [Sch97] aux ECATNets. Un outil implémentant de telles règles de réduction pour les ECATNets a été effectuée sous Maude. De telle implémentation a été possible grâce aussi au concept de méta-calcul. Le niveau d'abstraction très élevé de Maude a rendu de telle implémentation simple.

En deuxième étape, nous avons appliqué certains des outils d'analyse proposés et développés en première étape à la vérification des programmes Ada (propriétés concernant la concurrence). Nous avons montré comment les utiliser dans la vérification d'un programme concurrent Ada. Premièrement, nous avons proposé une méthode systématique de translation d'un programme Ada aux ECATNets. Nous nous sommes concentrés sur la translation des éléments syntaxiques du langage Ada relatifs en particulier au concept de 'tâche'. En deuxième point, nous avons développé un compilateur Ada-ECATNets en se basant sur les directions de translation que nous avons proposées [Bou04a]. Finalement, nous avons proposé une représentation très compactée des Ada-ECATNets obtenus [Bou06c], [Bou06d]. En utilisant les réseaux de Petri (ordinaires ou colorés) pour représenter des programmes Ada, les représentations compactées Ada-nets proposés dans la littérature sont dues à l'application des règles de réduction des réseaux de Petri après une translation complète d'un programme Ada vers les réseaux de Petri. Dans notre approche, nous faisons une réduction au cours de translation des programmes Ada vers les ECATNets. De cette façon, la représentation Ada-ECATNets obtenue peut subir une double réduction en appliquant les règles de réduction des réseaux de Petri algébriques que nous avons déjà adaptées et implémentées pour les ECATNets.

Le mémoire est divisé en trois parties. La première partie concerne l'état de l'art. Elle contient une présentation du langage Ada, tout en se concentrant sur le concept de tâche. Cette partie contient aussi un second chapitre sur la logique de réécriture. Ce chapitre précède un troisième consacré aux ECATNets. La seconde partie concerne la présentation de certains algorithmes et outils que nous avons développés pour les ECATNets. Le premier chapitre est consacré au simulateur graphique développé aux ECATNets. Le second chapitre décrit notre analyseur dynamique proposé aux ECATNets. Le dernier chapitre dans cette partie explique notre démarche d'adaptation et d'implémentation des règles de réduction pour les ECATNets. La dernière partie concerne la translation d'un programme Ada vers les ECATNets et l'application des outils pour les ECATNets sur la vérification de ce programme. Dans le premier chapitre, nous décrivons notre approche systématique de translation de certains concepts de langage Ada vers les ECATNets. Le chapitre 2 décrit une représentation compactée des Ada-ECATNets. Le chapitre 3 concerne notre implémentation de la translation de Ada-ECATNets en utilisant le langage Maude. Nous expliquons l'utilisation de quelques outils d'analyse des ECATNets pour l'analyse des programmes Ada dans le chapitre 4.

Partie 1 : Etat de l'Art

La première partie de ma thèse concerne l'état de l'art. Dans cette partie, nous présentons les différents domaines évoqués dans la réalisation de ce travail. Il s'agit de la logique de réécriture, les ECATNets et enfin le langage Ada.

Nous commençons par introduire la logique de réécriture à travers un premier chapitre. Nous introduisons d'abord brièvement la sémantique de la logique de réécriture avant de se concentrer sur le langage Maude. Nous présentons les deux niveaux de description dans Maude. Il s'agit de niveau de spécification et du niveau de vérification des propriétés. Le premier niveau inclut les trois types de modules définis dans Maude. Il s'agit de module fonctionnel pour la description des données, de module système pour la description du comportement concurrent et du module orienté-objet pour une syntaxe plus approprié au paradigme orienté-objet. Le niveau de spécification des propriétés définit, en fait, le concept de la vérification par le biais du Model Checking dans Maude. Nous finissons ce chapitre en décrivant l'exécution sous le système Maude tout en introduisant certaines de ses commandes.

Ce chapitre précède un deuxième consacré aux ECATNets. Nous avons préféré la présentation de la logique de réécriture avant les ECATNets pour que l'intégration des ECATNets dans la logique de réécriture soit facile à comprendre. Dans ce second chapitre, nous abordons l'aspect graphique ainsi que l'aspect textuel des ECATNets. La version textuelle des ECATNest est supportée par la logique de réécriture. Nous présentons aussi dans ce chapitre deux exemples des systèmes réels décrits à l'aide des ECATNets. Le premier exemple est à propos d'un réseau de communication et le second est à propos d'un robot.

Nous finissons cette première partie par un chapitre présentant notre domaine d'application pour certains des outils des ECATNets : les programmes Ada. Ce chapitre décrit brièvement d'abord les concepts de base de langage Ada comme les unités de programmes, les instructions et les types des données. Ensuite, nous entamons en détail le concept de tâche défini dans Ada. A l'aide d'un exemple, nous introduisons les types des tâches, la communication entre les tâches par le biais des rendez-vous ou à travers les types de données partagés.

Chapitre 1 :
Logique de Réécriture et Maude

1. Introduction

La logique de réécriture, dotée d'une sémantique saine et complète, a été introduite par Meseguer [Mes92], [Mes96]. Elle permet de décrire les systèmes concurrents. Cette logique unifie plusieurs modèles formels qui expriment la concurrence. La logique de réécriture est une logique qui permet de raisonner d'une manière correcte sur les systèmes concurrents non-déterministes ayant des états et évoluant en termes de transitions. Elle explique n'importe quel comportement concurrent dans un système avec une sémantique de 'vraie concurrence', par des déductions dans cette logique. Le langage de la logique de réécriture Maude [Mes00], [Cla03] est l'un des langages les plus puissants dans la spécification formelle, la programmation et la vérification des systèmes concurrents.

Dans ce chapitre, nous présentons la logique de réécriture et son langage Maude. Les concepts de cette logique présentés au sein de ce chapitre sont ceux utilisés pour le développement de nos différents outils pour les ECATNets ainsi que les outils d'analyse du langage Ada à l'aide des ECATNets.

Le reste de ce chapitre est organisé comme suit : dans la section 2, nous introduisons la logique de réécriture et sa sémantique. Dans la section 3, nous présentons le langage de la logique de réécriture Maude et ses différents niveaux de spécification. Quelques commandes de Maude sont introduites brièvement dans la section 4. Le concept méta-niveau dans Maude ainsi que la notion de fonction descente, seront expliqués dans la section 5. Dans la section 6, nous citons les plates-formes de Maude utilisées dans le développement de nos différentes applications. Finalement, la section 7 conclut le chapitre.

2. Logique de Réécriture

Dans la logique de réécriture, chaque système concurrent, est représenté par une théorie de réécriture $\Re = (\Sigma, E, L, R)$. Sa structure statique est décrite par la signature (Σ, E), tandis que sa structure dynamique est décrite par les règles de réécriture R. Une conséquence intéressante des définitions de la logique de réécriture est qu'une théorie $\Re = (\Sigma, E, L, R)$ devient une spécification exécutable du système concurrent qu'elle formalise. Dans cette section nous rappelons les définitions de base de la logique de réécriture, pour plus de détails, les lecteurs intéressés peuvent se rapporter à [Mes96], [Mes00]. Une théorie de réécriture $\Re$ est une 4-tuple $\Re = (\Sigma, E, L, R)$, tel que (Σ, E) est une signature ; Σ est un ensemble des sortes et opérations, et E est un ensemble de Σ-équations. La signature (Σ, E) est une théorie équationnelle qui décrit la structure algébrique particulière des états du système (multi-ensemble, arbre binaire, etc...) qui sont distribués selon cette même structure. L'ensemble $R \subseteq L \times (T_{\Sigma,E}(X))^2$ est l'ensemble des pairs tel que le premier composant est une étiquette et le second est un pair des classes d'équivalence des termes modulo les équations E, avec $X = \{x_1,\ldots,x_n,\ldots\}$ un ensemble comptable et infini des variables. Les éléments de R s'appellent les règles conditionnelles de réécriture. Ils décrivent les transitions élémentaires et locales dans un système concurrent. Chaque

règle de réécriture correspond à une action pouvant se produire en concurrence avec d'autres actions. La réécriture opérera les classes d'équivalence des termes, modulo l'ensemble des équations E. Pour une réécriture (r, [t], [t']), ([u_1],[v_1]),…., ([u_k],[v_k])), nous utilisons la notation : r : [t] $\rightarrow$ [t'] if [u_1] $\rightarrow$ [v_1] $\wedge$…$\wedge$ [u_k] $\rightarrow$[v_k], tel que [t] représente la classe d'équivalence du terme t. Une règle r exprime que la classe d'équivalence contenant le terme t a changé à la classe d'équivalence contenant le terme t' si la partie conditionnelle de la règle, [u_1] $\rightarrow$ [v_1] $\wedge$ ….$\wedge$ [u_k] $\rightarrow$ [v_k], est vérifiée. Etant donné une théorie $\Re$, nous disons qu'une séquence r : [t] $\rightarrow$[t'] est prouvable dans $\Re$ ou r : [t] $\rightarrow$[t'] est $\Re$-réécriture concurrente et nous écrivons $\Re$ |- r : [t] $\rightarrow$ [t'] Ssi [t] $\rightarrow$ [t'] est dérivable des règles dans $\Re$ par une application finie des règles de déduction suivantes :

1- Réflexivité. Pour chaque $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \rightarrow [t]}$

2- Congruence. Pour chaque $f \in \sum_n, n \in \mathrm{N}$ $\dfrac{[t_1] \rightarrow [t_1'] \quad \ldots \quad [t_n] \rightarrow [t_n']}{[f(t_1,\ldots,t_n)] \rightarrow [f(t_1',\ldots,t_n')]}$

3- Remplacement. Pour chaque règle de réécriture $r : [t(x_1,\ldots,x_n)] \rightarrow [t'(x_1,\ldots,x_n)]$ dans R,

$$\frac{[w_1] \rightarrow [w_1'] \quad \ldots \quad [w_n] \rightarrow [w_n']}{[t(\bar{w}/\bar{x})] \rightarrow [t'(\bar{w}'/\bar{x})]}$$

Tel que $t(\bar{w}/\bar{x})$ indique la substitution simultanée des w_i pour x_i dans t.

4- Transitivité. $\dfrac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

Une théorie de réécriture est une description statique d'un système concurrent. Sa sémantique est définie par un modèle mathématique qui décrit son comportement. Le modèle pour une théorie de réécriture étiquetée $\Re$ = (Σ, E, L, R), est une catégorie $\tau_{\Re}$(X) dont les objets (états) sont des classes d'équivalence des termes [t] ∈ $T_{\Sigma, E}$ (X) et dont les morphismes (transitions) sont des classes d'équivalence des termes-preuves (proof-terms) représentant des preuves dans la déduction de réécriture.

3. Maude

Maude [Mes00] est un langage de spécification et de programmation basé sur la logique de réécriture. Maude est simple, expressif et performant. Maude offre peu de constructions syntaxiques et une sémantique bien définie. Il est, par ailleurs, possible de décrire naturellement différents types d'applications. Maude est un langage qui supporte facilement le prototypage rapide et représente un langage de programmation avec des performances compétitives. Dans le langage Maude, deux niveaux de spécification sont définis. Un premier niveau concerne la spécification du système tandis que le second est lié à la spécification des propriétés [Cla03], [Cla05].

3.1. Niveau de Spécification de Système

Ce niveau est basé sur la théorie de réécriture. Il est principalement décrit par les modules systèmes. Pour une bonne description modulaire, trois types de modules sont définis dans Maude. Les modules *fonctionnels* permettent de définir les types de données. Les modules *systèmes* permettent de définir le comportement dynamique d'un système.

Enfin, les modules *orienté-objet* qui peuvent, en fait, être réduits à des modules systèmes offrent explicitement les avantages du paradigme objet. Dans le cadre de cette thèse, nous faisons appel uniquement à des modules fonctionnels et systèmes, alors nous écartons la présentation des modules orienté-objet.

Modules Fonctionnels. Les modules fonctionnels définissent les types de données et les opérations qui leur sont applicables en termes de la théorie des équations. L'algèbre initiale est le modèle mathématique (dénotationel) pour les données et les opérations. Les éléments de cette algèbre sont des classes d'équivalence des termes sans variables (ground terms) modulo des équations. Si deux termes sans variables sont égaux par le biais des équations, alors ils appartiennent à la même classe d'équivalence. En utilisant des équations comme des règles de réduction, chaque expression pourra être évaluée à sa forme réduite dite représentation canonique. Cette dernière est unique et représente tous les termes de la même classe d'équivalence. Les équations dans un module fonctionnel sont orientées. Elles sont utilisées de gauche à droite et le résultat final de la simplification d'un terme initial est unique quelque soit l'ordre dans lequel ces équations sont appliquées. En plus des équations, ce type de modules supporte les axiomes d'adhésions (memberships axioms). Ces axiomes précisent l'appartenance d'un terme à une sorte. Cette appartenance peut être sous certaines conditions ou non. Cette condition est une conjonction des équations et des tests d'adhésions inconditionnels. Nous donnons dans ce qui suit un exemple classique d'un type abstrait de données, celui des naturels :

```
fmod NAT is
  sort Nat . sort PositiveNat .
  subsort PositiveNat < Nat .
  op 0 : -> Nat .
  op S_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .   *** N + M = M + N
  vars N M : Nat .
  var P : PositiveNat .
  cmb P : PositiveNat if P =/= 0 .       *** axiome d'adhésion indiquant que P est de type PositiveNat
                                         *** si P est différent de 0
  eq N + 0 = N .                         *** N + 0 et N dénotent le même terme (même classe d'équivalence) qui est [N]
  eq (S N) + (S M) = S S (N + M) .
endfm
```

Modules Systèmes. Les modules systèmes permettent de définir le comportement dynamique d'un système. Ce type de module augmente les modules fonctionnels par l'introduction de règles de réécriture. Un degré maximal de concurrence est offert par ce type de module. Un module système décrit une "théorie de réécriture" qui inclut des sortes, des opérations et trois types d'instructions : équations, adhésions et règles de réécriture. Ces trois types d'instructions peuvent être conditionnels. Une règle de réécriture spécifie une "transition concurrente locale" qui peut se dérouler dans un système. L'exécution de telle transition spécifiée par la règle peut avoir lieu quand la partie gauche d'une règle correspond à (match) une portion de l'état global du système et bien sûr la condition de la règle est valide. Un exemple simple montrant la puissance de ce type de module est donné dans ce qui suit :

```
mod NAT is

extending NAT .

op _? _ : Nat Nat -> Nat .

vars N M : Nat .

rl [R1] : N ? M => N .

rl [R2] : N ? M => M .

endm
```

L'opérateur ? modélise le choix aléatoire entre deux entiers. Si nous avions utilisé uniquement la logique équationnelle pour décrire ce système, alors nous aurions les équations suivantes : N ? M = N et N ? M =M. Ce qui signifie que N = M, c'est-à-dire que tous les entiers sont égaux, alors le modèle dénotationnel (l'algèbre initiale) de ce module s'effondre à un seul élément! Le module NAT n'est pas protégé. La logique équationnelle est faible pour décrire ce système. La logique équationnelle est réversible, alors que l'évolution des systèmes concurrents ne l'est pas forcément. Les règles R1 et R2 sont irréversibles et elles sont concurrentes. Dans ce cas, la sémantique de l'opérateur ? est bien définie.

3.2. Niveau de Spécification de Propriété

Ce niveau de spécification définit les propriétés du système à vérifier. Le système est bien sûr décrit à l'aide d'un module système. En évaluant l'ensemble des états accessibles à partir d'un état initial, le Model Checking permet de vérifier une propriété donnée dans un état ou un ensemble d'états. La propriété est exprimée dans une logique temporelle LTL (Linear Temporel Logic) ou BTL (Branching Temporel Logic). Le Model Checking supporté par la plate-forme de Maude utilise la logique LTL essentiellement pour sa simplicité et les procédures de décision bien définies qu'il offre (pour plus de détails, voir [Cla03] ou [Cla05]). Dans un module prédéfini LTL, on trouve la définition des opérateurs pour la construction d'une formule (propriété) dans la logique temporelle linéaire. Dans ce qui suit, et en écartant certains détails d'implémentation, on trouve une partie des opérateurs LTL dans la syntaxe de Maude:

```
fmod LTL is
…
*** opérateurs définis de LTL
 op _->_ : Formula Formula -> Formula .       *** implication
 op _<->_ : Formula Formula -> Formula .      *** equivalence
 op <>_ : Formula -> Formula .                *** eventualy
 op []_ : Formula -> Formula .                *** always
 op _W_ : Formula Formula -> Formula .        *** unless
 op _|->_ : Formula Formula -> Formula .      *** leads-to
 op _=>_ : Formula Formula -> Formula .       *** strong implication
 op _<=>_ : Formula Formula -> Formula .      *** strong equivalence
…
endfm
```

Les opérateurs LTL sont représentés dans Maude en utilisant une forme syntaxique semblable à leur forme d'origine. Par exemple, l'opération [] est défini dans Maude par l'opérateur (*always*). Cet opérateur s'applique sur une formule pour donner une nouvelle formule. Nous avons, par ailleurs, besoin d'un opérateur indiquant si une formule donnée est vraie ou fausse dans un certain état. Nous trouvons un tel opérateur (|=) dans un module prédéfini appelé SATISFACTION :

```
fmod SATISFACTION is
 protecting LTL .
 sort State .
 op _|=_ : State Formula ~> Bool .
endfm
```

L'état State est générique. Après avoir spécifier le comportement de son système dans un module système de Maude, l'utilisateur peut spécifier plusieurs prédicats exprimant certaines propriétés liées au système. Ces prédicats sont décrits dans un nouveau module qui importe deux modules : celui qui décrit l'aspect dynamique du système et le module SATISFACTION. Soit, par exemple, M-PREDS le nom du module décrivant les prédicats sur les états du système :

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Configuration < State .
  …
endm
```

M est le nom du module décrivant le comportement du système. L'utilisateur doit préciser que l'état choisi (configuration choisie dans cet exemple) pour son propre système est sous type de type State. A la fin, nous trouvons le module MODEL-CHECKER qui offre la fonction model-Check. L'utilisateur peut appeler cette fonction en précisant un état initial donné et une formule. Le Model Checker de Maude vérifie si cette formule est valide (selon la nature de la formule et la procédure du Model Checker adoptée par le système Maude) dans cet état ou l'ensemble de tous le états accessibles depuis l'état initial. Si la formule n'est pas valide, un contre exemple (counterexample) est affiché. Le contre exemple concerne l'état dans lequel la formule n'est pas valide:

```
fmod MODEL-CHECKER is
including SATISFACTION .
…
op counterexample : TransitionList TransitionList -> ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult .
…
endfm
```

4. Exécution de Maude

Nous donnons une vue générale de l'exécution d'une session de Maude sous Windows pour mettre le lecteur dans le contexte (pour plus de détails voir [Cla03] ou [Cla05]). Nous pouvons commencer une session avec Maude en appelant le fichier de commande MS-DOS (même chose pour les autres plates-formes). Nous avons trois fenêtres, mais la principale à la forme qui se trouve dans la figure 1. Le système Maude est maintenant prêt pour accepter des commandes ou des modules. Durant une session l'utilisateur interagit avec le système en saisissant sa requête au 'Maude prompt'. Par exemple, si on veut quitter Maude

```
Maude> quit
```

'q' peut être utilisé comme abréviation pour la commande 'quit'. On peut aussi saisir des modules et utiliser autres commandes. Ce n'est pas vraiment pratique de saisir un module dans le 'prompt', plutôt l'utilisateur peut écrire un ou plusieurs modules dans un fichier texte et faire entrer le fichier dans le système en faisant appel à la commande 'in' ou la commande 'load'. Supposant que le fichier s'appelle my-nat.maude contenant le module NAT décrit au-dessus, alors nous pouvons l'introduire au système en faisant la commande suivante :

```
Maude> in my-nat.maude
```

Après entrer le module NAT nous pouvons par exemple, réduire le terme s s zero + s s s zero (qui correspond à 2 + 3 dans la notation Peano) dans de tel module. Le lancement et la validation de la commande 'reduce' retourne un résultat comme suit :

```
Maude> reduce in NAT : s s zero + s s s zero .
reduce in NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s s s s zero
```

Ce n'est pas nécessaire de donner le nom du module explicitement dans lequel on réduit le terme. Toutes les commandes dont un module a besoin référent au module en cours par défaut, sinon il faut donner explicitement le nom du module. Le module en cours est en général le dernier introduit ou utilisé, ou bien nous pouvons utiliser la commande 'select' pour sélectionner un module pour être le module en cours.

```
Maude> reduce s s zero + s s s zero .
reduce in NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s s s s zero
```

En général, les termes sont plus longs que ce simple exemple. Il est préférable d'écrire la commande dans le fichier contenant le module. La requête doit être introduite après le module, mais à l'extérieur de tout module.

Maintenant, nous introduisons quelques commandes de Maude que nous avons utilisé beaucoup pendant ce travail. Il s'agit de 'reduce', 'rewrite', 'continue' et 'search'.

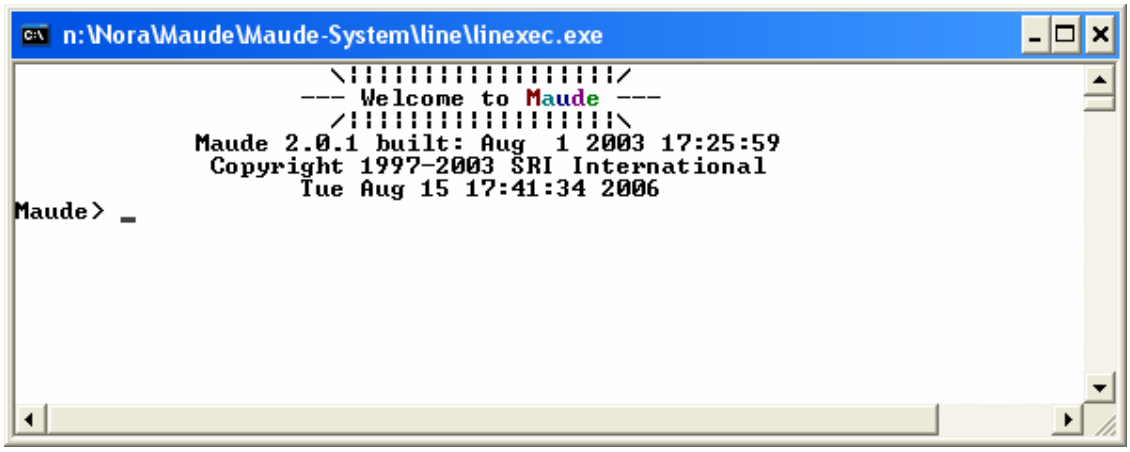

Figure 1. Exécution de Maude

Maude possède une librairie standard des modules prédéfinis qui, par défaut, sont chargés par le système automatiquement au début de chaque session. Chacun de ses modules prédéfinis peut être importé par un autre module défini par l'utilisateur. Ces modules prédéfinis se trouvent dans des fichiers au même répertoire que le fichier exécutable de Maude. Parmi ces fichiers, on trouve prelude.maude et model-checker.maude. En fait, le premier fichier contient plusieurs modules prédéfinis. Nous pouvons trouver comme exemples de modules prédéfinis dans ce fichier : BOOL, STRING et NAT. Ces modules déclarent les sortes et les opérations pour manipuler respectivement les valeurs booléennes, les chaînes de caractères et les nombres naturels. Le fichier model-checker.maude contient des modules prédéfinis déjà introduits dans la section 3.2. Ces modules contiennent les outils nécessaires pour l'utilisation du Model Checker de Maude.

4.1. Commande 'reduce'

Cette commande permet au terme désigné 'term' d'être réduit par le biais des équations et les axiomes d'adhésion dans le module donné 'module'. 'reduce' peut être abrévié à 'red'. Si la clause 'in' est omise, le système prend en considération le module en cours :

reduce {in module :} term .

4.2. Commandes 'rewrite' et 'continue'

Cette commande permet au terme 'term' d'être réécrit en utilisant les règles, les équations et les axiomes d'adhésion dans le module donné. L'interpréteur par défaut applique les règles en utilisant une stratégie top-down (paresseuse ou 'lazy') et il s'arrête si le nombre des applications des règles atteint la borne considérée 'bound'. Aucune règle de réécriture ne peut être appliquée si une équation peut être appliquée. Si la clause 'in' est omise, alors le système considère le module en cours. Si la clause 'bound' est omise, l'infinité est envisageable. 'rewrite' peut être elle aussi abréviée à 'rew' :

rewrite {[bound]} {in module :} term .

Une commande 'rewrite' peut être suivie par une commande 'continue'. La commande 'continue' (qui peut être abréviée à 'cont') permet le système Maude de continuer la réécriture du terme (qui a été réécrit par la dernière

‘rewrite’) en utilisant un nombre ‘number’ au plus des règles de réécriture. Une commande ‘continue’ sans le paramètre ‘number’ permet de continuer la réécriture du terme jusqu’à il n’ y aura plus de règles à appliquer :
continue {number} .

4.3. Commande ‘search’

La commande search performe une recherche dans les calculs de la commande ‘rewrite’. Cette recherche débute du terme ‘subject’ jusqu’à un état final qui correspond à ‘pattern’ et qui satisfait une condition optionnelle :
search {[bound]} {in module :} subject searchtype pattern { such that condition} .

Les valeurs possibles pour ‘searchtype’ sont :
=>1 un pas de calcul
=>+ un ou plusieurs pas de calcul
=>* zéro ou plusieurs pas de calcul
=>! Seulement les états finaux sous forme canonique sont permis

Le nombre maximal des solutions trouvées est borné mais il peut s’étendre à l’infini. Pour afficher le graphe généré par ‘search’ il faut saisir la requête suivante :
show search graph .

5. Calcul Méta-niveau dans Maude

La description méta-niveau est l’un des services offerts par Maude. Ce service permet de décrire une méta-représentation d’un module le rendant ainsi une entrée pour un autre module. Nous utiliserons par la suite la méta-représentation dans Maude pour décrire un ECATNet qui devient une entrée pour notre outil. La syntaxe de méta-représentation est différente de la représentation ordinaire dans Maude. Les termes et les modules dans le méta-niveau sont appelés méta-terme et méta-module respectivement. Un méta-terme (resp. méta-module) est considéré comme un terme d’un type générique dit Term (resp. Module). Pour manipuler un module dans le méta-niveau, Maude fournit le module prédéfini META-LEVEL. Ce module comprend certains services appelés les fonctions descentes (descent functions). Les fonctions descentes performent dans le méta-niveau, la réduction et la réécriture d’un méta-terme, selon les équations et les règles dans le méta-module correspondant.

5.1. Meta-Représentation dans Maude

La méta-représentation d’un module rend ce module une entrée pour un autre module. Le méta-module est considéré comme un terme générique appelé Module. L’utilisateur n’est pas obligé d’écrire son ECATNet dans une méta-représentation. Il peut l’écrire dans le mode ordinaire, puis il utilise la fonction de Maude upModule qui permet de transformer la représentation d’un module à sa méta-représentation. Le passage dans l’autre sens aussi est possible grâce à la fonction downModule. Pour plus de clarité, nous présentons le module représentant l’ECATNet précédent dans sa méta-représentation. Dans le module META-LEVEL-ROBOT-ECATNET-SYSTEM, le module META-ROBOT est défini autant qu’une constante de type Module et son contenu est décrit à l’aide d’une équation. ‘GENERIC-ECATNET est la description en méta-niveau de module GENERIC-ECATNET décrit précédemment.

5.2. Fonctions de Descente dans Maude

Maude offre à l'utilisateur plusieurs fonctions de descente prédéfinies pour le traitement de son méta-module. Nous décrivons dans cette section quelques unes de ces fonctions de descente. Ces fonctions ont été utilisées dans le cadre de développement des outils de notre thèse.

Fonction metaMatch. C'est le processus de faire la correspondance (matching) à la racine entre deux termes :
op metaMatch : Module Term Term Condition Nat –> Substitution? .
L'opération metaMatch(R, t, t', Cond, n) cherche à correspondre à la racine les termes t et t' dans le module R de telle façon que la substitution résultante satisfait la condition Cond. Le nombre naturel n est utilisé pour indiquer le nombre possible des correspondances. Dans le cas succès, cette fonction retourne une substitution, sinon elle retourne noMatch.
De point de vue théorique, soit TΣ : Σ–algèbre des termes sans variables (ground terms) dans la signature Σ et TΣ(X) : algèbre des termes qui peuvent avoir des variables dans un ensemble des variables X. Etant donné un $t \in T\Sigma(X)$, correspondant à la partie gauche d'une équation orientée et un terme sans variables $t' \in T\Sigma$, on dit que t correspond (matchs) t' s'il existe une substitution σ telle que $\sigma(t) \equiv t'$, donc, σ(t) et t' sont syntaxiquement des termes égaux.

Fonctions de sélection. Ces fonctions permettent d'extraire des parties d'un module. Par exemple, la fonction de sélection getRls prend une méta-représentation d'un module et retourne la méta-représentation des règles de réécriture. Les opérateurs utilisés pour la méta-représentation des ensembles des règles sont :

```
sorts Rule RuleSet .
subsort Rule < RuleSet .
op rl_=>_[_] : Term Term AttrSet  –> Rule [ctor] .
op crl_=>_if_[_] . : Term Term Condition AttrSet  –> rule [ctor] .
op none : -> RuleSet [ctor] .
op __ : RuleSet RuleSet  –> RuleSet [ctor assoc comm id: none] .
op getRls : Module –> RuleSet .
```

Fonction metaApply. metaApply est le processus d'appliquer une règle de réécriture d'un module système à un terme:

```
sort ResultTriple ResultTriple? .
subsort ResultTriple < ResultTriple? .
op {_,_,_} : Term Type Substitution –> ResultTriple [ctor] .
op failure : –> ResultTriple? [ctor] .
op metaApply : Module Term Qid Substitution Nat –> ResultTriple? .
```

Les quatre premiers paramètres sont des représentations dans le méta-niveau d'un module, un terme dans un module, quelques étiquettes de règles dans le module et un ensemble des affectations (potentiellement vide) définissant une substitution partielle pour des variables dans cette règle. Le dernier paramètre est un nombre naturel. Soit N le cinquième paramètre, la fonction metaApply retourne le résultat $(N+1)^{th}$ de l'application de chaque substitution. Dans nos applications, nous n'avons pas besoin des substitutions, alors nous considérons que la substitution vide none. Dans ce cas, nous prenons 0 comme dernier paramètre. Pour plus de détails à propos des deux derniers paramètres, voir

[Cla03] ou [Cla05]. La fonction metaApply retourne un triple formé du terme résultat, le type de ce terme et une substitution, autrement metaApply retourne 'failure'.

Fonction getTerm. Nous appliquons la fonction getTerm sur le résultat de metaApply pour extraire le terme résultat :

op getTerm : ResultTriple –> Term .

6. Plates Formes de Maude Utilisées

Dans le cadre de ce travail, nous avons utilisé comme plate-forme plus qu'une seule version de Maude. Pour la simplicité, nous avons utilisé la version 2.0.1 de Maude sous Windows dans le développement de certains outils. Cependant, le développement d'autres outils a nécessité l'utilisation des versions de Maude sous Linux. Par exemple, l'implémentation de certaines règles de réduction a nécessité l'utilisation d'un outil dit 'Sufficient Completeness Checker' crée par Hendrix dans [Hen05] qui ne fonctionne que sur le système Maude sous Linux. Cependant, toutes les applications développées avec Maude sous Windows fonctionnent normalement avec Maude sous Linux. Enfin, nous pouvons dire que tous nos outils s'exécutent normalement sous la version 2.0.1 ou la version 2.1.1 de Maude sous Linux. Notons que le système Maude est toujours en pleine évolution. De nouvelles versions sont en cours de développement.

7. Conclusion

Dans ce chapitre, nous avons introduit la logique de réécriture et son langage Maude. Les concepts de cette logique présentés dans le cadre de ce chapitre sont ceux utilisés dans la réalisation du travail de thèse. La logique de réécriture présente plusieurs concepts théorique et pratique facilitant une description fiable d'un système concurrent. Maude supporte la modularité, la description des données par le biais des modules fonctionnels, la description des systèmes concurrents par le biais des modules systèmes avec un degré très élevé de concurrence, la description des objets avec la concurrence inter et intra objets et la vérification basé Model Checking par une simple extension décrivant les propriétés du module système. Un autre concept très important présenté par la logique de réécriture et supporté aussi pratiquement par son langage Maude est celui du méta-calcul : le pouvoir de cette logique de s'exprimer en elle-même. Ce concept permet de décrire un module qui sera une donnée pour un autre module. Ceci sert à modifier librement la structure de ce module 'donnée'. Nous verrons par la suite comment ces concepts ont été de grande utilité dans le développement des outils propres aux ECATNets et des outils pour l'analyse d'Ada à l'aide des ECATNets.

Chapitre 2 : ECATNets

1. Introduction

Les ECATNets (Extended Concurrent Algebraic Term Nets) [Bet93] sont une catégorie des réseaux de Petri algébriques (RPAs) basés sur une combinaison saine des réseaux de Petri de haut niveau et des types abstraits algébriques. La sémantique des ECATNets est définie en termes de la logique de réécriture [Mes96]. Motivant les ECATNets nous, conduit à motiver les réseaux de Petri, les types abstraits algébriques ainsi que leur combine dans un seul cadre. Les réseaux de Petri sont utilisés pour leur pouvoir d'exprimer la concurrence et l'aspect dynamique et les types abstraits algébriques sont utilisés pour leur puissance de décrire abstraitement les données. Leur association dans un cadre uni est motivée par le besoin de spécifier explicitement le comportement des processus et les structures des données. Autant que des réseaux de Petri, les ECATNets fournissent un formalisme rapidement compris dû à leur description graphique. En outre, les ECATNets ont une théorie puissante et des outils de développement basés sur une logique ayant une sémantique saine et complète. De telle sémantique offre une base solide et rigoureuse pour toute démarche de vérification des propriétés des systèmes décrits.

Le reste de ce chapitre est organisé comme suit : La section 2 est une présentation générale des ECATNets et de leur description dans la logique de réécriture. Dans la section 3, nous donnons l'implémentation des ECATNets dans Maude. Un premier exemple d'un ECATNet et sa description dans Maude, sont donnés dans la section 4. Cet exemple est à propos d'un réseau de communication. Un second exemple d'un ECATNet et son implémentation dans Maude sont décrits dans la section 5. Cet exemple concerne le comportement d'un robot. Finalement, la section 6 conclut l'article.

2. ECATNets

Les ECATNets [Bet93] sont une catégorie de modèle de réseau/données combinant les forces des réseaux de Petri avec ceux des types de données abstraits. Les places sont identifiées par des multi-ensembles des termes algébriques. Les arcs entrants de chaque transition t, c.-à-d. (p, t), sont marqués par deux inscriptions IC(p, t) (Input Conditions) et DT(p, t) (Destroyed Tokens). Les arcs sortants de chaque transition t, c.-à-d. (t, p'), sont marqués par CT(t, p') (Created Tokens). Finalement, chaque transition t est marquée par une inscription TC(t) (Transition Condition) (voir la figure 1). IC(p, t) indique une condition qui valide une transition t, DT(p, t) indique le marquage (un multi-ensemble) qui doit être enlevé de p lors du franchissement de t et CT(t, p') indique le marquage (un multi-ensemble) qui doit être ajouté à p' lors du franchissement de t. En conclusion, TC(t) représente un terme booléen qui indique une condition additionnelle pour le franchissement de la transition t. L'état courant d'un ECATNet est donné par l'union des termes ayant la forme (p, M(p)). Par exemple, soit un réseau contenant une transition t ayant une place d'entrée p marquée par le multi-ensemble $a \oplus b \oplus c$ et une place de sortie vide p', l'état distribué s de ce réseau est donnée par le multi-ensemble : $s = (p, a \oplus b \oplus c)$.

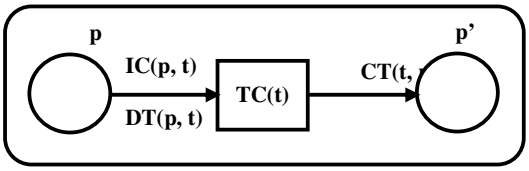

Figure 1. Un ECATNet générique

Une transition t est franchissable quand des diverses conditions sont simultanément vraies. La première condition est que chaque IC(p, t) pour chaque place d'entrée p est tirable. La deuxième condition est que TC(t) est vraie. Enfin l'addition de CT(t, p') à chaque place de sortie p' ne doit pas avoir comme conséquence d'excéder sa capacité quand cette capacité est finie. Si t est franchise alors DT(p, t) est complètement enlevé (cas positif) de la place d'entrée p et simultanément CT(t, p') est ajouté à la place de sortie p'. Notons que dans le cas non positif, on enlève les éléments en commun entre DT(p, t) et M(p). Le franchissement d'une transition et les conditions de ce franchissement sont formellement exprimés par des règles de la logique de réécriture. Les notions de base de cette logique dans le cadre de leur utilisation dans les ECATNets sont résumées comme suit. Les axiomes sont en réalité des règles de réécriture conditionnelles décrivant les effets des transitions comme des types élémentaires de changement. Les règles de déduction nous permettent d'avoir des conclusions valides des ECATNets à partir des changements. Une règles de réécriture est de la forme est de la forme "t: u $\rightarrow$ v if boolexp"; où u et v sont respectivement les côtés gauche et droit de la règle, t est la transition liée à cette règle, et boolexp est un terme booléen. Plus précisément, u et v sont des multi-ensembles des pairs de la forme (p, $[m]_{\oplus}$), telle que p est une place de réseau, $[m]_{\oplus}$ est un multi-ensemble des termes algébriques, et l'union multi-ensemble sur ces termes. Notons que les termes sont considérés comme singletons. L'union des multi-ensembles sur les pairs ((p, $[m]_{\oplus}$) est dénoté par $\otimes$. $[x]_{\otimes}$ dénote la classe d'équivalence de x, selon les axiomes ACI (Associativity Commutativity Identity = ϕ_M) pour $\otimes$. Un état d'un ECATNet lui-même est représenté par un multi-ensemble de telles paires. Nous rappelons maintenant les formes des règles de réécriture (c.-à-d., les méta-règles) à associer avec les transitions d'un ECATNet donné. Soit R un ensemble de règles de réécriture (définissant tous les types élémentaires de changement), R valide une séquence s $\rightarrow$ s' (définissant un changement global d'un état à un autre) ssi s $\rightarrow$ s' peut être obtenu par des applications finies et concurrentes des règles de déduction suivantes : Réflexivité (Reflexivity), Congruence (Congruence), Remplacement (Replacement), Décomposition (Splitting), Recombinaison (Recombination) et l'Identité (Identity). Nous rappelons que la règle de réflexivité décrit que chaque état se transforme en lui-même. La règle de congruence que les changements élémentaires doivent correctement propagés. La règle de remplacement est utilisée quand les instanciations de variables deviennent nécessaires. Les règles de décomposition et de recombinaison nous permettent, en décomposant et recombinant "judicieusement" les multi-ensembles des classes d'équivalences des termes, pour détecter les calculs montrant un maximum de concurrence. Nous rappelons maintenant les formes des règles de réécriture (i.e, les méta-règles) à associer avec les transitions d'un ECATNet donné.

2.1. Formes des Règles de Réécriture : Cas Positif sans Arcs Inhibiteurs

IC(p,t) est de la forme $[m]_{\oplus}$

Cas 1. $[IC(p, t)]_{\oplus} = [DT(p, t)]_{\oplus}$

La forme de cette règle est : t : (p, $[IC(p ,t)]_{\oplus}$) $\rightarrow$ (p', $[CT(t, p')]_{\oplus}$)

Cas 2. $[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} = \phi_M$

Cette situation correspond à vérifier que IC(p, t) est inclus dans M(p) et, retirer DT(p,t) de M(p). La forme de cette règle est : $t : (p, [IC(p, t)]_{\oplus}) \otimes (p, [DT(p, t)]_{\oplus}) \rightarrow (p, [IC(p, t)]_{\oplus}) \otimes (p', [CT(t, p')]_{\oplus})$

Cas 3. $[IC(p, t)] \cap [DT(p, t)] \neq \phi_M$

Cette situation correspond au cas le plus général. Elle peut cependant être résolue d'une manière élégante en remarquant qu'elle pourrait être apportée aux deux cas précédents. Ceci est réalisé en remplaçant la transition t de ce cas par deux transitions t1 et t2. Ces transitions, une fois elles sont franchises concurremment, donnent même effet global que notre transition. Nous éclatons IC(p, t) en deux multi-ensembles IC1(p, t1) et IC1(p, t2). Nous éclatons également DT(p, t) en deux multi-ensembles DT1(p, t1) et DT1(p, t2) :

$IC(p, t) = IC1(p, t1) \cup IC1(p, t2)$, $DT(p, t) = DT1(p, t1) \cup DT1(p, t2)$.

Les quatre multi-ensembles obtenus doivent réaliser $IC1(p, t1) = DT1(p, t1)$ et $IC2(p, t2) \cap DT2(p, t2) = \phi_M$. Le franchissement de la transition t est identique au franchissement en parallèle des deux transitions t1 et t2.

2.2. Formes des Règles de Réécriture : Cas Général

IC(p,t) est de la forme $[m]_{\oplus}$

Cas 1. $[IC(p,t)]_{\oplus} = [DT(p,t)]_{\oplus}$

La forme de cette règle est : $t : (p, [IC(p, t)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$

Cas 2. $[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} = \phi_M$

Cette situation correspond à vérifier que IC(p, t) est inclus dans M(p) et, retirer DT(p, t) de M(p). La forme de cette règle est : $t : (p, [IC(p, t)]_{\oplus}) \otimes (p, [DT(p, t)]_{\oplus}) \rightarrow (p, [IC(p, t)]_{\oplus}) \otimes (p', [CT(t, p')]_{\oplus})$

Cas 3. $[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} \neq \phi_M$

Ce cas peut être résolu aussi en combinant les deux cas précédents. Ceci peut se faire en remplaçant la transition de ce cas par deux transitions, une fois franchises concurremment, donne le même effet global comme cette transition. En réalité, ce remplacement montre comment spécifier une situation donnée en deux niveaux d'abstraction.

IC(p, t) est de la forme ˜$[m]_{\oplus}$

La forme de cette règle est:

$t : (p, [DT(p, t)]_{\oplus} \cap [M(p)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$ if $([IC(p, t)]_{\oplus} \setminus ([IC(p, t)]_{\oplus} \cap [M(p)]_{\oplus})) = \phi_M \rightarrow$ [false]

IC(p, t) = empty

La forme de cette règle est de la forme :

$t : (p, [DT(p, t)]_{\oplus} \cap [M(p)]_{\oplus}) \rightarrow (p', [CT(t, p')]_{\oplus})$ if $[M(p)]_{\oplus} \rightarrow \phi_M$

Telle que la capacité C(p) est finie, la partie conditionnelle de règle de réécriture va inclure le composant suivant :

$[CT(p, t)]_{\oplus} \oplus [M(p)]_{\oplus} \cap [C(p)]_{\oplus} \rightarrow [CT(p, t)]_{\oplus} \oplus [M(p)]_{\oplus}$ (**Cap**)

Dans le cas où il y a une condition de transition TC(t), la transition est représentée dans la logique de réécriture par une règle non conditionnelle.

3. Implémentation des ECATNets dans Maude

Considérons la version textuelle des ECATNets dans Maude. Le module générique qui décrit des opérations de base d'un ECATNet est le suivant :

```
fmod GENERIC-ECATNET is
  sorts Place Marking GenericTerm .
  op mt : -> Marking .
  op <_;_> : Place GenericTerm -> Marking .
  op _._ : Marking Marking -> Marking [assoc comm id: mt] .
endfm
```

Comme illustré dans ce code, mt dénote le marquage vide dans un ECATNet. Nous définissons l'opération "<_;_>" qui permet la construction des marquages élémentaires. Les sous-lignes dans cette définition indiquent les positions des paramètres. Le premier paramètre de cette opération est une place et le second est un terme algébrique quelconque qui peut être dans cette place. Pour mettre un terme 0 par exemple de sorte Nat dans une place p, il faut préciser dans le code subsort Nat < GenericTerm, c.-à-d., que Nat est sous-sorte de GenericTerm. Dans ce cas, le terme < p ; 0 > est valide. Nous n'avons pas défini une opération qui implémente ⊕. L'opération "_._" implémentant l'opération ⊗ est suffisante grâce au concept de décomposition. Si une place contient plusieurs termes, par exemple (p, a ⊕ b), alors on peut l'écrire < p ; a > . < p ; b >.

Pour la méta-représentaion des ECATNets, notons que l'utilisateur n'est pas obligé d'écrire son ECATNet dans une méta-représentation. Il peut l'écrire dans le mode ordinaire, puis il utilise la fonction de Maude upModule qui permet de transformer la représentation d'un module à sa méta-représentation. Le passage dans l'autre sens aussi est possible grâce à la fonction downModule.

4. Exemple 1 : Réseau de Communication

Nous considérons un exemple dans [Bet93] à propos d'un réseau de communication qui relie des messages émetteurs aux messages récepteurs. D'abord, nous présentons l'ECATNet décrivant cet exemple. Puis, nous donnons la méta-représentation de l'exemple dans le langage Maude.

4.1. Présentation de l'Exemple

Cet exemple est à propos d'un réseau de communication qui relie des messages émetteurs M aux messages récepteurs N. Chaque émetteur (respectivement récepteur) est connecté à un port de réseau. Chaque groupe d' émetteurs (ou/et de récepteurs) envoie des messages en parallèle. Les places, les transitions et les inscriptions d'arc sont comme suit :

Places : R_1, R_2, S_1,..,S_m, $Queue_1$,.., $Queue_n$, Adr_1,.., Adr_n

Transitions : From-S_1,.., From-S_1, To-R_1,.., To-R_n, Check-Adr_1,.., Check-Adr_n

Inscriptions des arcs : Nous utilisons la définition en termes de spécification algébrique de la file d'attente (queue) : q est une variable de type queue. front(q) est une fonction qui retourne le message m qui est en tête de la file q. addq(m,

q) est une fonction qui ajoute le message à la fin de q. remove(q) est une fonction qui retourne le reste de la file q après supprimer le premier message (en tête de q).

4.2. Le Modèle ECATNet de l'Exemple

La figure 2 présente un ECATNet du problème du routeur.

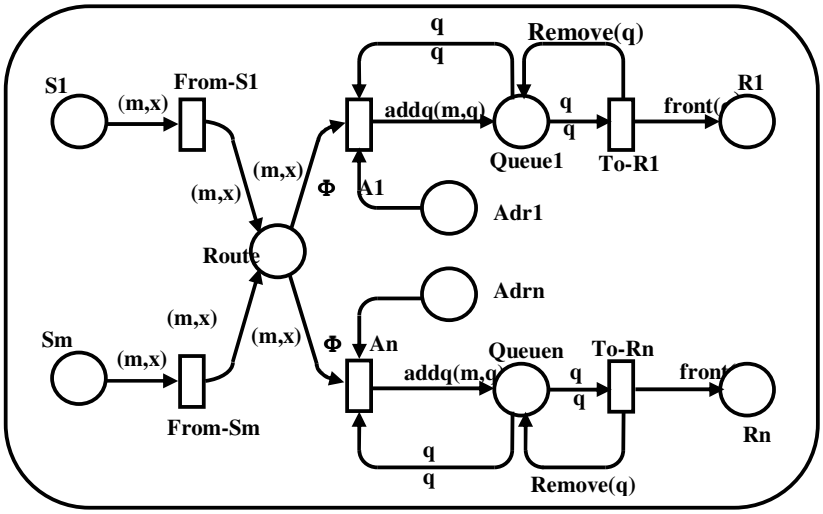

Figure 2. ECATNet modélisant le problème du routeur

4.3. Meta–Représentation de l'Exemple dans Maude

Pour une raison de simplicité, nous prenons un petit nombre des messages émetteurs et récepteurs (M = N = 3). Pour éviter la redondance, nous donnons seulement la méta-représentation de l'exemple. A l'intérieur du module META-LEVEL-ROUTER-ECATNET-SYSTEM, nous définissons un module constant META-ROUTER. Ce module est présenté en utilisant le style méta-niveau :

```
mod META-LEVEL-ROUTER-ECATNET-SYSTEM is
op META-ROUTER : -> Module .
...
eq META-ROUTER = (mod 'META-ROUTER is
including 'META-ECATNET . including 'META-QUEUE . including 'INT . including 'BOOL .

sorts 'Adress ; 'Message .
subsort 'Message < 'Exp . subsort 'Adress < 'Exp . subsort 'Queue < 'GenericTerm . subsort 'Exp < 'Queue .
op 'route : nil -> 'Place [ctor] . op 'R1 : nil -> 'Place [ctor] .op 'R2 : nil -> 'Place [ctor] . op 'R3 : nil -> 'Place [ctor] .
op 'S1 : nil -> 'Place [ctor] . op 'S2 : nil -> 'Place [ctor] . op 'S3 : nil -> 'Place [ctor] .
op 'Queue1 : nil -> 'Place [ctor] .  op 'Queue2 : nil -> 'Place [ctor] .op 'Queue3 : nil -> 'Place [ctor] .
op 'Adr1 : nil -> 'Place [ctor] . op 'Adr2 : nil -> 'Place [ctor] . op 'Adr3 : nil -> 'Place [ctor] .
op 'm1 : nil -> 'Message [ctor] . op 'm2 : nil -> 'Message [ctor] .op 'm3 : nil -> 'Message [ctor] .
op 'A1 : nil -> 'Adress [ctor] . op 'A2 : nil -> 'Adress [ctor] . op 'A3 : nil -> 'Adress [none] .
op '_;_ : 'Exp 'Exp -> 'Exp [none] .
none  none
```

```
rl  '<_;_>['S1.Place, '_;_['m:Message, 'x:Adress]] => '<_;_>['route.Place, '_;_['m:Message, 'x:Adress]] [label('FromS1)] .
rl  '<_;_>['S2.Place, '_;_['m:Message, 'x:Adress]] => '<_;_>['route.Place, '_;_['m:Message, 'x:Adress]] [label('FromS2)] .
rl  '<_;_>['S3.Place, '_;_['m:Message, 'x:Adress]] => '<_;_>['route.Place, '_;_['m:Message, 'x:Adress]] [label('FromS3)] .

crl  '<_;_>['Queue1.Place, 'q:Queue]
      => '_._['<_;_>['Queue1.Place, 'remove['q:Queue]], '<_;_>['R1.Place, 'front['q:Queue]]]
          if '_==_['isempty['q:Queue], 'false.Bool] = 'true.Bool [label('ToR1)] .
crl  '<_;_>['Queue2.Place, 'q:Queue]
      => '_._['<_;_>['Queue2.Place, 'remove['q:Queue]], '<_;_>['R2.Place, 'front['q:Queue]]]
          if '_==_['isempty['q:Queue], 'false.Bool] = 'true.Bool [label('ToR2)] .
crl  '<_;_>['Queue3.Place, 'q:Queue]
        => '_._['<_;_>['Queue3.Place, 'remove['q:Queue]], '<_;_>['R3.Place, 'front['q:Queue]]]
            if '_==_['isempty['q:Queue], 'false.Bool] = 'true.Bool [label('ToR3)] .

crl  '_._['<_;_>['Queue1.Place, 'q:Queue], '_._['<_;_>['route.Place, '_;_['m:Message, 'x:Adress]],
     '<_;_>['Adr1.Place, 'A1.Adress]]]
      => '_._['<_;_>['Adr1.Place, 'A1.Adress], '<_;_>['Queue1.Place, 'addq['m:Message, 'q:Queue]]]
          if '_==_['x:Adress, 'A1.Adress] = 'true.Bool [label('CheckAdr1)] .

crl  '_._['<_;_>['Queue2.Place, 'q:Queue], '_._['<_;_>['route.Place, '_;_['m:Message, 'x:Adress]],
     '<_;_>['Adr2.Place, 'A2.Adress]]]
      => '_._['<_;_>['Adr2.Place, 'A2.Adress], '<_;_>['Queue2.Place, 'addq['m:Message, 'q:Queue]]]
          if '_==_['x:Adress, 'A2.Adress] = 'true.Bool [label('CheckAdr2)] .

crl  '_._['<_;_>['Queue3.Place, 'q:Queue], '_._['<_;_>['route.Place, '_;_['m:Message, 'x:Adress]],
     '<_;_>['Adr3.Place, 'A3.Adress]]]
 => '_._['<_;_>['Adr3.Place, 'A3.Adress], '<_;_>['Queue3.Place, 'addq['m:Message, 'q:Queue]]]
      if '_==_['x:Adress, 'A3.Adress] = 'true.Bool [label('CheckAdr3)] .
endm) .
```

5. Exemple 2 : Robot

L'objectif de cette section est l'application du programme précédent en Maude sur un exemple simple. Cet exemple est présenté dans [Lin95] et il est décrit en utilisant le formalisme des ECATNets dans [Mao97]. Nous reprenons cette description avec quelques modifications.

5.1. Présentation de l'Exemple

L'exemple est à propos d'une cellule de production qui fabrique des pièces forgées de métal à l'aide d'une pression. Cette cellule se compose de table A qui sert à alimenter la cellule par les pièces brutes, d'un robot de la manipulation, d'une pression et d'une table B qui sert au stockage des pièces forgées. Le robot inclut deux bras, disposés perpendiculairement sur un même plan horizontal, interdépendant d'un même axe de rotation et sans possibilité verticale de mobilité. La figure 3 représente la disposition spatiale des éléments de la cellule. Le robot peut saisir une pièce brute de la table A et le déposer dans la pression à l'aide du bras 1. Il peut également saisir une pièce forgée de la pression et peut la déposer sur la table du stockage B à l'aide du bras 2. En bref, le robot peut faire deux mouvements de rotation.

Le premier lui permet de passer de sa position initiale à sa position secondaire. Ce mouvement permet au robot de déposer une pièce brute dans la pression et probablement celui d'une pièce forgée sur la table du stockage B. Le second lui permet de passer de sa position secondaire vers sa position initiale et de poursuivre le cycle de la rotation.

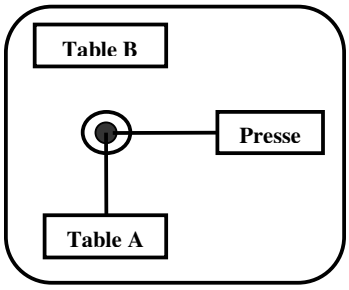

Figure 3. Cellule de Production

5.2. Le Modèle ECATNet de l'Exemple

La figure 4 représente le modèle ECATNet de la cellule. Le symbole ϕ est utilisé pour dénoter le multi-ensemble vide dans les inscriptions des arcs. Veuillez noter que r dénote 'raw' (brute) et f dénote 'forge' (forgée). Si les inscriptions IC(p, t) et DT(p, t) sont identiques, alors nous présentons uniquement IC(p, t) sur l'arc (p, t). Les règles de réécriture du système seront présentées dans la section suivante directement dans Maude.

Les places de l'ECATNet.

Ta : table A ; ensemble, potentiellement vide, des pièces brutes.

Tb : table B ; ensemble, potentiellement vide, des pièces forgées.

Ar1 : bras 1 de robot ; au plus une pièce brute.

Ar2 : bras 2 de robot ; au plus une pièce forgée.

Pr : presse ; au plus une pièce brute ou une pièce forgée.

PosI : position initiale du robot ; elle est marquée "ok" si elle est la position courante du robot.

PosS : position secondaire du robot ; elle est marquée "ok" si elle est la position courante du robot.

EA : cette place a été ajoutée pour tester si les deux bras du robot sont vides.

Les transitions de l'ECATNet.

T1 : prise d'une pièce brute par le bras 1 du robot.

T2 : prise d'une pièce forgée par le bras 2 du robot.

D1 : dépôt d'une pièce brute dans la presse.

D2 : dépôt d'une pièce forgée sur la table B.

TS1, TS2 : rotation du robot de la position initiale vers la position secondaire.

TI : rotation du robot de la position secondaire vers la position initiale.

F : forge de la pièce brute introduite dans la presse.

E : dépôt d'une pièce brute sur la table A.

R : retire des pièces forgées de la table B.

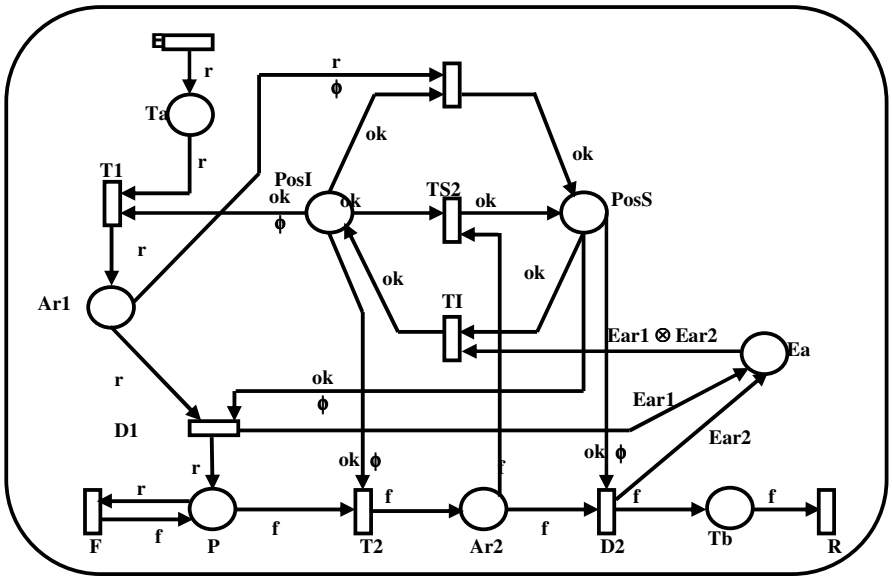

Figure 4. Modèle ECATNet du robot

5.3. Méta-représentation de l'Exemple Dans Maude

Pour plus de clarté, nous présentons le module représentant l'ECATNet précédent dans sa méta-représentation. Dans le module META-LEVEL-ROBOT-ECATNET-SYSTEM, le module META-ROBOT est défini autant qu'une constante de type Module et son contenu est décrit à l'aide d'une équation. 'GENERIC-ECATNET est la description en méta-niveau de module GENERIC-ECATNET décrit précédemment :

```
mod META-LEVEL-ROBOT-ECATNET-SYSTEM is
...
op META-ROBOT : -> Module .
eq META-ROBOT = (mod 'META-ROBOT is
protecting 'GENERIC-ECATNET .
protecting 'INT .
sorts 'Cointype ; 'RPosType ; 'EmptyArmType  .
subsort 'Cointype < 'GenericTerm .
subsort 'RPosType < 'GenericTerm .
subsort 'EmptyArmType < 'GenericTerm .
subsort 'Place < 'Marking .
op 'ok : nil -> 'RPosType [ctor] . op 'Ear1 : nil -> 'EmptyArmType [ctor] . op 'Ear2 : nil -> 'EmptyArmType [ctor] .
op 'raw : nil -> 'Cointype [ctor] . op 'forge : nil -> 'Cointype [ctor] .
op 'Ta : nil -> 'Place [ctor] . op 'Tb : nil -> 'Place [ctor] .
op 'Pr : nil -> 'Place [ctor] . op 'Ar1 : nil -> 'Place [ctor] . op 'Ar2 : nil -> 'Place [ctor] .
op 'PosI : nil -> 'Place [ctor] . op 'PosS : nil -> 'Place [ctor] . op 'Ea : nil -> 'Place [ctor] .
none none

rl '_._['<_;_>['PosI.Place, 'ok.RPosType], '<_;_>['Ta.Place, 'raw.Cointype]]
        => '_._['<_;_>['Ar1.Place, 'raw.Cointype], '<_;_>['PosI.Place, 'ok.RPosType]] [label('T1)]  .
rl '_._['<_;_>['PosI.Place, 'ok.RPosType], '<_;_>['Pr.Place, 'forge.Cointype]]
```

```
        => '_._['<_;_>['Ar2.Place, 'forge.Cointype], '<_;_>['PosI.Place, 'ok.RPosType]] [label('T2)]  .

 rl '_._['<_;_>['PosS.Place, 'ok.RPosType], '<_;_>['Ar1.Place, 'raw.Cointype]]
  => '_._['<_;_>['Pr.Place, 'raw.Cointype], '_._['<_;_>['PosI.Place, 'ok.RPosType], '<_;_>['PosS.Place, 'ok.RPosType]]] [label('D1)]  .

 rl '_._['<_;_>['PosS.Place, 'ok.RPosType], '<_;_>['Ar2.Place, 'forge.Cointype]]
  => '_._['<_;_>['Tb.Place, 'forge.Cointype], '_._['<_;_>['Ea.Place, 'Ear2.EmptyArmType], '<_;_>['PosS.Place, 'ok.RPosType]]]
[label('D2)]  .

rl '_._['<_;_>['Ar1.Place, 'raw.Cointype], '<_;_>['PosI.Place, 'ok.RPosType]]
          => '_._['<_;_>['Ar1.Place, 'raw.Cointype], '<_;_>['PosS.Place, 'ok.RPosType]] [label('TS1)]  .
rl '_._['<_;_>['Ar1.Place, 'forge.Cointype], '<_;_>['PosI.Place, 'ok.RPosType]]
          => '_._['<_;_>['Ar2.Place, 'forge.Cointype], '<_;_>['PosS.Place, 'ok.RPosType]] [label('TS2)]  .

rl '_._['<_;_>['Ea.Place, 'Ear1.EmptyArmType], '_._['<_;_>['Ea.Place, 'Ear2.EmptyArmType], '<_;_>['PosS.Place, 'ok.RPosType]]]
=> '<_;_>['PosI.Place, 'ok.RPosType] [label('TI)]  .

rl '<_;_>['Pr.Place, 'raw.Cointype] => '<_;_>['Pr.Place, 'forge.Cointype] [label('F)]  .
rl 'm:Marking => '_._['<_;_>['Ta.Place, 'raw.Cointype], 'm:Marking] [label('E)]  .

rl '<_;_>['Tb.Place, 'forge.Cointype] => 'mt.Marking [label('R)]  .
endm) .

endm
```

6. Conclusion

Dans ce chapitre, nous avons introduit le formalisme des ECATNets. Ces derniers disposent d'un côté graphique facilitant le développement d'un système au cours de sa spécification. En plus, les ECATNets sont intégrés dans la logique de réécriture, ce qui leur permet d'avoir une sémantique saine et complète. D'autre part cette version textuelle permet le développement des outils pour l'analyse des ECATNets. Nous avons présenté dans ce chapitre deux exemples des ECATNets ainsi que leurs implémentations dans Maude. Ces deux exemples seront utilisés ultérieurement dans l'explication des nos différents outils développés.

Chapitre 3 :
Programmes Ada Concurrents

1. Introduction

Ada [Iso95], [Iso01], [Eng97] est un langage de programmation, conçu par l'équipe dirigée par Jean Ichbiah, en réponse à un cahier de charges établi par le Département de la Défense américain (DoD). Son développement a commencé au début des années 1980 pour donner Ada 83. Il a été ensuite repris et amélioré au milieu des années 1990, pour donner Ada-95, le premier langage objet standardisé de manière internationale. Sous les auspices de l'organisation internationale de normalisation (ISO), le langage Ada a subi des révisions sous la forme d'un amendement au standard en 2005.

Ada est un langage modulaire, puissant (orienté objet, multitâche, normalisé). Le langage Ada est souvent utilisé dans la programmation des systèmes temps réel et embarqués nécessitant un haut niveau de fiabilité. A l'origine, Ada a été conçu avec trois soucis primordiaux : fiabilité de programme et maintenance, programmation comme une activité humaine et efficacité. Le langage offre l'appui pour la compilation séparée d'unités de programme dans une voie qui facilite le développement de programme et la maintenance et qui fournit le même degré de vérification entre des unités comme dans une unité. La conception aspire particulièrement à fournir des constructions du langage correspondant intuitivement aux espérances normales d'utilisateurs. Comme beaucoup d'autres activités humaines, le développement des programmes devient plus que jamais décentralisé et distribué. La capacité d'assembler un programme de composants de logiciel produits indépendamment continue à être une idée centrale dans la conception. Les concepts de packages, de types privés et d'unités génériques sont directement rapprochés de cette idée, qui a des ramifications dans beaucoup d'autres aspects du langage. Aucun langage ne peut éviter le problème d'efficacité. Les langages qui exigent trop en ce qui concerne l'efficacité finissent par compliquer les compilateurs, ou conduire à l'utilisation inefficace des unités de stockage ou du temps d'exécution. Ils imposent ces inefficacités à toutes les machines et à tous les programmes. Chaque construction du langage Ada a été examinée sous la lumière des techniques actuelles d'implémentation. N'importe quelle construction proposée dont la mise en oeuvre était peu claire ou cela a exigé des ressources de machine qu'excessives aient été rejetées.

Ada est souvent utilisé en introduction aux cours de programmation avancée. Actuellement, il est possible de trouver des compilateurs Ada de très bonne qualité pour toutes sortes de systèmes d'exploitation et d'architectures. Par exemple, le langage Ada est portable sur Win32, Linux et Unix.

Nous introduisons dans ce chapitre quelques concepts du langage Ada tout en se concentrant sur le concept de tâche. On définit les différentes constructions pour permettre une communication entre tâches ou le partage des données entre eux. Nous faisons appel à des exemples simples pour expliquer certains concepts du langage Ada.

Le reste de ce chapitre est organisé comme suit : dans la section 2, nous introduisons quelques concepts de base du langage Ada. Dans la section 3, nous présentons un exemple qui nous aide à expliquer ultérieurement le concept de tâche dans Ada. Les définitions de tâche et de type de tâches sont présentées dans la section 4. Les différents styles de communication entre tâches seront discutés dans la section 5. Dans la section 6, nous présentons le transfert des données pendant un rendez-vous. Nous expliquons le partage de données entre tâches dans la section 7. La section 8 conclut le chapitre.

2. Concepts de Base

Un programme Ada est composé d'une ou plusieurs unités de programmes. Les unités de programmes peuvent être des sous-programmes (définissant des algorithmes exécutables), des packages (définissant des collections d'entités), des unités de tâche (définissant des calculs concurrents), des unités protégées (définissant des opérations pour un partage coordonné entre tâches), ou des unités génériques (définissant des formes paramétrées des packages ou sous-programmes). Normalement, chaque unité de programme contient deux parties : une spécification contenant l'information qui peut être visible par d'autres unités, un corps contenant les détails d'implémentation, qui n'a pas besoin d'être visible par d'autres unités. La plupart des unités de programme peuvent être compilées séparément. Cette distinction entre la spécification et le corps, et la capacité de compiler les unités séparément, permet à un programme d'être conçu, écrit et testé comme un ensemble des composants larges indépendants d'un logiciel. Normalement, un programme Ada peut être mis dans une librairie pour une utilité générale. Le langage fournit des moyens par lequel des organisations individuelles peuvent construire leurs propres librairies. Toutes les librairies sont structurées dans une manière hiérarchique. Ceci permet une décomposition logique d'un sous-système à des composants individuels. Le texte d'une unité de programme compilée séparément doit nommer les unités de librairie dont il a besoin.

2.1. Unités de Programme

Un sous-programme est l'unité de base pour exprimer un algorithme. Le langage Ada supporte les deux types de sous-programmes : procédures et fonctions. Un package est l'unité de base pour définir une collection d'entités reliées logiquement. Par exemple, un package peut être utilisé pour définir un ensemble de déclarations de type et d'opérations associées. Des parties d'un package peuvent être cachées de l'utilisateur, de ce fait permettant l'accès seulement aux propriétés logiques exprimées par les spécifications de package. Des unités de sous-programme et de package peuvent être compilées séparément et arrangées dans les hiérarchies des unités de parent et d'enfant donnant un bon contrôle de visibilité des propriétés logiques et de leur exécution détaillée. Une unité de tâche est l'unité de base pour définir une tâche dont l'ordre des actions peut être exécuté en concurrence avec celles d'autres tâches. De Telles tâches peuvent être mises en application sur des multi computers, multiprocesseurs, ou avec l'exécution intercalée (interleaved) sur un processeur simple. Une unité de tâche peut définir soit une exécution simple d'une tâche ou un 'type de tâche' permettant la création de plusieurs tâches semblables. Une unité protégée est l'unité de base pour définir des opérations protégées pour l'usage coordonné des données partagées entre les tâches. L'exclusion mutuelle simple est fournie automatiquement, et des protocoles de partage plus raffinés peuvent être définis. Une opération protégée peut être un sous-programme ou une entrée. Une entrée protégée indique une expression booléenne (une barrière d'entrée) qui doit être vraie

avant que le corps de l'entrée soit exécuté. Une unité protégée peut définir un objet protégé simple ou un type protégé permettant la création de plusieurs objets semblables.

2.2. Déclarations et Instructions

Le corps d'une unité de programme contient généralement deux parties : une partie déclarative et une séquence d'instructions. La première partie définit les entités logiques au sein de l'unité du programme. La seconde partie définit l'exécution de l'unité du programme.

La partie déclarative associe des noms à des entités déclarées. Par exemple, un nom peut dénoter un type, une constante, une variable ou une exception. Cette partie introduit les noms et les paramètres des autres sous-programmes imbriqués, packages, unités de tâche, unités protégées et des unités génériques pour être utilisés dans l'unité de programme.

La séquence des instructions décrit une séquence des actions qui peuvent être exécutées. Les instructions sont exécutées en succession (à moins qu'un transfert de contrôle provoque l'exécution de continuer d'un autre endroit). Une instruction d'affection change la valeur d'une variable. Un appel d'une procédure invoque l'exécution d'une procédure après l'association des paramètres actuels fournis par l'appel aux paramètres formels correspondants.

Les instructions **case** et **if** permettent une sélection basée sur la valeur d'une expression ou celle d'une condition, d'une séquence d'instructions appropriée. L'instruction **loop** fournit le mécanisme itératif de base du langage. Une instruction **loop** spécifie qu'une séquence des instructions s'exécute d'une façon répétitive dirigée selon un arrangement itératif, ou jusqu'à une instruction **exit** est déclenchée. Une instruction **block** comprend une séquence des instructions précède par des entités locales utilisées par les instructions.

Certaines instructions sont associées à des exécutions concurrentes. Une instruction **delay** retarde l'exécution d'une tâche pour une durée indiquée ou jusqu'à un temps indiqué. L'instruction d'appel d'une entrée **entry call** est similaire à une instruction d'appel de procédure. Il demande une opération sur une tâche ou sur un objet protégé, bloquant l'appelant jusqu'à ce que l'opération puisse être exécutée. Une tâche appelée peut accepter un appel d'entrée en exécutant l'instruction correspondante **accept** qui spécifie les actions qui vont être exécutées. Ces actions font partie du rendez-vous avec la tâche appelante. Un appel d'entrée sur un objet protégé est traité quand la barrière d'entrée correspondante est évaluée vraie, sur quoi le corps de l'entrée est exécuté. L'instruction '**requeue**' permet la disposition d'un service comme un nombre d'activités liées avec le contrôle privilégié. Une forme de l'instruction **select** permet un sélectif attendant d'un ou de plusieurs rendez-vous possibles. D'autres formes de l'instruction **select** permettent des appels d'entrée conditionnels ou temporisés et le transfert asynchrone de contrôle en réponse à un certain événement de déclenchement.

L'exécution d'une unité de programme peut rencontrer des situations d'erreur dans lesquelles l'exécution de programme normale ne peut pas continuer. Par exemple, un calcul arithmétique peut excéder la valeur maximale permise pour un nombre, ou une tentative peut être faite pour avoir accès à un composant de tableau en utilisant

une valeur d'index incorrecte. Pour traiter de telles situations d'erreur, les instructions d'une unité de programme peuvent être textuellement suivies par les gestionnaires d'exception qui spécifient les actions à être prises quand la situation d'erreur surgit. Les exceptions peuvent être soulevées explicitement par une instruction **raise**.

2.3. Types de Données

Chaque objet dans le langage a un type, qui caractérise un ensemble de valeurs et un ensemble d'opérations applicables. Les classes principales des types sont des types élémentaires (incluant l'énumération, le type numérique et les types **access**) et des types composés (incluant les tableaux et les enregistrements). Un type d'énumération définit un ensemble ordonné de littéraux distinct d'énumération, comme une liste d'états ou un alphabet de caractères. Les types d'énumération **Boolean**, **Character**, et **Wide_Character** sont prédéfinis. Les types numériques fournissent le moyen d'exécuter exactement ou de rapprochement des calculs numériques. Les calculs exacts utilisent des types d'entier qui dénotent les ensembles d'entiers consécutifs. Des calculs approximatifs utilisent soit des types de point fixes, avec des limites absolues sur l'erreur, ou des types de virgule flottante, avec des limites relatives sur l'erreur. Les types numériques **Integer**, **Float**, et **Duration** sont prédéfinis.

Les types composés permettent des définitions des objets structurés avec les composants relatifs. Les types composés dans le langage Ada sont les tableaux et les enregistrements. Un tableau (**array**) est un objet avec des composants classés du même type. Un enregistrement (**record**) est un objet avec des composants de types probablement différents. Les tâches et les types protégés sont également des formes de types composés. Les types tableaux de **String** et **Wide_String** sont prédéfinis. Les types d'enregistrement, de tâche et les types protégés peuvent avoir des composants spéciaux appelés les discriminants qui paramétrisent le type. Des différentes structures **record** qui dépendent des valeurs des discriminants, peuvent être définies dans un type **record**. Les types **access** permettent la construction des structures de données liées. Une valeur d'un type **access** représente une référence à un objet déclaré comme **aliased** ou à un objet créé par l'évaluation d'un allocateur. Plusieurs variables d'un type **access** peuvent indiquer le même objet, et les composants d'un objet peuvent indiquer le même ou d'autres objets. Les deux éléments dans de telles structures de données liées et leur relation avec d'autres éléments peuvent être changés pendant l'exécution du programme. Les types **access** permettent également à des références aux sous-programmes d'être stockées, d'être passées comme paramètres, et finalement d'être déréférenciés en tant qu'une partie d'un appel indirect. Les types privés permettent des vues limitées d'un type. Un type privé peut être défini dans un package de sorte que seulement les propriétés logiquement nécessaires soient rendues visibles aux utilisateurs du type. Tous les détails structuraux qui sont extérieurement non pertinents sont alors seulement disponibles dans le package et toutes les unités enfant. De n'importe quel type, un nouveau type peut être défini par dérivation. Un type, ainsi que ses dérivés (les deux direct et indirect) forment une classe de dérivation. On peut définir des larges classes d'opérations qui acceptent comme paramètre un opérande de tout type dans une classe de dérivation. Pour les types **record** et privés, les dérivés peuvent être des extensions du type parent. Les types qui soutiennent ces possibilités orienté-objet des larges classes d'opérations et l'extension du type doivent être étiquetés, de sorte que le type spécifique d'un opérande dans une classe de dérivation puisse être identifié au temps d'exécution. Quand une opération d'un type étiqueté est appliquée à un opérande dont le type spécifique n'est pas connu jusqu'au temps d'exécution, l'expédition implicite est effectuée en se basant sur l'étiquette (**tag**) de l'opérande. Le concept d'un type est

encore raffiné par le concept d'un sous-type, par lequel un utilisateur puisse contraindre l'ensemble de valeurs permises d'un type. Des sous-types peuvent être employés pour définir des sous-ranges (**subranges**) des types scalaires, des tableaux avec un ensemble limité des valeurs d'index, des types **record** et des types privés avec des valeurs discriminantes particulières.

2.4. Objets Actifs

Les objets passifs agissent par l'intermédiaire des sous-programmes appelés (finalement) par le programme principal. Ils ne font rien de leur propre entente, seulement si on leurs demande de faire ainsi. Cependant, les objets actifs sont souvent bien utiles aussi. Ada nous permet de définir les tâches qui sont exécutées indépendamment. Le programme lui-même est exécuté par un environnement des tâches, et ceci est permis de créer d'autres tâches qui sont exécutées parallèlement avec d'autres tâches dans le programme. Notons que ce dispositif d'Ada dépend fortement du système d'exploitation ; un système d'exploitation comme le MS-DOS n'a aucune possibilité de traitement de sorte que les implémentations d'Ada pour des systèmes de MS-DOS puissent seulement fournir des possibilités faibles de traitement multitâche, le cas échéant. Heureusement la situation a changé, et les systèmes d'exploitation comme Unix, OS/2 et Windows/NT permettent les traitements multitâches.

3. Exemple : Spreadsheet Cells

L'exemple suivant nous aidera tout au long de ce chapitre à expliquer quelques concepts du langage Ada. Considérons un type de cellule de bilan (**spreadsheet cell**) qui se met à jour continuellement changeant constamment des prix obtenus en ligne à partir du marché boursier (**stock market**). La conception courante exigerait de la cellule (**cell**) d'obtenir les derniers prix toutes les fois que le bilan (**spreadsheet**) a été recalculé. Le programme principal pourrait recalculer le bilan à plusieurs reprises s'il était autrement libre, mais cela met la responsabilité du recalcul sur le programme plutôt que sur l'objet qui l'exige, dans ce cas, un type particulier de la cellule de bilan. Dans un bilan sans cellules de ce type, le recalcul continu est un gaspillage de ressources. Une cellule active de bilan résoudrait le problème en fonctionnant pendant que (parallèlement à) le reste du programme et en faisant une rêquete au bilan pour faire un recalcul toutes les fois que sa valeur a changé. De cette façon que le programme n'a besoin de rien savoir au sujet de la façon dont le bilan fonctionne ou quel type de cellules il pourrait contenir et n'aurait aucune responsabilité supplémentaire de contrôler le bilan. Les cellules actives comme celles-ci pourraient changer leurs valeurs en réponse aux conditions externes et puis appeler le bilan pour se recalculer. Le bilan lui-même pourrait être un objet actif que les attentes pour les demandes du recalcul, alors il se recalcule et s'affiche, parallèlement à toute autre chose qui se produit.

4. Types de Tâche

Comme les packages, les tâches dans Ada sont définies dans deux parties, une partie de spécification et une partie de corps. À la différence des packages, elles ne sont pas des unités de compilation ; elles ne peuvent pas être compilées indépendamment et ajoutées à la librairie. Au lieu de cela, elles doivent être déclarées à l'intérieur d'un package ou d'un sous-programme. Une fois déclarées dans un package, les spécifications entrent dans celles des packages et le corps entre dans celui du package. Une fois déclarées dans un sous-programme, les spécifications et le corps doivent être définis dans le sous-programme. Les spécifications de tâche définissent un type de tâche (**task type**). La forme la plus simple que ceci peut prendre ressemble à celle-ci :

```
task type Repeat;
```

Ceci définit un type appelé **Repeat**; nous pouvons alors déclarer autant de tâches de ce type que nous voulons :

```
A, B : Repeat;                    -- deux tâches
C : array (1..100) of Repeat;     -- un tableau de 100 tâches
```

Ces tâches commenceront à la fin de la section de déclaration où elles ont été créés, c.-à-d. juste avant la première instruction dans le bloc où elles sont déclarés :

```
declare
  A, B : Repeat;
  C : array (1..100) of Repeat;
begin             -- toutes les 102 tâches démarrent à ce point
  ...
end;              -- attente ici pour les 102 tâches pour terminer
```

Quand elles ont commencé, chacun veut exécuter une copie du corps de tâche parallèlement avec toute autre chose qui se produit. Les tâches sont locales au bloc, ainsi elles cessent d'exister à la fin du bloc. Quand la tâche exécutante le bloc atteint la fin du bloc, elle devra attendre les 102 tâches de finir avant qu'elle puisse procéder. La tâche exécutante le bloc est dite le maître des tâches créées dans lui, et les tâches dans le bloc sont dites dépendantes de leur tâche maître. Le corps de tâche contient une boucle. Voici un exemple simple :

```
task body Repeat is
begin
   for I in 1..5 loop
      Put_Line ("Hello!");   delay 2.0;
   end loop;
end Repeat;
```

Ceci affiche le message ‘Hello!’ cinq fois avant la terminaison. Après que le message soit affiché, la tâche va retarder pour deux secondes avant la continuation à l’aide de l’instruction **delay** (**delay 2.0;**). Ici 2.0 est une valeur du type standard de point fixe **Duration** qui indique la longueur du retard en secondes. Nous pouvons également retarder jusqu'à un moment particulier comme ceci :

```
delay until Ada.Calendar.Time_Of (Day => 25, Month=>12, Year => 1999);
```

Cette instruction causera la tâche qui l'exécute pour attendre jusqu'au Noël 1999. Le temps est spécifié comme valeur du type **Ada.Calendar.Time**. Les types de tâche sont des types limités, ainsi nous ne pouvons pas affecter des tâches à une autre tâche ou faire une comparaison entre elles. Ceci signifie également que n'importe quel type qui contient une tâche autant qu'un composant doit également être un type limité. Si nous voulons seulement une seule tâche d'un type particulier, nous pouvons déclarer les spécifications de tâche comme ceci :

```
task Repeat;
```

Le type de tâche est maintenant anonyme, et **Repeat** est le seul objet appartenant à ce type anonyme. En d'autres termes, c'est plus efficace d'écrire comme suit :

```
task type ???;    -- ??? est le 'nom' de type de tâche anonyme
Repeat : ???;     -- déclare un objet de ce type
```

Les tâches peuvent également avoir des discriminants, qui peuvent être un dispositif utile pour fournir des valeurs initiales :

```
task type Repeat (Count : Natural);
task body Repeat is
begin
   for I in 1..Count loop          -- le discriminant contrôle la longueur de la boucle
      Put_Line (Integer'Image(Count) & " Hello!");    delay 2.0;
   end loop;
end Repeat;
A : Repeat (Count => 10);         -- cette tâche affiche hello 10 fois
```

Notons que le discriminant est seulement décrit dans la spécification de la tâche et pas dans le corps, mais il peut encore être mentionné dans le corps. En outre, puisque les types de tâche sont des types limités, les discriminants d'accès (access discriminants) sont parfaitement acceptables.

5. Communication Entre Tâches

Normalement, il est nécessaire pour les tâches de communiquer entre eux; par exemple, une tâche dans **spreadsheet cell** a besoin d'être demandée pour connaître la valeur de **cell** de temps en temps, ou **spreadsheet** peut avoir besoin d'être demandée pour se recalculer. Dans de tels cas, la spécification de tâche a besoin d'être étendue pour mentionner les services qu'une tâche peut fournir. Dans ce qui suit, une spécification d'une tâche **spreadsheet** qui permet à d'autres tâches de demander d'elle son recalcul :

```
task type Spreadsheet_Task is
   entry Recalculate;
end Spreadsheet_Task;
Sheet : Spreadsheet_Task;        -- declare a Spreadsheet_Task
```

Ce type de tâche fournit une specification d'entrée (**entry**) qu'une autre tâche peut appeler comme une procédure. Par exemple, une tâche peut demander à **Sheet** de recalculer avec un appel d'entrée comme le suivant :

```
Sheet.Recalculate;
```

Le corps de tâche doit fournir une manière pour servir les appels à ses entrées. Ceci peut être fait en utilisant une instruction **accept** :

```
task body Spreadsheet_Task is
begin
   loop     accept Recalculate;  Do_Recalculation;     end loop;
```

```
end Spreadsheet_Task;
```

Quand le corps de tâche commence à s'exécuter, il attendra au niveau de l'instruction **accept** jusqu'à ce qu'un appel sur **Recalculate** soit fait. Il appellera alors une procédure **Do_Recalculation** pour exécuter le recalcul et pour aller autour de la boucle encore attendre le prochain appel de **Recalculate**. Si une autre tâche appelle **Recalculate** avant que la tâche **spreadsheet** revienne de nouveau à l'instruction **accept**, la tâche appelante est forcée d'attendre. Ainsi la tâche appelante et celle appelée attendront l'une l'autre jusqu'à ce qu'elles soient les deux prêtes. Ceci est réalisée quand l'appelante attend son appel d'entrée d'être accepté et l'appelante attend au niveau de l'instruction **accept**. Cette synchronisation des deux tâches est connue sous le nom de rendez-vous. Le corps de tâche décrit ci-dessus a un problème principal ; puisque c'est une boucle infinie il n'y a aucune manière de l'arrêter, ainsi la tâche maître ne pourra pas se terminer non plus ; il va attendre infiniment à la fin du bloc où la tâche **spreadsheet** a été déclaré. Une solution consiste à arrêter (**abort**) en utilisant l'instruction **abort**:

```
abort Sheet;
```

Ceci forcera la tâche et toutes les tâches dépendantes d'elle de se terminer. Cependant, la tâche **spreadsheet** pourrait être à mi-chemin d'un recalcul pendant ce temps là. Une meilleure manière serait d'ajouter une autre entrée pour permettre la tâche maître de lui demander de s'arrêter d'une façon ordonnée :

```
task type Spreadsheet_Task is
   entry Recalculate;
   entry Shutdown;
end Spreadsheet_Task;
```

Maintenant le corps de tâche doit pouvoir répondre aux appels à n'importe quelle de ses entrées. Ce n'est pas bien de les accepter l'un après l'autre dans une boucle puisque ceci forcera les entrées à être appelées alternativement. Une solution serait d'examiner si des appels sont en suspension avant de les accepter. Nous pouvons faire ceci en utilsant l'attribut de compte (**'Count attribute**) pour une entrée qui donne le nombre d'appels en suspension pour cette entrée :

```
task body Spreadsheet_Task is
begin
   loop
      if Recalculate'Count > 0 then
         accept Recalculate;   Do_Recalculation;
      elsif  Shutdown'Count > 0 then
         accept Shutdown;   exit;
      end if;
   end loop;
end Spreadsheet_Task;
```

Cependant, ce n'est pas particulièrement fiable. Car nous verrons plus tard, les tâches peuvent choisir d'abandonner des appels d'entrée s'elles ne sont pas répondues dans une certaine période de temps, et ceci

signifie que même si **Recalculate'Count** est différent de zéro, avant que nous exécutons l'instruction **accept** pour **Recalculate**, la tâche appelante pourrait avoir chronométré dehors et abandonné son appel, dans ce cas nous serons coincé à l'instruction **accept** jusqu'à ce qu'une autre tâche appele **Recalculate**. Et si cela ne se produit jamais, nous ne pourons jamais accepter un appel à **Shutdown**. La solution correcte à ceci est de mettre les appels à l'intérieur de l'instruction **select** :

```
task body Spreadsheet_Task is
begin
   loop
      select
         accept Recalculate;   Do_Recalculation;
      or
         accept Shutdown;   exit;
      end select;
   end loop;
end Spreadsheet_Task;
```

Cette instruction **select** contient deux alternatives **accept** qui doivent chacune être dirigées (headed) par une instruction **accept**. Elle attend jusqu'à ce qu'une des entrées décrites dans les instructions **accept** soit appelée, et elle éxécute alors l'alternative appropriée. Si les appels aux deux entrées sont déjà en suspension, un sera accepté d'une manière non-déterministe. L'instruction **select** s'achève après que l'alternative choisie a été exécutée; si **Recalculate** est appelé, la tâche revient au début de la boucle et attend un autre appel d'entrée, mais si **Shutdown** est appelé elle sort de la boucle et se termine. Il faut noter que la tâche ne répond pas aux appels à **Shutdown** si un **Recalculate** est en évolution; il répond seulement quand il attend un appel d'entrée au niveau de l'instruction **select**, qui s'avérera justement après la fin de l'appel de **Recalculate** et le retour au début de la boucle. Cette solution exige que la tâche maître appele **Shutdown** explicitement quand elle veut terminer la tâche. L'inconvénient de cette approche est qu'il est possible d'oublier d'appeler **Shutdown**. Une meilleure solution est d'ajouter une alternative de terminaison à l'instruction **select** :

```
task body Spreadsheet_Task is
begin
   loop
      select
         accept Recalculate;  Do_Recalculation;
      or
         accept Shutdown;   exit;
      or
         terminate;
      end select;
   end loop;
end Spreadsheet_Task;
```

L'alternative de terminaison doit être la dernière dans une instruction **select**, et elle ne peut contenir qu'une instruction **terminate** comme celle qui est montrée ci-dessus. Quand la tâche maître arrive à la fin du bloc où **spreadsheet** a été déclaré, la tâche **spreadsheet** se terminera à la prochaine fois quand l'instruction **select** est exécutée (ou immédiatement, si la tâche attend déjà dans l'instruction **select**). Ceci signifie que le maître ne doit rien faire pour provoquer la terminaison de la tâche, mais il peut encore appeler **Shutdown** s'il veut terminer la tâche avant d'atteindre la fin du bloc où elle a été déclarée. Notons qu'une fois qu'une tâche s'est terminée, nous serons récompensés par une exception de **Tasking_Error** si nous essayons d'appeler une de ses entrées. Les instructions **select** peuvent également être utilisées pour des appels d'entrée. Si une tâche appelée n'est pas prête pour attendre un rendez-vous, elle peut utiliser une instruction **select** avec une alternative **else** comme ceci :

```
select
   Sheet.Recalculate;
else
   Put_Line ("Sheet is busy -- giving up");
end select;
```

Dans ce cas, si **Sheet** ne peut pas accepter l'appel d'entrée pour **Recalculate** immédiatement, l'appel d'entrée sera abandonné et l'alternative **else** sera exécutée. Si la tâche appelée est disposée à attendre un temps limité, on peut utiliser une instruction **select** avec une alternative **delay** comme ceci :

```
select
   Sheet.Recalculate;
or
   delay 5.0;
   Put_Line ("Sheet has been busy for 5 seconds -- giving up");
end select;
```

Si l'appel d'entrée n'est pas accepté dans le temps indiqué au niveau de l'instruction **delay** (cinq secondes dans ce cas), il est abandonné et c'est l'alternative **delay** qui sera exécutée. Une instruction **delay until** peut toujours être utilisée au lieu d'une instruction **delay** :

```
select
   Sheet.Recalculate;
or
   delay until Christmas;   Put_Line ("Sheet has been busy for ages -- giving up");
end select;
```

Nous pouvons également fixer une limite supérieure sur le temps qu'elle prend pour traiter un appel d'entrée :

```
select
   delay 5.0;   Put_Line ("Sheet not recalculated yet -- recalculation abandoned");
then abort
   Sheet.Recalculate;
end select;
```

Ceci commence à évaluer les instructions entre **then abort** et **end select**, qui est dans ce cas un appel à **Sheet.Recalculate**. Si le délai indiqué par l'instruction **delay** (ou **delay until**) après **select** expire avant que ceci se termine, l'appel à **Sheet.Recalculate** est abondonné (comme par une instruction **abort**) et le message '**Sheet not recalculated yet -- recalculation abandoned**' sera affiché. L'utilisateur n'est pas limité à utiliser ceci en liaison avec le traitement multitâche; par exemple, nous pourions l'utiliser pour abondonner des calculs prolongés où le temps total d'exécution est important (ou où le calcul pourrait diverger pour donner des temps d'exécution potentiellement infinis) :

```
select
   delay 5.0;  Put_Line ("Horribly long calculation abandoned");
then abort
   Horribly_Long_And_Possibly_Divergent_Calculation;
end select;
```

Une instruction **select** à l'intérieur d'un corps de tâche, peut également faire une alternative **else** ou une ou plusieurs alternatives **delay** au lieu d'une alternative **terminate**. Nous devons également avoir au moins une alternative **accept**. Une alternative **else** est activée si aucune des instructions **accept** n'a d'un appel en attente d'entrée ; une alternative **delay** est activée si aucune des instructions **accept** n'accepte un appel d'entrée dans le temps indiqué dans l'instruction **delay**. Ces trois possibilités (**else**, **delay** et **terminate**) sont mutuellement exclusives. Nous ne pouvons pas avoir une alternative de **delay** en même temps qu'une alternative **terminate**, par exemple.

6. Transfert des Données Pendant un Rendez-vous

Il peut également être nécessaire de transférer des données entre tâches pendant un rendez-vous. Par exemple, un **spreadsheet cell** pourrait avoir besoin d'une entrée pour permettre à d'autres tâches d'avoir sa valeur. Pour permettre à celle-ci de se produire, les entrées de tâche peuvent également avoir des paramètres juste comme des procédures :

```
task type Counter_Task is
   entry Get (Value : out Integer);
end Counter_Task;
task body Counter_Task is
   V : Integer := 0;
begin
   loop select
         accept Get (Value : out Integer) do    Value := V;  V := V + 1;  end Get;
      or
         terminate;
      end select;   end loop;
end Counter_Task;
```

L'instruction **accept** dans le corps de cette tâche agit fondamentalement comme une procédure qui est appelée par l'appel de l'entrée. Il peut même contenir des instructions **return** juste comme une procédure. Quand l'appel d'entrée est accepté, les paramètres **in** sont transférés de l'appelant. Le corps de l'instruction **accept** est alors exécuté, et à la fin les paramètres **out** sont transférés de nouveau à l'appelant. Le rendez-vous est alors complété, et l'appelant est autorisé à continuer. Dans ce cas, la tâche produira des valeurs toujours croissantes de nombres entiers chaque fois que **Get** est appelée. Nous ne pourrions pas toujours être disposés à accepter un appel d'entrée. Considérons cette tâche qui contient une pile sur laquelle d'autres tâches peuvent empiler des données ou dépiler des items (des articles) :

```
task type Stack_Manager is
   entry Push (Item : in Integer);
   entry Pop (Item : out Integer);
end Stack_Manager;
task body Stack_Manager is
   package Int_Stacks is new JE.Stacks (Integer);
   Stack : Int_Stacks.Stack_Type;
begin
   loop
      select
         accept Push (Item : in Integer) do    Int_Stacks.Push (Stack,Item);    end Push;
      or
         accept Pop (Item : out Integer) do    Int_Stacks.Pop (Stack,Item);    end Pop;
      or
         terminate;
      end select;
   end loop;
end Stack_Manager;
```

Une exception sera submergée si une tentative d'appeler **Pop** est faite sur une pile vide. Notons que si une exception se produit pendant un rendez-vous, l'exception sera submergée dans la tâche appelante et celle appelée. Pour empêcher ça de se produire, nous pouvons ajouté une garde à l'instruction **accept** pour **Pop** comme ceci :

```
when not Int_Stacks.Empty (Stack) =>  accept Pop (Item : out Integer) do ...
```

Une instruction **guarded accept** peut seulement être activée quand la condition indiquée dans la garde est vraie. Ce qui signifie que **Pop** peut être appelé seulement quand la pile n'est pas vide. Chaque tâche qui appele **Pop** quand la pile est vide, sera forcée d'attendre jusqu'à ce que d'autres tâches appelent **Push**. Dès que **Push** est appelée, la pile ne sera plus vide de sorte que la prochaine fois que l'instruction **select** est exécutée, l'appel de **Pop** en attente est immédiatement accepté.

Les entrées gardées peuvent également être utiles pour abondonner des actions dans une construction **select ... then abort**. Cette dernière est gouvernée par une alternative **triggering** (la première instruction après **select**) qui doit être un appel d'entrée ou une instruction **delay**. Quand l'alternative **triggering** est activée (l'appel d'entrée est

accepté ou le délai est expiré) la partie abortable entre **then abort** et **end select** est abondonnée comme par une instruction **terminate** :

```
select
   User.Interrupt;
   Put_Line ("Horribly long calculation interrupted by user");
then abort
   Horribly_Long_And_Possibly_Divergent_Calculation;
end select;
```

Dans ce cas, si l'appel à **User.Interrupt** (l'entrée d'interruption de l'utilisateur de tâche) n'est jamais accepté, le calcul terriblement long sera arrêté. Si **User.Interrupt** a une garde, ceci signifie que, quand la condition de garde devient vraie, le calcul terriblement long entre **then abort** et **end select** sera arrêté.

7. Partage de Données Entre Tâches

En utilisant une tâche pour permettre multiple tâches d'accéder à une pile commune comme l'exemple décrit ci-dessus est une manière très raffinée et chère de partager des données. Il signifie qu'il doit y avoir une extra tâche pour gérer la pile en moitié avec les autres tâches qui veulent l'utiliser, et un rendez-vous est exigé pour accéder à la pile. Un rendez-vous est une opération relativement prolongée, ainsi il ajoute tout à fait de grands frais généraux à ce qui serait autrement un appel assez simple de procédure. On pourrait bien sûr déclarer la pile dans la même portée que les types de tâche qui doivent y avoir accès, mais c'est extrêmement risqué.

```
procedure Pop (Stack : in out Stack_Type;
   Item  : out Item_Type) is
begin
   Item := Stack.Body(Stack.Top);    -- 1
   Stack.Top := Stack.Top - 1;       -- 2
exception    when Constraint_Error =>   raise Stack_Underflow;
end Pop;
```

Cela montre comment **Pop** est implémenté en utilisant un tableau. Dans un environnement où seulement une tâche exécute ce code, c'est parfaitement sûr. Si plus qu'une tâche l'exécute simultanément, les deux tâches pourraient exécuter l'instruction 1 en même temps, alors toutes les deux peuvent avoir une copie du même item. Quand elles exécutent l'instruction 2, **Stack.Top** pourrait être décrementé deux fois ou les deux tâches pourraient retrouver la même valeur de **Stack.Top**, soustrairent 1 et stockent ensuite le résultat dans **Stack.Top**, alors que **Stack.Top** semblera avoir été decrementé seulement une fois. En d'autres termes, le résultat sera complètement imprévisible (unpredictable) puisqu'il dépend du rapport précis de synchronisation entre les deux tâches. L'imprévisibilité (Unpredictability) sur cette échelle est rarement une bonne propriété pour les systèmes informatiques. La morale de l'histoire est que les tâches ne devraient jamais accéder à des données externes ; elles devraient seulement accéder à leurs propres objets locaux.

Pour venir au bout de ce problème, Ada permet à des données d'être encapsulées dans un enregistrement protégé qui garantit que cette sorte de situation ne peut pas surgir. Un enregistrement protégé est un type de données passif plutôt qu'un type actif comme une tâche, ainsi les coûts d'un rendez-vous et l'ordonnacement d'une extra

tâche sont évités. Des enregistrements protégés sont divisés en une spécification et un corps, juste comme une tâche. Les spécifications contiennent une partie visible qui déclare un ensemble de fonctions, procédures et des entrées. Les tâches sont permises d'appeler ces entrées comme une partie privée qui contient des données à protéger. Voici un enregistrement protégé qui encapsule une pile de nombres entiers :

```
protected type Shared_Stack_Type is
   procedure Push (Item : in Integer);
   entry Pop (Item : out Integer);
   function Top return Integer;
   function Size return Natural;
   function Empty return Boolean;
private
   package Int_Stacks is new JE.Stacks (Integer);
   Stack : Int_Stacks.Stack_Type;
end Shared_Stack_Type;

Stack : Shared_Stack_Type;    -- declare an instance of Shared_Stack_Type
```

Comme pour les tâches, nous pouvons déclarer un seul enregistrement protégé d'un type anonyme en eliminant le mot **type** :

```
protected Shared_Stack_Type is ... ; -- same as: protected type ???;
                               --  Shared_Stack_Type : ???;
```

Le corps fournit les implémentations des fonctions, des procédures et des entrées déclarées dans les spécifications. La différence entre les trois types d'opération est qu'on permet seulement à des des fonctions de lire les valeurs des items d'informations privées (**private**); de tels items appraîssent à la fonction comme des constantes et la fonction ne peut pas les changer. Puisqu'il est sûr pour plusieurs tâches de lire les mêmes données en même temps, des multiples tâches sont autorisées à exécuter des fonctions dans un objet protégé en même temps. Des procédures et des entrées sont permises de changer les informations privées, ainsi une tâche ne peut appeler aucune des opérations protégées tandis que d'autres tâches exécutent un appel de procédure ou d'entrée. La différence entre une procédure et une entrée est que les entrées ont des gardes qui agissent comme les gardes des instructions **accept**; une entrée peut seulement être exécutée quand sa garde est vraie, et n'importe quelle tâche qui appelle une entrée dont la garde est fausse sera suspendue jusqu'à ce que la garde devienne vraie (à quel point l'appel d'entrée peut alors être exécuté). Dans le type protégé **Shared_Stack_Type**, il y a trois fonctions (**Top**, **Size** et **Empty**) qui n'affectent pas la pile privée qu'il contient. Les tâches pourront appeler ces fonctions aussi longtemps qu'aucun appel de procédure ou d'entrée n'est en évolution; s'il y a déjà un appel de procédure ou d'entrée en évolution, on ne permettra pas à la tâche appellante la fonction de procéder jusqu'à ce que l'appel en cours finisse l'exécution. Il y a une procédure (**Push**) ; chaque tâche appelante **Push** devrait attendre jusqu'à ce que tous les autres appels en cours finissent l'exécution. Il y a une entrée (**Pop**) ; n'importe quelle tâche appelante **Pop** devrait attendre, non seulement jusqu'à ce que tous les autres appels en cours

finissent l'exécution, mais également jusqu'à ce que la garde d'entrée soit vraie. Voici le corps protégé, la condition de garde pour l'entrée **Pop** est indiquée après la liste de paramètres entre **when** et **is** :

```
protected body Shared_Stack_Type is
   procedure Push (Item : in Integer) is    begin  Int_Stacks.Push (Stack,Item);   end Push;
   entry Pop (Item : out Integer)
      when not Int_Stacks.Empty (Stack) is    begin     Int_Stacks.Pop (Stack,Item);  end Pop;
   function Top return Integer is   begin   return Int_Stacks.Top (Stack);   end Top;
   function Size return Natural is   begin   return Int_Stacks.Size (Stack);  end Size;
   function Empty return Boolean is   begin  return Int_Stacks.Empty (Stack);  end Empty;
end Shared_Stack_Type;
```

Ainsi, autant de tâches que possible peuvent inspecter simultanément l'item en top de la pile, lisent la taille de la pile ou examinent si elle est vide aussi longtemps qu'auccune autre tâche est entrain d'empiler ou de dépiler un item. Le dépilement d'un item est seulement autorisé si la pile n'est pas vide ; si elle est vide la tâche appelante devra attendre jusqu'à ce que d'autres tâches appelent **Push**. Les appels de **Push** et de **Pop** avanceront seulement si l'enregistrement protégé n'est pas en cours d'utilisation par une autre tâche.

8. Conclusion

Dans ce chapitre, nous avons introduit le langage Ada. Nous avons mis l'accent en particulier sur la notion de 'multitâche' dans ce langage. Cette notion de multitâche a été travaillée soigneusement par l'ISO. Notons que beaucoup de concepts ainsi que leurs constructions syntaxiques ont été introduits pour assurer un degré élevée de concurrence. Ceci permet une utilisation efficace du langage Ada dans la programmation des systèmes distribués et décentralisés.

Partie 2 : Développement des Outils pour Les ECATNets à l'aide de Maude

La seconde partie concerne la présentation de certains algorithmes et outils que nous avons développés pour les ECATNets. Il s'agit de trois outils manquant aux ECATNets. Ces applications ne sont pas supportées par le système Maude. Nos outils couvrent : l'édition/simulation graphique, le graphe de couverture et les règles de réduction. Le premier chapitre est consacré au simulateur graphique développé aux ECATNets. Il s'agit d'un outil avec lequel l'utilisateur peut créer son ECATNet graphiquement. Notre outil transforme cette notation graphique en son équivalente description dans la logique de réécriture pour être exécutée sous le système Maude. Ensuite, l'outil convertit de nouveau le résultat obtenu dans Maude à une représentation graphique. Ce simulateur préserve l'avantage des ECATNets concernant leur aspect graphique et leur simple construction. A travers un exemple, nous montrons comment il est plus facile de développer un ECATNet à l'aide de notre simulateur qu'avec Maude directement. Puis, nous décrivons les différentes étapes de notre éditeur/simulateur.

Le second chapitre décrit notre analyseur dynamique que nous proposons pour les ECATNets. Cet analyseur a pour objectif de créer un graphe de couverture pour les ECATNets. A travers cet outil, nous voulons donner la possibilité de vérifier certaines propriétés statiques comme l'absence de deadlock dans les ECATNets à états infinis parce que les outils disponibles de l'analyse d'accessibilité et le Model Checking de Maude ne permettent pas l'analyse des systèmes à états infinis. Nous introduisons ce chapitre par une étude des places non-bornées. Dans cette section, nous présentons une série de propositions avec leurs preuves. Ces propositions détectent les cas des places non-bornées dans un ECATNet. Ensuite, nous proposons une couverture de l'infinité des termes algébriques dans une place en utilisant la représentation des ECATNets dans la logique de réécriture. En détectant les places non bornées et en proposant leur couverture, nous avons extrait un algorithme pour construire un graphe de couverture pour les ECATNets. Nous proposons une implémentation basé sur le paradigme fonctionnel de Maude de notre algorithme. Grâce à la réflectivité de la logique de réécriture, de telle implémentation était possible et n'était pas très compliquée. Nous présentons en détail notre outil en décrivant les types de données nécessaires et les fonctions principales. Enfin, l'application de notre l'outil sur un exemple à états infinis (exemple du robot) nous donne un graphe de couverture fini.

Le dernier chapitre dans cette partie explique notre démarche d'adaptation et d'implémentation des règles de réduction pour les ECATNets. Les règles de réduction présentent l'avantage de diminuer le problème de l'explosion combinatoire. Parmi les sept règles adaptées et implémentées pour les ECATNets. Dans le cadre de ce chapitre, nous présentons trois règles de réduction : 'les transitions sans places d'entrée', 'les transitions parallèles' et 'les places équivalentes'. Nous discutons leur adaptation aux ECATNets et leur implémentation dans Maude. A travers l'exemple du robot, nous montrons comment l'application de la règle de réduction 'les transitions sans places d'entrée' peut transformer un ECATNet non-borné à un autre borné, ce qui permet d'appliquer plusieurs méthodes d'analyse comme l'analyse d'accessibilité et le Model Checking.

Chapitre 1 :
Un outil Basé sur la Logique de Réécriture
Pour l'Edition et la Simulation des ECATNets

1. Introduction

La logique de réécriture offre aux ECATNets une version textuelle simple, intuitive et pratique pour les analyser sans perdre la sémantique formelle. Cependant, le système Maude offre un style textuel au utilisateur pour créer et traiter son ECATNet. On exécute un programme sous Maude en faisant appel au 'Command prompt'. Dans ce cas, nous perdons l'aspect graphique du formalisme ECATNet qui est important pour la clarté, la simplicité et la lisibilité dans le développement d'un système.

Dans ce chapitre, nous présentons un outil pour l'édition et la simulation des ECATNets. L'outil proposé est interactif pour bien exploiter les avantages des ECATNets comme l'aspect graphique, la simplicité et la lisibilité. Le principe de ce travail est comme suit : l'outil permet à l'utilisateur de faire une édition graphique d'un modèle ECATNet et de convertir cette représentation à sa description dans Maude. Après, l'outil appelle le système Maude pour l'exécution de l'ECATNet et reconvertit le marquage résultant décrit en Maude à une représentation graphique. Cette étape nous permet d'exploiter les avantages de Maude. A travers un exemple d'ECATNet, nous comparons les simulations de l'exemple sous le système Maude et notre outil.

Plusieurs outils d'édition, de simulation et d'analyse ont été proposés pour les réseaux de Petri standard et les réseaux de Petri de haut niveau (entre autres, les réseaux de Petri colorés (CPN) et les réseaux de Petri prédicats/transition). Le manque de tout outil pour une utilisation pratique des ECATNets en termes de description graphique, nous a motivé pour effectuer ce travail. La plupart des outils pour les réseaux de Petri sont implémentés en utilisant les langages impératifs comme Java ou C++. A notre connaissance, l'outil présenté dans le cadre de ce chapitre est le premier simulateur basé sur la logique de réécriture pour une catégorie des réseaux de Petri.

Le reste de ce chapitre est organisé comme suit : la section 2 souligne des travaux similaires. Dans la section 3, nous présentons l'exécution d'un exemple d'ECATNet sous le système Maude. Les fonctions principales de notre outil sont présentées dans la section 4. Nous montrons également, dans cette section comment nous exécutons l'exemple précédent en utilisant notre application. Les ressources logiciel et matériel utilisées dans le cadre de notre application sont mentionnées dans la section 5. Finalement, la section 6 conclut le chapitre.

2. Travaux Similaires

Il n'est pas possible que nous examinions ici la littérature étendue sur des outils de réseaux de Petri. Nous mentionnons juste quelques travaux pour mettre les choses dans leurs propres contextes. Dans le cadre de la simulation et de l'analyse des réseaux de Petri, plusieurs outils sont disponibles. La plupart des environnements sont matures et inclurent plusieurs outils d'analyse. De tells environnements concernent plusieurs types des réseaux de Petri. Nous mentionnons dans cette section quelques travaux. Un simulateur interactif des réseaux de Petri est proposé dans [Tou99]. PN-tools [Zbi95] est un ensemble d'outils graphiques pour les réseaux de Petri pour la construction, la modification et l'analyse des réseaux et les réseaux hiérarchiques. Une méthodologie des prototypes pour les systèmes concurrents larges modélisés par le biais des réseaux de Petri algébriques hiérarchiques est présentée dans [Hul96]. PNS [Bra93] est un simple système de simulation pour les réseaux de Petri place/transition étendu. Des réseaux de Petri peuvent être construits en utilisant un éditeur graphique et peuvent être exécutés dans un mode séquentiel ou parallèle. CPN/Tools [Bea01] est un 'redisign' de l'outil Design/CPN [Met96] pour l'édition, la simulation et l'analyse dynamique des réseaux de Petri colorés. Les outils CPN [Wel02] sont utilisés pour l'édition, la simulation et l'analyse des réseaux de Petri colorés. Le simulateur traite aussi bien les réseaux non-temporisés et temporisés. La fonctionnalité de l'ingénierie de simulation et les facilités de l'espace d'état sont semblables aux composants correspondants dans Design/CPN. Un simulateur des réseaux de Petri de haut niveau pour une machine parallèle est développé dans [Bul91]. Dans cet outil, une forme spéciale des réseaux Prédicat/Transition (FunPrE-net: réseaux predicate/event avec fonctions) est utilisé comme modèle d'une spécification. L'outil RTPNS (Real-Time Petri Net Simulator) [Cha90], fournit un outil de conception (design tool) avec une orientation graphique et une capacité de déboguage convenable pour les réseaux de Petri temps-réel. TimeNET [Kel95] est un logiciel package pour la modélisation et l'évaluation des performances avec les réseaux de Petri non-Markovian. In [Nic95], les auteurs décrit un outil de simulation pour les réseau de Petri fluides stochastique (FSPN : Fluid Stochastic Petri Net). La description d'entrée des ces derniers étend la syntaxe de l'outil SPNP (Stochastic Petri Nets Package) [Tri99]. Un simulateur pour les réseaux de Petri temporels (time Petri nets) SimPRES est proposé dans [Cor02]. Un simulateur de réseaux de Petri défini par la compagnie MathTools [Mat05] donne à l'utilisateur la capacité de construire et de simuler des larges types de réseaux de Petri. Les types des réseaux de Petri supportés inclurent : Discrete Petri Net, Autonomous Petri Net, Non-Autonomous T-Timed and P-Timed, Petri Net, Stochastic Petri Net, Continuous Petri Net, CSPN (Constant Speed Petri Net), VSPN (Variable Speed Petri Net) and Hybrid Petri-Net. Le simulateur Petri. Net est une application pour dessiner et simuler des RPs [Gor04]. Il a été conçu pour la modélisation, l'analyse et la simulation des systèmes de fabrication flexibles, mais il peut être utilisé pour les systèmes à événements discrets. Finalement, l'outil PNK (Petri Net Kernel) version 2.0 [Hau98] fournit une infrastructure pour construire des outils pour les RPs.

3. Exemple

Nous considérons l'exemple à propos des 'réseaux de communication'. Nous trouvons un exemple d'un état initial et le résultat de la réécriture illimitée de cet exemple sous le système Maude dans la figure 1. Pour montrer comment il est plus facile d'introduire un système ECATNet en utilsant notre outil comparant au système Maude, nous présentons ultérieurement comment créer et simuler ce même exemple avec notre application.

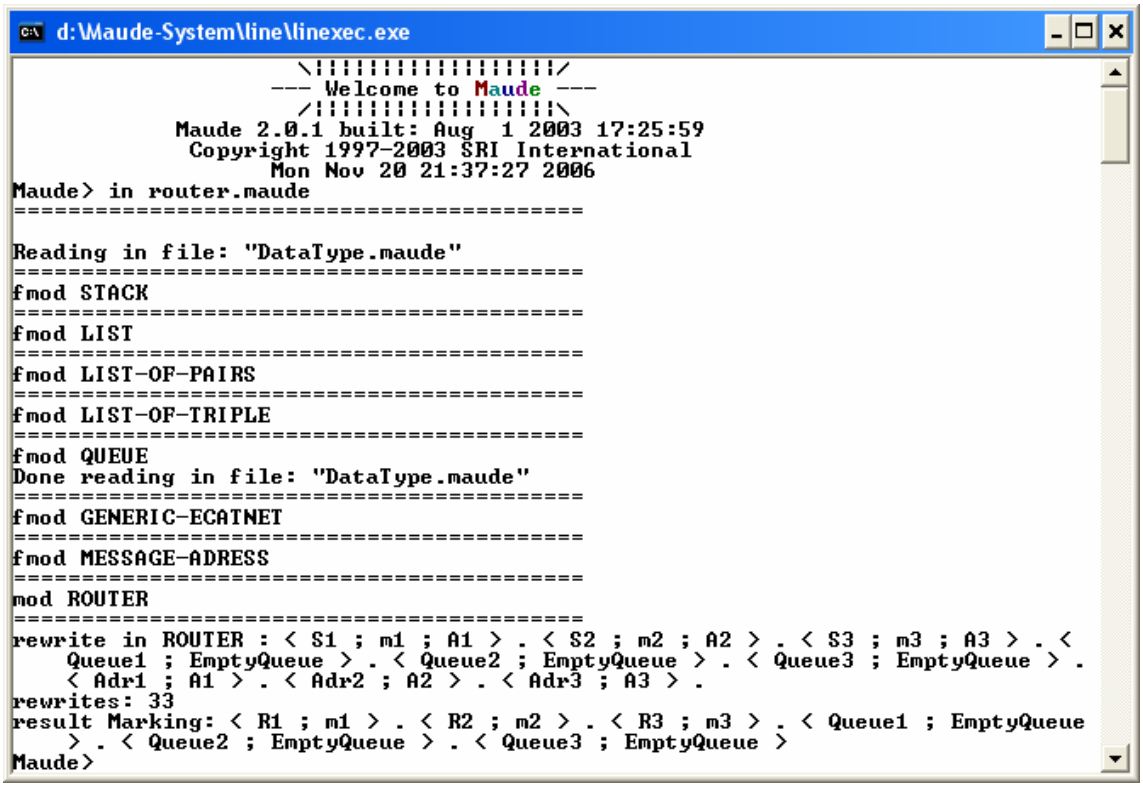

Figure 1. Exécution de l'exemple ECATNet sous le système Maude

4. Etapes du Simulateur des ECATNet

Comme il est décrit dans la figure 2, les principales étapes de cet éditeur/simulateur sont données. A travers l'exemple précédent, nous expliquons dans les sections suivantes, les options principales de cette application.

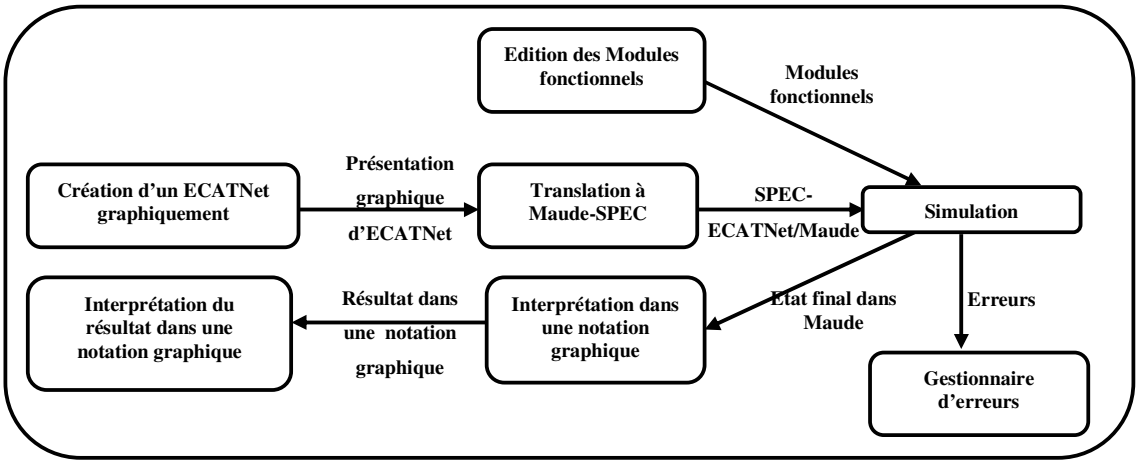

Figure 2. Vue méthodique du simulateur ECATNet

4.1. Interface Graphique

Dans cette étape, l'utilisateur peut créer son propre système ECATNet. Dans la figure 3, nous trouvons la fenêtre principale, sur laquelle l'exemple précédent est créé. Notons que, les marquages des places et les inscriptions des arcs n'apparaissent pas sur l'ECATNet. Pour cela, l'interface n'est pas très chargée.

Quand nous mettons le pointeur de la souris sur une place, un arc ou une transition, nous pouvons atteindre les informations à propos d'eux, incluant les multi-ensembles des termes algébriques appropriés. De telles informations sont affichées dans le coin le plus à droite de l'écran. Dans la figure 3, nous avons à droite de l'écran une boite indiquant le marquage initial de l'ECATNet exemple. Chaque ligne représente un marquage d'une place. Nous avons le nom de la place à gauche et son contenu (marquage) à droite. Les places comme Adr1 sont vides. Dans la figure 4, nous présentons une boite pour la manipulation des arcs. Cette dernière concerne la suppression de l'arc, le création et la destruction des inscriptions de l'arc.

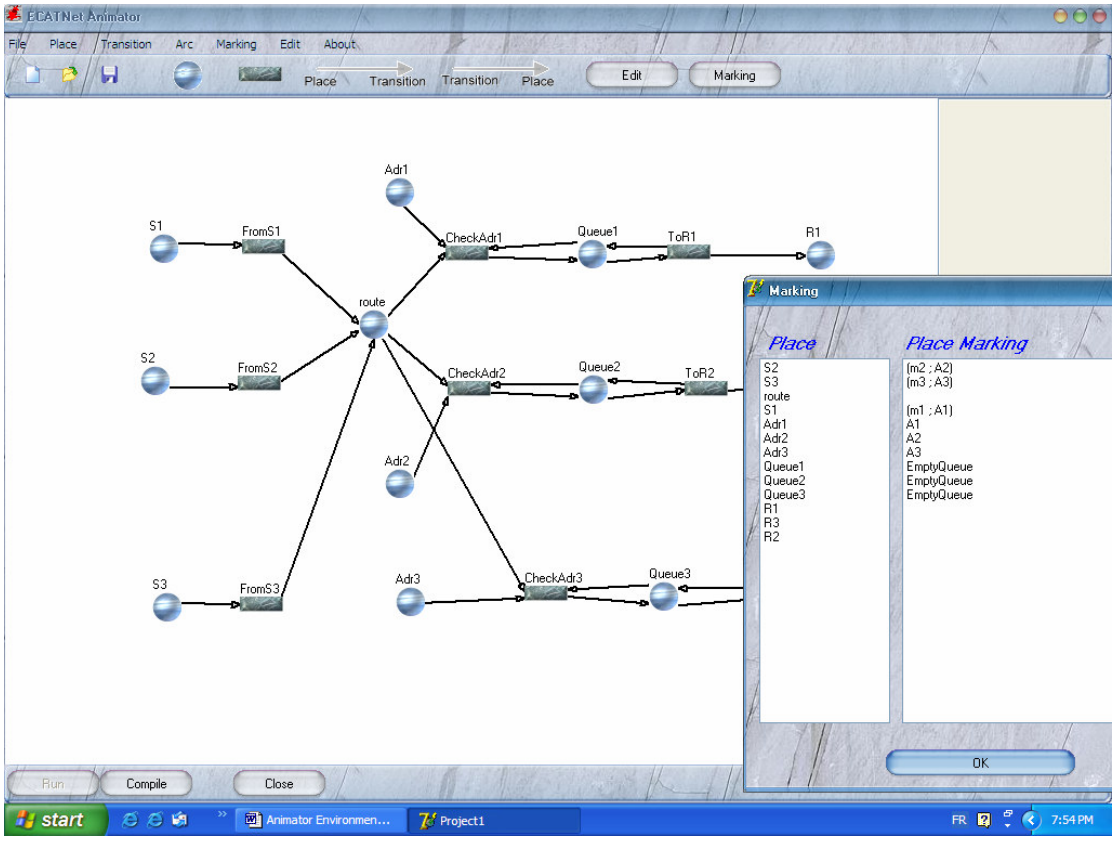

Figure 3. Menu principal de l'application avec un exemple d'ECATNets et un marquage initial

4.2. Translation de la Représentation Graphique des ECATNets vers une Description Maude

Cette étape doit avoir une représentation graphique d'un modèle ECATNet comme entrée. L'étape consiste à translater cette représentation graphique à une équivalente description dans Maude. En fait, cette représentation contient d'une part la structure de l'ECATNet et d'autre part, l'état initial de cet ECATNet. Le résultat de cette étape sont deux éléments : une équivalent code dans Maude pour la structure de l'ECATNet et l'état initial dans la syntaxe Maude.

4.3. Simulation

Le résultat de l'étape précédente est l'entrée de celle-ci. Pour effectuer la simulation, nous avons besoin de la part de l'utilisateur l'état initial de l'ECATNet. La simulation consiste à transformer ce marquage initial à un autre en faisant une ou plusieurs actions de réécriture. Par conséquent, en plus de l'état initial, l'utilisateur peut donner au simulateur le nombre des étapes de réécriture s'il veut contrôler les états intermédiaires. Si ce nombre

n'est pas donné, le simulateur continue le travail jusqu'à un état final. Dans la figure 3, nous demandons à l'application de faire la simulation sur l'exemple précédent du marquage initial sans indiquer le nombre des étapes de réécriture. L'état final (résultat) de la simulation est donné dans la même manière que l'état initial. Notons que le cas de l'infinité de calcul est possible.

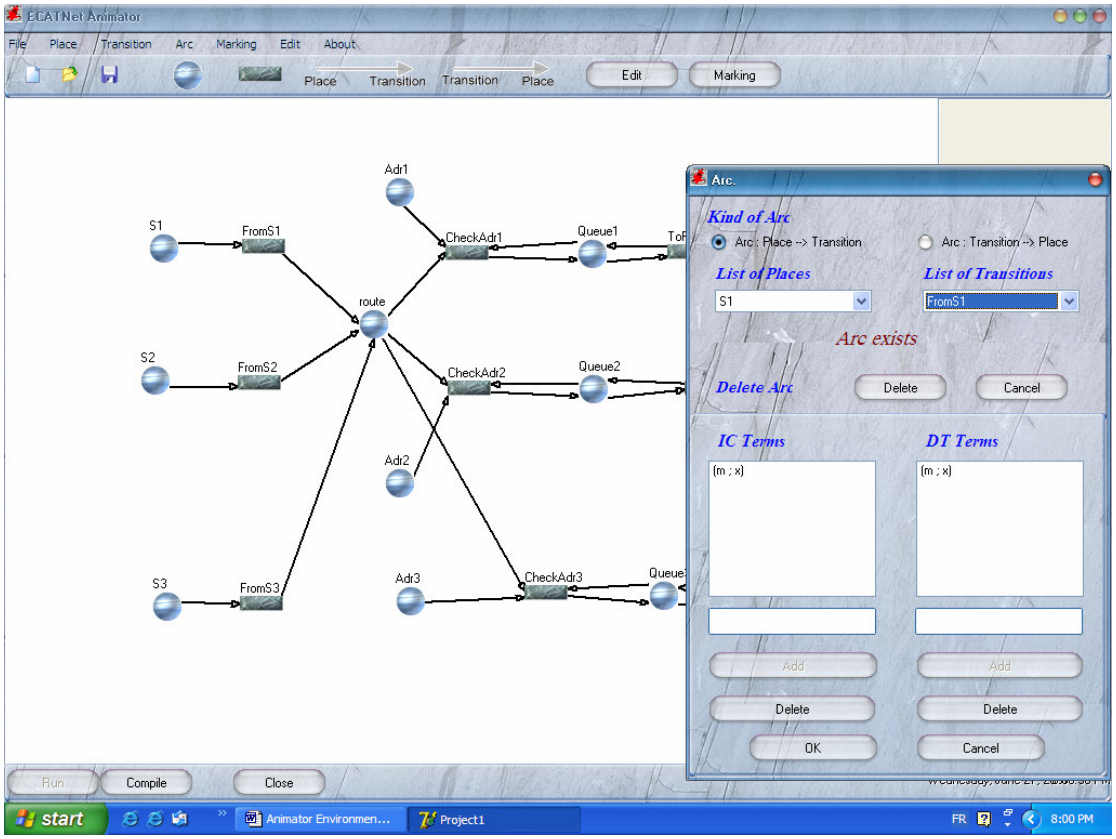

Figure. 4. Menu principal de l'application avec un exemple d'ECATNet et la boite de manipulation des arcs

4.4. Interprétation du Résultat dans une Notation Graphique

Dans la situation où il n'y a pas d'erreurs détectées, l'utilisateur obtient son résultat dans une boite comme celle du marquage initial. Ce résultat est retourné par Maude après l'avoir réinterpréter dans notre description.

4.5. Gestionnaire des Erreurs

Dans le cas de présence d'erreurs, nous fournissons à l'utilisateur les erreurs générées par Maude.

4.6. Edition des Modules Fonctionnels

Nous proposons un simple éditeur pour le développement des types de données. Nous utilisons la syntaxe de Maude des modules fonctionnels pour implémenter les types de données. Les modules fonctionnels et les modules systèmes générés (implémentant la structure de l'ECATNet) sont intégrés pour former un code exécutable dans Maude.

5. Aspect Technique du Simulateur des ECATNets

Ce simulateur est implémenté sous MS-Windows XP avec les outils suivants : le langage de programmation Delphi 7.0 est utilisé pour implémenter l'éditeur graphique et la translation de la représentation graphique d'un ECATNet vers une équivalente description dans Maude. La version 2.0.1 du système Maude est utilisée pour la simulation de la description générée de l'ECATNet.

6. Conclusion

Dans ce chapitre, nous décrivons notre outil d'édition et de simulation pour les ECATNets. L'éditeur permet à un utilisateur de dessiner un ECATNet graphiquement et de translater cette représentation graphique à une spécification dans Maude. Ce dernier est utilisé pour la simulation de la spécification. Notre objectif à travers ces outils c'est de faciliter à l'utilisateur de spécifier son système avec les ECATNets sans perdre l'avantage de l'aspect graphique. Nous avons montré à travers un exemple d'ECATNet, le gain obtenu en utilisant les outils proposés par rapport à une utilisation directe de la version textuelle des ECATNets dans Maude.

Chapitre 2 :
Analyse Dynamique des ECATNets

1. Introduction

Le développement des systèmes concurrents qu'on peut prouver qu'ils sont dépourvus d'erreurs reste un défi de l'ingénierie des systèmes. La modélisation et l'analyse des systèmes en termes de réseaux de Petri sont l'une des approches les plus connues parmi celles des méthodes formelles. Trois catégories des techniques d'analyse peuvent être distinctes dans le formalisme des réseaux de Petri : analyse structurel, analyse de réduction et analyse d'accessibilité. L'analyse structurelle et l'analyse de réduction sont dites les techniques d'analyse statique parce qu'elles sont basées sur la structure du modèle de réseau de Petri. Les techniques d'analyse d'accessibilité sont dites les techniques d'analyse dynamique parce qu'elles sont basées sur la construction du graphe d'accessibilité. Toutes les techniques d'analyse statique ont une caractéristique en commun, c'est qu'ils ne construisent pas le graphe d'accessibilité pour éviter le problème de l'explosion combinatoire. Les techniques d'analyse dynamique doivent être utilisés si les efforts d'analyse statique n'ont pas abouti à un succès, ou si les propriétés voulues ne peuvent pas être analysées statiquement (par exemple, réversibilité, 'live-lock freedom', etc…). A cause du problème très connu de l'explosion combinatoire, les méthodes paresseuses de construction sont élaborées. Ces méthodes considèrent seulement un espace d'états réduit. Par exemple, l'ensemble 'stubborn' réduisant le graphe d'accessibilité est d'habitude beaucoup plus petit que le graphe complet, mais il expose tous les états morts s'il y'en a. Pour les réseaux non bornés, les graphes de couverture doivent être utilisés. Quelques propriétés qualitatives comme l'absence de deadlock peuvent être vérifiées en utilisant la construction de graphe de couverture.

Les techniques automatisées comme le Model Checking [Bur92] sont efficaces pour spécifier et vérifier les modèles à états finis. Cependant, ils ne sont pas convenables pour la spécification des systèmes à état infinis. Maude implémente le Modèle Checking LTL. Ceci ne permet que la vérification des systèmes à états finis. Le fait que le formalisme des ECATNets est intégré dans Maude, alors il bénéficie de la possibilité de la vérification basée sur le Model Checking des systèmes décrits à l'aide des ECATNets. Cependant, il n'est pas possible de vérifier une propriété qualitative pour les systèmes à états infinis.

Dans l'objectif d'étendre ces possibilités de vérification pour le formalisme des ECATNets, nous décrivons dans ce chapitre, une analyse d'accessibilité pour les ECATNets. Cette analyse est basée sur la construction de graphe de couverture pour un modèle à états infinis ou finis. Pour cela, nous proposons une étude des places non bornées.

Le reste de ce chapitre est organisé comme suit : Dans la section 2, nous étudions quelques travaux similaires à notre travail. Une étude plus eu moins détaillée des places non bornées dans un ECATNet est présentée dans la

section 3. La section 4 est consacrée à la proposition de la couverture de l'augmentation infinie des nombres des éléments dans une place. La section 5 contient notre algorithme proposé pour une analyse dynamique des ECATNets. Dans la section 6, la construction d'un graphe de couverture pour un exemple est donnée. La section 7 conclut le chapitre.

2. Travaux Similaires

Plusieurs algorithmes ont été proposés pour construire le graphe de couverture pour les réseaux de Petri. L'outil d'analyse d'accessibilité PROD [Var95] pour les réseaux Pr/T implémente plusieurs méthodes pour une analyse d'accessibilité efficace. PAPETRI [Ber92] construit des graphes d'accessibilité et de couverture pour les réseaux P/T (Place/Transitions), RPCs et RPAs. CPN/AMI [Gir93], DESIGN/CPN [Pin93], et INA (Integrated Net Analyzer) [Roc99] calculent le graphe de couverture pour les réseaux P/T et RPCs avec temps et priorités en utilisant l'algorithme de KARP et MILLER. Dans le cas où le réseau est borné, le graphe de couverture correspond à l'habituel graphe d'accessibilité. Dans [Fin93], FINKEL considère les graphes d'accessibilité et de couverture comme des cas spéciaux d'une structure générale dite graphes -état. Parmi tous ces états des graphes, il existe une seule qui est minimale en ce qui concerne le nombre des nœuds. Pour les réseaux sans allocation de temps et priorités, et sous la règle de franchissement normal, INA offre la possibilité de calculer le graphe de couverture selon la méthode de FINKEL, et par conséquent réduit le nombre des nœuds et des arcs durant la construction. Pour les réseaux non bornés, le graphe résultat a peu de nœuds par rapport au graphe de couverture classique. Pour les réseaux non bornés, la dépense de temps complémentaire se trouve dans des limites acceptables, mais aucune réduction essentielle n'est faite. La connaissance du graphe -état minimal est suffisante pour décider des propriétés comme la bornétude du réseau, couverture des marquages ou l'existence des chemins infinis dans le graphe d'accessibilité.

3. Etude des Cas des Places non Bornées

Pour une analyse dynamique d'un ECATNet, la construction d'un graphe de couverture est faisable. Le développement d'un algorithme qui détecte et couvre tous les cas des places non bornées dans un ECATNet est un problème délicat. Dans notre sens, une place est non bornée si le nombre des termes algébriques augmente infiniment dans cette place.

L'étude de cas d'une place non bornée revient à l'étude de la propriété de monotonie. Dans un ECATNet, cette propriété dépend fortement des affectations des variables des termes algébriques apparaissant dans les inscriptions des arcs. Nous séparons trois cas des ECATNets : ECATNet simple (sans conditions de transitions avec places à capacité infinie), ECATNet avec conditions de transitions et avec places à capacité infinie, ECATNets sans conditions de transitions et avec places à capacité infinie, et enfin, le cas général.

Dans le cas des ECATNets avec places à capacité infinie, la propriété de monotonie est respectée. Dans le cas où les places à capacité finie, la monotonie peut ou ne pas être respectée.

Nous concentrons, dans notre étude, sur le cas quand IC(p, t) est de la forme $[m]_{\oplus}$ et nous excluons les deux autres formes (IC(p, t) est de la forme ˜$[m]_{\oplus}$, et IC(p, t) = empty).

3.1. ECATNet avec des Places à Capacité Infinie

Nous avons par la suite quelques propriétés et leurs preuves. Nous concentrons dans ces propositions sur des ECATNets du premier cas ($[IC(p, t)]_{\oplus} = [DT(p, t)]_{\oplus}$). Nous obtenons des résultats semblables pour les deux cas ($[IC(p, t)]_{\oplus} \cap [DT(p, t)]_{\oplus} = \phi_M$ et $[IC(p, t)] \cap [DT(p, t)] \neq \phi_M$).

3.1.1 Absence Conditions de Transitions

Proposition 1. Let M , M' deux marquages et S une séquence de transitions ; si $M \xrightarrow{S}$ et $M \subseteq M'$ **alors** $M' \xrightarrow{S}$

Preuve 1. Nous faisons appel à la preuve par récurrence. Pour $S = t$ (une transition), si t est franchissable dans M , alors nous avons :

$\forall p \in P \quad IC(p,t) \subseteq M(p)$ ou $DT(p,t) \subseteq M(p)$

$\forall p \in P$ si $M(p) \subseteq M'(p)$ alors $IC(p,t) \subseteq M'(p)$ ou $DT(p,t) \subseteq M'(p)$

Par conséquent, t est franchissable dans M'. Supposant que cette propriété est valide pour $S = t_1..t_k$ et nous prouvons que c'est le cas pour $S = t_1..t_k t_{k+1}$. Nous avons :

$M \xrightarrow{t_1..t_k} M_k \xrightarrow{t_{k+1}} M_{k+1}$

Par supposition, $M \subseteq M'$ alors $M' \xrightarrow{t_1..t_k} M'_k$

Maintenant, est ce que t_{k+1} est franchissable dans M'_k ?

Nous avons $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 ... \xrightarrow{t_k} M_k$

$\forall p \in P \qquad M_1(p) = (M(p) \setminus DT(p,t_1)) \otimes CT(p,t_1)$

Si $IC(p,t_1) \subseteq M(p)$ et $DT(p,t_1) \subseteq M(p)$

Tel que \ et $\otimes$ sont la soustraction et l'union des multi-ensembles. Tant que $DT(p,t_1) \subseteq M(p)$ alors, sans risques, nous pouvons écrire :

$M_1(p) = (M(p) \otimes CT(p,t_1)) \setminus DT(p,t_1)$

…

$M_k(p) = (M_{k-1}(p) \otimes CT(p,t_{k-1})) \setminus DT(p,t_{k-1})$

Alors $M_k(p) = ((M_{k-2}(p) \otimes CT(p,t_{k-2})) \setminus DT(p,t_{k-2})) \otimes CT(p,t_{k-1}) \setminus DT(p,t_{k-1})$

Nous pouvons écrire aussi :

$M_k(p) = ((M_{k-2}(p) \otimes CT(p,t_{k-2}) \otimes CT(p,t_{k-1})) \setminus (DT(p,t_{k-2}) \otimes DT(p,t_{k-1}))$

Par conséquent, nous écrivons :

$$M_k(p) = ((M(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l)) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l))$$

D'ailleurs et d'un autre côté, nous avons :

$$M'_k(p) = ((M'(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l))$$

Si $M_k(p) \subseteq M_k^{'}(p)$ alors $M(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l)) \subseteq M'(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))$

Et par conséquent :

$$(M(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l)) \subseteq (M^{'}(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l))$$

Parce que $\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l) \subseteq M(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))$ qui signifie que $M_k(p) \subseteq M_k^{'}(p)$

Si t_{k+1} est franchissable dans M_k alors elle est franchissable dans $M_k^{'}$.

Si $M \xrightarrow{t_1..t_{k+1}}$ et $M \subseteq M'$ alors $M^{'} \xrightarrow{t_1..t_{k+1}}$. Ceci signifie que la propriété de monotonie est respectée.

Proposition 2. If $M \xrightarrow{s} M_1$ et $M \subseteq M_1$ alors pour une place p, tel que $M(p) \subset M_1(p)$ est une places non bornée.

Preuve 2. Nous avons dans ce cas :

$M \xrightarrow{s} M_1 ... \xrightarrow{s} M_k$ tel que k tends vers l'infini avec $M \subseteq M_1$ $M_1 \subseteq M_2$ et $M_{k-1} \subseteq M_k$

Pour $p \in P$ si $M(p) \subset M_1(p)$ alors $\exists m(p) \neq \phi$ $M_1(p) = m(p) \otimes M(p)$

D'un autre côté, $M_2(p) = ((M_1(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l))$

Par conséquent, $M_2(p) = ((m(p) \otimes M(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l))$

Sans risque, nous pouvons écrire $M_2(p) = m(p) \otimes ((M(p) \otimes (\overset{K}{\underset{l=1}{\otimes}} CT(p,t_l))) \setminus (\overset{K}{\underset{l=1}{\otimes}} DT(p,t_l))$

Qui signifie que $M_2(p) = m(p) \otimes M_1(p)$ ou $M_2(p) = m(p) \otimes m(p) \otimes M(p)$

Nous avons $m(p) \subset m(p) \otimes m(p)$ alors $M(p) \subset M_1(p) \subset M_2(p)$

par récurrence, nous allons avoir $M(p) \subset M_1(p) \subset M_2(p) \subset ... \subset M_k(p)$ ou $M_k(p) = \underbrace{m(p) \otimes ... \otimes m(p)}_{k \ \ time} \otimes M(p)$ qui veut dire que si k tend vers l'infinie, alors le nombre des termes algébriques dans la place p tend aussi vers l'infini.

Interprétation 2. Pour une transition *t*, nous avons :

$\forall p \in {}^{\bullet}t \quad M^{'}(p) = M(p) \setminus DT(p,t)$ si $IC(p,t) \subseteq M(p)$

$\forall p \in t^{\bullet} \quad M^{'}(p) = M(p) \otimes CT(p,t)$

Pour $p \in t^{\bullet}$ $M(p) \subseteq M^{'}(p)$ alors $M(p) \subseteq M^{'}(p) \otimes CT(p,t)$

De telle situation est réalisée quelques soient les circonstances:

$p \in {}^{\bullet}t \ M(p) \subseteq M'(p)$ alors $M(p) \subseteq M^{'}(p) \setminus DT(p,t)$

Il est possible, seulement dans deux cas :

- $DT(p,t) = \phi$, nous sensibilisons sans retirer,
- Une place d'entrée est aussi une place de sortie $DT(p,t) \subseteq CT(p,t)$

Nous ajoutons des termes algébriques que nous venons de retirer.

3.1.2. Présence des Conditions de Transitions

La présence d'une condition d'une transition ne crée pas de problèmes en ce qui concernent la préservation de la monotonie. Une condition d'une transition est vraie si les valeurs la rendant vraie sont à l'intérieur des places d'entrée. Dans le cas d'une augmentation des multi-ensembles des termes algébriques dans ces places, ces valeurs sont toujours présentes et les conditions sont toujours vraies. Alors, si une transition est franchissable dans un marquage M , elle est toujours dans M' tel que $M \subseteq M'$.

3.1.3. ECATNet avec des Places à Capacité Finie

Nous distinguons deux cas, nous les discutons dans les propositions suivantes :

Proposition 3. Si $M_1 \xrightarrow{S} M_2$ et $M_1 \subseteq M_2$ et pour chaque place finie *p*, $M_1(p) = M_2(p)$, alors pour chaque place infinie *p'* tel que $M_1(p') \subset M_2(p')$ est une place non bornée.

Preuve 3. Pour simplicité, nous prenons uniquement en considération $S = t$ (une seule transition).

Considérons que $P = IP \cup FP$ (IP : Places à capacité infinie, FP : Places à capacité finie)

Si t est franchissable dans M_1 , alors nous avons :

$\forall p \in P \quad IC(p,t) \subseteq M_1(p)$ et $DT(p,t) \subseteq M_1(p)$

$\forall p \in P$ si $M_1(p) \subseteq M_2(p)$ alors $IC(p,t) \subseteq M_2(p)$ et $DT(p,t) \subseteq M_2(p)$

D'un autre côté, $\forall p \in FP : M_1(p) = M_2(p)$ si *t* ne change pas de marquages dans des places finies

Parce que si *t* retire des jetons d'une place finie, *t* met les mêmes jetons dans cette place ou *t* est indépendante de cette place. Alors, *t* est toujours franchissable dans M_2 .

$M_2(p) = (M_1(p) \setminus DT(p,t)) \otimes CT(p,t)$

Parce que $M_1(p) \subseteq (M_1(p) \setminus DT(p,t)) \otimes CT(p,t)$ alors $CT(p,t) \setminus DT(p,t)$ est un multi-ensemble positif. Dans ce cas, nous pouvons écrire : $M_2(p) = M_1(p) \otimes (CT(p,t) \setminus DT(p,t))$

Parce que $\forall p \in FP : M_1(p) = M_2(p)$ nous concluons que $\forall p \in FP : CT(p,t) \setminus DT(p,t) = \phi$

Nous pouvons continuer dans ce chemin et nous obtenons $M_3(p) = M_2(p) \otimes (CT(p,t) \setminus DT(p,t))$

$M_3(p) = (M_1(p) \otimes (CT(p,t) \setminus DT(p,t))) \otimes (CT(p,t) \setminus DT(p,t))$

$M_3(p) = M_1(p) \overset{2}{\underset{i=1}{\otimes}} (CT(p,t) \setminus DT(p,t))$, t est toujours franchissable dans M_3. Par récurrence, nous obtenons:

$M_k(p) = M_1(p) \overset{k}{\underset{i=1}{\otimes}} (CT(p,t) \setminus DT(p,t))$

Nous mettons $CT(p,t) \setminus DT(p,t) = m(p)$, $M_k(p) = M_1(p) \overset{k}{\underset{i=1}{\otimes}} (m(p))$, nous pouvons dire que $M_1 \xrightarrow{t^*}$

Pour une place $p \subset IP$ avec $M_1(p) \subset M_2(p)$, $m(p) \neq \phi$. Alors, si k tend vers l'infini, alors le nombre des termes algébriques dans la place p tend vers l'infini.

Proposition 4. if $M \xrightarrow{S} M'$ et $M \subseteq M'$ et la première transition dans S n'est pas franchissable dans M'. S'il existe *S'* tel que $M' \xrightarrow{S'} M''$ S' s'arrête quand S devient non franchissable. Si nous avons le cas suivant :

$M \xrightarrow{S} M' \xrightarrow{S'} M'' \xrightarrow{S} M'''$ et si $M' \subseteq M'''$ et $M''(p) \subseteq M(p)$ pour chaque place *p* à capacité infinie rendant S non franchissable dans M', alors chaque place infinie *p'* tel que $M'(p') \subset M'''(p')$ est une place non bornée.

Interprétation 4. La seule raison qui rend la première transition de *S* non tirable dans M', est le problème de dépassement de capacités de certaines places finies. *S'* diminue le nombre des termes algébriques dans certaines places finies. Pour cela, *S* devient de nouveau tirable, alors nous obtenons M'''. Nous pouvons avoir l'infinité dans certaines places à capacité infinie.

3.2. Couverture de l'Augmentation Infinie du Nombre des Eléments dans un Multi-ensemble

Pour chaque multi-ensemble dont les éléments sont de type T, nous créons ω_T qui couvre l'augmentation de nombre des éléments dans ce multi-ensemble. Soit m un multi-ensemble dont les éléments sont de type T, alors : $\forall m : T$, $m \otimes \omega_T = \omega_T$, $m \subseteq \omega_T$, $\omega_T \setminus m = \omega_T$, $\omega_T \subseteq \omega_T$, $\omega_T \otimes \omega_T = \omega_T$.

Tel que $\setminus$, $\subseteq$, $\otimes$ sont des extensions des opérations usuelles sur les multi-ensembles. Les nouvelles opérations s'appliquent sur l'union de tous les multi-ensembles de type T et le nouveau multi-ensemble ω_T. Nous interprétons les idées précédentes concernant la couverture de l'augmentation infinie du nombre des éléments dans un multi-ensemble en utilisant la logique de réécriture:

- $(p, \omega_T) \otimes (p, \omega_T) = (p, \omega_T)$
- $(p, e) \otimes (p, \omega_T) = (p, \omega_T)$ tel que e est un terme algébrique de type T. Par récurrence:
  $(p, e_1) \otimes \ldots \otimes (p, e_n) \otimes (p, \omega_T) = (p, \omega_T)$ tel que $e_1, \ldots, e_n$ sont de type T et $(n \geq 1)$
- Si nous avons une règle de réécriture de la forme : $(p, e_1) \otimes \ldots \otimes (p, e_n) \otimes m \rightarrow m'$ if C (m est indépendant de p), alors nous la changeons à la règle de réécriture : $(p, \omega_T) \otimes m \rightarrow (p, \omega_T) \otimes m'$ if C.

4. Un Algorithme pour Construire le Graphe de Couverture pour les ECATNets

D'abord, nous définissons quelques fonctions qui vont être utilisées dans le cadre de notre algorithme :

SubMarking(m, m') : retourne vrai si m est inclus dans m'.

StSubMarking(m, m') : retourne vari si m est sous-marquage de m' et $m \neq m'$.

ReachableMarking(m, l) : retourne le marquage résultant de franchissement d'une règle de réécriture dans m. Si ReachableMarking(m, l) = mt, alors la règle l n'est pas franchissable dans m.

CoveredPlaceInMarking(p, m) : pour une place p. Elle retourne un marquage après couvrir le sous-marquage concernant la place p. c'est-à-dire, si $m = (p, e_1) \otimes \ldots \otimes (p, e_n) \otimes m'$ (m' est indépendant de p et $n \geq 1$), alors CoveredPlaceInMarking(p, m) = $(p, \omega_T) \otimes m'$

CoveredSetofPlacesInMarking(P, m) : pour un ensemble de places P.

GetSubMarkingConcerningPlace(m, p) : donne un sous-marquage de m concernant la place p.

PlacesTobeCovered(m, m', IP) : retourne un ensemble de places non bornées E, un sous-ensemble de IP. Pour chaque p dans E, la fonction vérifie s'il y'a une réelle augmentation des jetons dans cette place en comparant entre m' et m. i.e, GetSubMarkingConcerningPlace(m', p) est strictement le sous-marquage de GetSubMarkingConcerningPlace(m, p). Un algorithme pour cette fonction est décrit comme :

function PlacesTobeCovered(m, m', IP) : ensemble de places;
var E, F : ensemble de places;
E := {}; F := IP;
for $p \in F$ **do**
if StSubMarking(GetSubMarkingConcerningPlace(m', p), GetSubMarkingConcerningPlace(m, p)) = true **then**
begin $E := E \cup \{p\}$; F:=F / {p}; **end;**
return(E);

Maintenant, nous donnons notre algorithme proposé pour la construction du graphe de couverture pour les ECATNets. Nous considérons que **(C1)** et **(C2)** sont les noms des conditions qui se trouvent après le *if*.

Algorithme Construction de Graphe de Couverture pour ECATNet

Input : ECATNet N sans arcs inhibiteurs, P: ensemble de places de N, $P = FP \cup IP$, FP : ensemble des places et IP : ensemble des places infinies, $FP \cap IP = \phi$. L : ensemble de transitions (règles de réécriture) de N, m_0 : marquage initial.

Output : Graphe de Couverture pour N

Méthode :

Begin var E : ensemble de places

1. La racine est étiquetée par le marquage initial m_0
2. Un marquage m n'a pas de successeur si et seulement si :
   - pour chaque règle de réécriture l, ReachableMarking(m, l) = ϕ
   - il existe sur le chemin de m_0 vers m, un autre marquage m' = m
3. **if** les deux conditions ne sont pas vérifiées, soit m'' le marquage tel que $m \xrightarrow{l} m''$ **then**

for (chaque règle l, tel que ReachableMarking(m, l) $\neq \phi$) **do**

if ($\exists$ m' : marquage sur le chemin de m_0 vers m **&** m' est sous-marquage de ReachableMarking(m, l)

& $\forall$ p $\in$ FP:

GetSubMarkingConcerningPlace(ReachableMarking(m, l), p)= GetSubMarkingConcerningPlace(m',p)) **(C1)**

or

($\exists$ m' : marquage sur le chemin de m_0 vers m **&** m' est strictement sous-marquage de ReachableMarking(m, l)

& ($\exists l_i$ (i =I, 2)$\in$ L **&** $l_1 \neq l_2$ sur le chemin de m' jusqu'à m tel que

& $\exists$m1, m2 deux marquages sur le chemin de m' jusqu'à m

& $m' \xrightarrow{l_1 S l} m_1$ **and** $m_1 \xrightarrow{l_2 S'} m_2$ **and** $m_2 \xrightarrow{l_1 S l}$ ReachableMarking(m, l)

& SubMarking(m', m_1) = true

& ReachableMarking(m_1, l_1) = ϕ

& $\forall$ p $\in$ FP: GetSubMarkingConcerningPlace(m_2, p)

is sub-marking of GetSubMarkingConcerningPlace(m', p)) **(C2)**

then E:= PlacesTobeCovered(ReachableMarking(m, l), m', IP, l);

m'' := CoveredSetofPlacesInMarking(E, ReachableMarking(m, l));

else m'' = ReachableMarking(m, l);

5. Implémentation de l'Algorithme dans Maude

Dans le cadre de ce travail, nous avons utilisé comme plate-forme, Maude dans sa version 2.0.1 sous Windows-XP. Le développement de cette application est possible grâce au concept de méta-calcul dans Maude. Plusieurs détails sont ajustés grâce à l'intégration des ECATNets dans Maude. Par exemple, une transition peut avoir une condition qui peut être intégrée dans la règle. L'appel de metaApply nous permet d'évaluer la condition et de nous libérer de traiter ce détail.

5.1. Représentation du Graphe de Couverture

Un triple < T ; L ; T' > signifie que la réécriture de T (T pour Terme) donne T' en utilisant la règle L. Ceci est équivalent de dire le franchissement de la transition représentée par la règle L dans le marquage représenté par le terme T, donne comme résultat le marquage représenté par le terme T'. Le graphe de couverture sera représenté par une liste des triples de cette forme.

5.2. Fonctions de l'Application

Plusieurs fonctions sont développées dans le paradigme de programmation fonctionnel pour implémenter l'algorithme décrit au-dessus. Dans cette section, nous décrivons les fonctions de base permettant le calcul des marquages accessibles pour expliquer la démarche de notre implémentation. La fonction ReachableMarking(M, T, L) retourne un successeur de tout le terme T quand on applique L, sinon, la fonction retourne le marquage vide mt. M est le module représentant le système ECATNet :

```
op ReachableMarking : Module Term Qid -> Term .
eq ReachableMarking(M, T, L) =
if metaApply(M, T, L, none, 0) =/= failure
 then if getTerm(metaApply(M, T, L, none, 0)) == 'mt.Marking
```

```
        then 'mt.Marking   else  getTerm(metaApply(M, T, L, none, 0)))    fi
 else mt   fi .
```

La fonction AccessibleMark(M, T, T', L, IP), avec IP l'ensemble des places à capacité infinie, T un terme à calculer son successeur en franchissant la règle L, T' est le terme qui se trouve sur le chemin entre T0 jusqu'à T. T' vérifie la condition **(C1)**. Cette fonction retourne, en cas de succès, un triple de la forme < T ; L ; RT > tel que RT est le marquage successeur de T en franchissant L:

```
op AccessibleMark : Module Term Term Qid List -> Triple .
eq AccessibleMark(M, T, T', L, IP) = MediumAccessibleMark(M, T, T', GetAllSubTerms(T), L, IP) .
```

Cette fonction appelle une autre MediumAccessibleMark(M, T, T', GetAllSubTerms(T), L, IP). La fonction GetAllSubTerms(T) retourne une pile de tous les sous-termes de T. Ceci est nécessaire parce que metaApply(M, T, L, none, 0) donne un terme résultant Ssi le terme T en entier correspond exactement à la partie gauche de la règle L. Pour un super-terme de T et qui est différent de lui, cette fonction ne retourne pas de terme résultant. Dans ce cas, nous procédons à la décomposition du terme pour extraire tous ses termes composants. Puis, nous appliquons à chaque sous-terme top(S), la fonction ReachableMarking(M, top(S), L) pour la règle L. Le sous-terme composant qui est une partie gauche de cette règle est soustrait de son super-terme. Le résultat RT de la soustraction est ajouté au terme résultat de ReachableMarking(M, top(S), L). La fonction MakingRefinement garantie l'application des propriétés de la couverture de l'augmentation de nombre des éléments dans certaines places:

```
op MediumAccessibleMark : Module Term Term Stack Qid List -> Triple .
ceq MediumAccessibleMark(M, T, T', S, L, IP) =
if S == emptystack
then errorltt
else if ReachableMarking(M, top(S), L) == mt
      then  MediumAccessibleMark(M, T, T', pop(S), L, IP)
      else  if T == top(S))
            then   < T ; L ; MarkingReffinement(CoveredSetofPlacesInMarking(PlacesToBeCovered(T',
                              ReachableMarking(M, T, L), IP),  ReachableMarking(M, T, L))) >
             else    < T ; L ; MarkingReffinement(CoveredSetofPlacesInMarking(PlacesToBeCovered(T', RT, IP),
                              RT)) >
fi   fi   fi   if RT := TermAddition(TermSubstraction(T, top(S)), ReachableMarking(M, top(S), L)) .
```

La fonction principale de notre application est CoverabilityGraph(M, T0). Cette fonction est une interface avec l'utilisateur :

```
op CoverabilityGraph : Module Term -> ListOfTriple .
eq CoverabilityGraph(M, T0) =   AllAccessibleMark(M, T0, T0, emptystack, emptystack, GetRulesName(M),
T0, emptyt, IPs, FPs, GetRulesName(M)) .
```

CoverabilityGraph(M, T0) appelle et initialise les paramètres de la fonction AllAccessibleMark(M, T0, S, S1, S', LS, APath, LT, IP, FP, LS1). Notons que FP est l'ensemble des places à capacité finie et S est une pile contenant les marquages à traiter. Quand nous obtenons les marquages successeurs de top(S), nous le mettons d'abord dans S1. Si S devient vide, nous passons au traitement des marquages qui se trouvent dans S1 et nous vidons le contenu de S1 dans S. S' est la pile contenant les marquages qu'on a déjà traité. Ceci est important pour éviter à chercher les marquages successeurs pour ceux qui sont déjà traités. LS est une liste contenant initialement toutes les étiquettes des règles du module et LS1 est une liste contenant constamment toutes les règles du module (GetRulesName(M)). APath est une pile des listes de termes, tel que chaque liste des termes est un chemin. Le premier terme de cette liste (chemin) est le marquage initial et le dernier est un marquage attendant le calcul de ses marquages successeurs. LT contient le graphe de couverture crée jusqu'à ce moment. Pour une raison de simplicité, nous nous limitons à présenter une partie de code de la fonction AllAccessibleMark décrivant seulement **(C1)** et nous excluons de notre explication la partie du code relative à **(C2)** :

```
op AllAccessibleMark : Module Term Stack Stack Stack List Stack ListOfTriple List List List -> ListOfTriple .
ceq AllAccessibleMark(M, T0, S, S1, S', LS, APath, LT, IP, FP, LS1) =
if S == emptystack
 then if S1 == emptystack
       then emptyt      else AllAccessibleMark(M, T0, S1, emptystack, S', LS1, APath, LT, IP, FP, LS1)      fi
 else     if LS == emptyList
          then AllAccessibleMark(M, T0, pop(S), S1, push(top(S), S'), LS1, APath, LT, IP, FP, LS1)
          else if findTermInStack(top(S), S') == true
              then AllAccessibleMark(M, T0, pop(S), S1, S', LS1, APath, LT, IP, FP, LS1)
               else   if   DirectAccessibleMark(M, top(S), head(LS)) == errorltt
                    then AllAccessibleMark(M, T0, S, S1, S', tail(LS), APath, LT, IP, FP, LS1)
                    else    if (T' =/= nil  and   ConstantMarksInFPlaces(M,  T',  RT, head(LS), FP) == true)
                            then
                   ConcatListOfTriple(AccessibleMark(M, top(S), T',  head(LS), IP),
                   AllAccessibleMark(MExt,  T0, S,
                    DeleteDupInStackExt(TrListToStack(AccessibleMark(M, top(S), T', head(LS), IP), S1)),
                    S', tail(LS),
                    PathSCreation(top(S),   GetRTerms(AccessibleMark(M, top(S), T', head(LS), IP)), APath),
                    ConcatListOfTriple(AccessibleMark(M, top(S), T', head(LS), IP), LT), IP, FP, LS1))

 else ConcatListOfTriple(DirectAccessibleMark(M, top(S), head(LS)),
                   AllAccessibleMark(M, T0, S,
                   DeleteDupInStackExt(TrListToStack(DirectAccessibleMark(M, top(S), head(LS)), S1)),
                   S', tail(LS),
                   PathSCreation(top(S), GetRTerms(DirectAccessibleMark(M, top(S), head(LS))), APath),
                   ConcatListOfTriple(DirectAccessibleMark(M, top(S), head(LS)), LT),
                   IP, FP, LS1))
```

fi fi fi fi fi

if T' := SubMarkingFoundOnPath(M, T0, top(S), RT, APath, head(LS))

/\ RT := 3rd(DirectAccessibleMark(M, top(S), head(LS)))

/\ MExt := ModuleExt(M, IP, 3rd(AccessibleMark(M, top(S), T', head(LS), IP)), RT) .

Nous calculons un marquage accessible de top(S) avec chaque règle (head(LS)) dans LS. A chaque fois que LS devient vide, nous réinitialisons son contenu avec les éléments de LS1 pour continuer le calcul des marquages accessibles d'un autre marquage. Si S et S1 sont vides, alors il n' y a plus un marquage à calculer ses successeurs. Ça marque la fin du calcul. Si S est vide et S1 n'est pas vide, nous vidons le contenu de S1 dans S. Ceci permet le calcul des marquages accessibles de ceux dans S1. Il est nécessaire de réinitialiser LS par LS1.

Quand S n'est pas vide (i.e, nous avons des marquages à calculer leurs successeurs), nous avons besoin de vérifier quelques conditions avant de calculer les successeurs de top(S). D'abord, nous contrôlons si LS n'est pas vide et top(S) n'existe pas dans S'. Si LS est vide, alors ça signifie que nous avons déjà calculé tous les marquages accessibles de top(S), nous l'écartons et nous le mettons dans S'. Si top(S) existe dans S', alors top(S) est déjà traité.

Si de telles conditions sont vérifiées, nous pouvons procéder au calcul des marquages accessibles de top(S) avec la règle head(LS). Pour cela, nous appelons DirectAccessibleMark(M, top(S), head(LS)). Notons que DirectAccessibleMark(M, top(S), head(LS)) ressemble à AccessibleMark, à l'exception que dans DirectAccessibleMark, nous calculons le marquage accessible de top(S) avec head(LS), sans couvrir aucune place. Cette fonction assure que la règle head(LS) est franchissable ou non dans la marquage top(S). Le terme errorltt indique que head(LS) n'est pas franchissable dans top(S). Dans ce cas, nous écartons cette règle et nous continuons de voir s'il existe d'autres règles (tail(LS)) franchissables dans top(S). Si head(LS) est franchissable, nous vérifions que les conditions des places non bornées sont valides ou non. Ceci est exprimé par le biais de la condition (T' =/= nil and ConstantMarksInFPlaces(M, T', RT, head(LS), FP) == true) qui exprime la condition **(C1)** de l'algorithme. Si la condition est vraie, nous appelons la fonction AccessibleMark pour calculer le marquage accessible avec la couverture nécessaire des places dans IP. Le triple résultant de cette fonction sera concaténé avec le résultat retourné par un nouvel appel de la fonction AllAccessibleMark. Cette fois, nous appelons cette fonction avec des nouvelles valeurs de ses paramètres. La couverture de certaines places non bornées permet l'introduction de nouvelles sortes et opérations et la modification de certaines règles. La fonction ModuleExt permet la modification du module en incluant ces nouveaux éléments. Nous apportons plus de clarification à propos de cette fonction à travers l'exemple présenté au niveau de ce chapitre. 3rd(DirectAccessibleMark(M, top(S), head(LS))) retourne le troisième élément dans le triple (le terme résultant de franchissement de head(LS) dans top(S)). TrListToStack(AccessibleMark(M, top(S), T', head(LS), IP), S1) met dans S1 le troisième élément dans le triple AccessibleMark(M, top(S), T', head(LS), IP), tel que DeleteDupInStackExt élimine n'importe quelle duplication dans la pile résultant. PathSCreation(top(S), GetRTerms(AccessibleMark(M, top(S), T', head(LS), IP)), APath) met le nouveau marquage accessible crée dans sa place appropriée dans APath. Finalement, ConcatListOfTriple(AccessibleMark(M, top(S), T', head(LS), IP), LT) permet l'ajout du nouveau triple crée AccessibleMark(M, top(S), T', head(LS), IP) au graphe de

couverture existant (LT). Nous n'entrons pas dans les détails de la description de cette information parce que nous n'avons pas besoin d'elle pour implémenter **(C1)**, mais plutôt **(C2)** comme il est décrit au-dessus.

6. Application de l'Outil sur un Exemple

L'objectif de cette section est l'application de l'exemple précédent sur un simple cas industriel. Il s'agit de l'exemple du Robot qui décrit un système réel à états infinis. Pour avoir le graphe de couverture de l'exemple, nous appelons la fonction CoverabilityGraph(META-ROBOT, '_._['<_;_>['PosI.Place, 'ok.RPosType], '<_;_>['Ea.Place, 'Ear2.EmptyArmType]]). Une partie de ce graphe est donnée dans la figure 1.

Figure 1. Partie de graphe de couverture du système Robot sous Maude

Comme nous le constatons, la méta-représentation est difficile à lire. Pour la lisibilité, nous expliquons manuellement et graphiquement le résultat dans la figure 2.

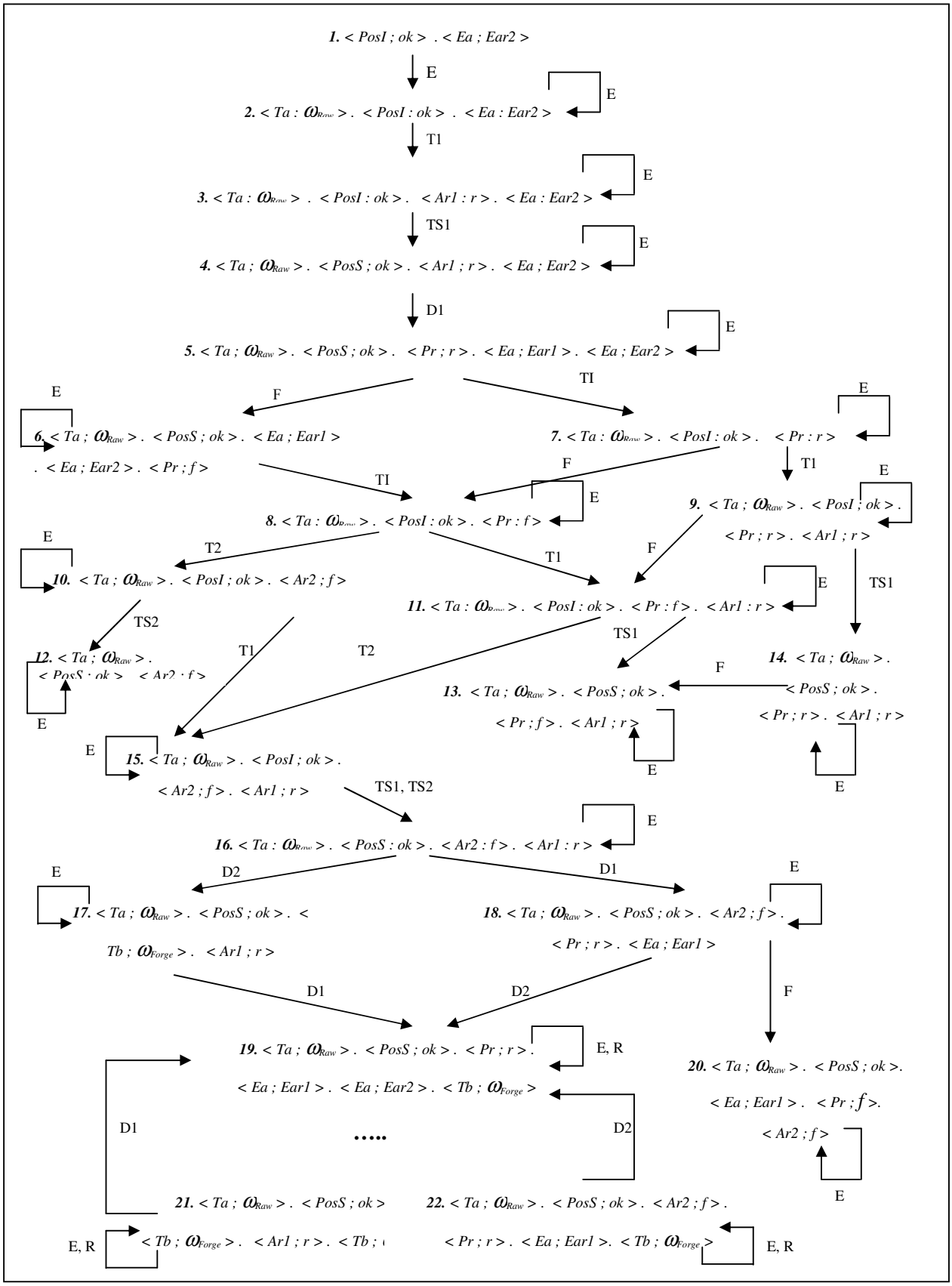

Figure 2. Graphe de couverture du système Robot

Le marquage initial est numéroté 1. La seule transition franchissable au niveau de marquage est E. Le franchissement de cette dernière dans la marquage initial donne comme résultat le marquage numéro 2. Notons que le marquage initial est inclus dans ce marquage et les places à capacité finie ont un contenu constant tout au long de ce chemin. La place Ta est non bornée, alors nous couvrons l'augmentation infinie de marquage dans cette place en utilisant le symbole ω_{Raw}, tel que Raw est le type énuméré contenant la valeur raw. A chaque marquage accessible, la transition E est toujours tirable. Le tir de E dans tel marquage donne le marquage lui-même. Le marquage numéro 8 est inclut dans le marquage numéro 11. Cependant, la place Pr est une place finie. La transition D2 est franchissable dans le marquage numéro 18. Le franchissement de cette transition donne le marquage : m'_{19} = < Ta ; ω_{Raw} > . < PosS ; ok > . < Pr ; r > . < Ea ; Ear1 > . < Ea ; Ear2 > . < Tb ; forge >. Notons que le marquage numéro 5 est inclut dans ce marquage. En plus, pour chaque place finie p, GetSubMarkingConcerningPlace(ReachableMarking(m'_{19}, l), p)) est égal à GetSubMarkingConcerningPlace(m_5, p). En appliquant l'algorithme précédent, la place Tb est non bornée et nous remplaçons ce marquage par le numéro 19. Si nous continuons la recherche de tous les marquages accessibles du numéro 16, nous trouvons un chemin similaire à celui commençant du marquage numéro 5. La différence est que les marquages accessibles dans le marquage numéro 18, contient le sous-marquage < Tb ; ω_{Forge} > dans chaque marquage et les transitions E et R sont toujours franchissables. Le tir de la transition D1 (resp. D2) dans le marquage numéro 21 (resp. 22), nous permet de retourner le marquage numéro 19. Le graphe de couverture obtenu est fini.

Après le franchissement de 'E dans le marquage initial et la création du marquage accessible approprié (numéro 2). Nous étions obligé de couvrir la place non bornée 'Ta. Place. Pour continuer le calcul du reste du graphe du couverture, nous devons modifier le module en ajoutant de nouveaux éléments qui ont servi à couvrir 'Ta. Place. Dans ce cas, la fonction ModuleExt nous permet d'ajouter (sort 'rawSort .), (subsort 'rawSort < 'GenericTerm .) et (op 'Omegaraw : nil -> 'rawSort [ctor] .). Les règles de réécriture 'E et 'T1 changent selon ce qui est décrit dans la section 3.3. Par exemple 'E devient :rl '<_;_>['Ta.Place,'Omegaraw.rawSort] => '<_;_>['Ta.Place,'Omegaraw.rawSort] [label('E)] .

7. Conclusion

Dans ce chapitre, nous avons proposé un outil basé sur la logique de réécriture pour une analyse dynamique des ECATNets. Cet outil nous permet d'énumérer l'espace d'états (construction de graphe de couverture) et de vérifier quelques propriétés de base. Le développement de cet outil n'était pas très compliqué grâce à la réflectivité du langage Maude. De tel algorithme est motivé par le fait que l'analyse de l'accessibilité ainsi que le Model Checking ne peuvent pas vérifier les modèles à états infinis. En plus, les ECATNets ne disposent d'aucun algorithme d'analyse dynamique par rapport à d'autres types de réseaux de Petri ordinaires ou de haut niveau.

Chapitre 3 :
Application des Règles de Réduction Aux ECATNets

1. Introduction

Les techniques d'analyse supportées par les réseaux de Petri en général et les ECATNets en particulier sont habituellement très coûteuses, et toute règle de réduction proposée pour diminuer une telle explosion combinatoire est intéressante. Plusieurs règles de réduction sont définies pour certaines catégories des réseaux de Petri, entre autres, ordinaires, colorées et algébriques. Le manque de toute tentative concernant l'application des règles de réduction pour les ECATNets justifie ce travail. Nous avons voulu enrichir les applications des ECATNets par un outil implémentant pour eux des règles de réduction. Dans [Sch97], Schmidt a présenté quelques règles de réduction pour les RPAs. L'application de certaines règles, par exemple, peut diminuer les étapes de la création du graphe d'accessibilité ou remplacer un réseau non-borné par un autre borné. Chaque règle de réduction préserve un ensemble des propriétés. Ce travail fait partie de notre contribution d'adapter les règles de réduction des RPAs proposées dans [Sch97] aux ECATNets. Dans son article, l'auteur a présenté neuf règles de réduction. Quelques unes ayant des besoins d'implémentation différentes par rapport à d'autres. Dans notre travail, sept règles de réduction sont adaptées et implémentées pour les ECATNets : 'les transitions sans places d'entrée', 'les places parallèles', 'les places latérales de condition', 'les transitions boucles', 'les places sans transitions d'entrée', 'les transitions parallèles' et 'les places équivalentes'.

Dans le cadre de ce chapitre, nous présentons trois règles de réduction : 'les transitions sans places d'entrée', 'les transitions parallèles' et 'les places équivalentes'. Pour plus de détails sur les autres règles, voir [Sch97]. Nous montrons après comment nous avons adapté les conditions d'application de ces règles aux ECATNets. Ensuite, un outil basé sur la logique de réécriture mettant en application ces règles de réduction sera présenté. Nous montrons aussi à travers un exemple comment l'application de l'une des règles de réduction peut transformer un ECATNet non-borné à un autre borné. Ceci permet d'appliquer certaines méthodes d'analyse tel que le Model Checking de Maude.

Le reste de ce chapitre est organisé comme suit : Dans la section 2, nous donnons une brève définition des RPAs et les règles de réduction : 'les transitions sans places d'entrée', 'les transitions parallèles' et 'les places équivalentes' telle qu'elles sont présentées dans [Sch97]. Notre adaptation de ces règles de réduction pour les ECATNets en utilisant leurs descriptions dans la logique de réécriture est présentée dans la section 3. Dans la section 4, nous décrivons l'implémentation de la première règle de réduction dans Maude. Dans la section 5, nous montrons comment nous appliquons l'outil développé sur un exemple. Finalement, la section 6 conclut le chapitre.

2. Réseaux de Petri Algébriques

2.1. Définition (Réseaux de Petri Algébrique)

Un tuple $AN = [D; P, T, F; \psi, \xi, \lambda; m_0]$ est un RPA Ssi :

- $[\Sigma, E]$ est une spécification avec $\Sigma = [S, \Omega]$;

- [P, T, F] est un réseau;
- $\psi : P \rightarrow S$;
- ξ affecte un ensemble des Σ-variables $\xi(t)$ à chaque transition $t \in T$;
- λ est l'inscription de l'arc tel que pour f = [p, t] ou f = [t, p] dans F, $\lambda(f)$ est le multi-terme sur $T_{\Omega,\psi(p)}(\xi(t))$;
- m_0 est le marquage initial, il affecte un multi-terme fini sur $T_{\Omega,\psi(p)}$ pour chaque $p \in P$.

Note. Pour une place p, l'ensemble de ses transitions de sortie est pF et l'ensemble de ses transitions d'entrée est Fp. N est le nom de réseau avant l'application de la règle de réduction et N' est le nom de réseau après l'application de la règle de réduction.

2.2. Règles de Réduction pour les RPAs

Dans cette section, nous présentons trois règles de réduction comme elles sont présentées dans [Sch97] : 'les transitions sans places d'entrée', 'les transitions parallèles' et 'les places équivalentes'. Les RPAs sont polymorphes, ils peuvent être interprétés selon différents modèles de spécification. Schmidt décrit les conditions d'application de chaque règle pour différents modèles de spécification. Dans notre travail, nous tenons compte seulement de l'algèbre standard (algèbre initiale).

2.2.1. Règle 1 : Transitions sans Places d'Entrée

Les transitions sans places d'entrée sont toujours franchissables. Leurs places de sortie peuvent être arbitrairement marquées. Par conséquent, ces places ne sont pas un obstacle réel pour rendre leurs post-transitions franchissables. Les transitions sans places d'entrée peuvent être enlevées ainsi que leurs places de sortie. Une légère généralisation de cette règle nous permet son application si pour chaque place de sortie d'une transition sans places d'entrée et pour chaque couleur de cette place il y a une transition sans places d'entrée qui produit les jetons de cette couleur dans cette place.

Condition d'Application. Soit T_0 non vide, l'ensemble de toutes les transitions sans places d'entrée. Soit $P_0 = T_0F$, l'ensemble des places de sortie de T_0. Pour chaque $p \in P_0$ et chaque couleur c de p, il existe une transition $t \in T_0$ et un mode de franchissement β de t tel que le franchissement de $[t, \beta]$ produit des jetons de couleur c dans p.

Application. Supprimer T_0, P_0 et tous les arcs connectés à eux.

Propriétés.
- Chaque transition supprimée est vivante dans N ;
- Chaque place supprimée est non-bornée dans N ;
- Chaque formule concernant uniquement les places dans N' est valide dans N' Ssi elle est valide dans N ;
- N est vivant Ssi N' est vivant, si N' est vide alors N est vivant et non-borné.

Intérêt. L'application de cette règle peut remplacer un réseau non-borné par un autre borné. Dans ce cas, plusieurs méthodes d'analyse peuvent être appliquées.

2.2.2. Règle 2: Transitions Parallèles

Informellement, deux transitions sont dites parallèles si elles sont connectées de la même manière à toutes les places du réseau. Dans ce cas, nous retirons une transition parce que le franchissement d'une remplace l'autre.

Propriétés.

- Pour chaque $m \in R_N(m_0)$, $R_N(m) = R_{N'}(m)$;
- Chaque place dans N' est bornée dans N, Ssi elle est bornée dans N';
- t_1 est vivante dans N Ssi t_2 est vivante dans N';
- Chaque transition dans N' est vivante dans N Ssi elle est vivante dans N';
- Chaque formule est valide dans N' Ssi elle est valide dans N.

Intérêt. Le processus de réduction diminue le nombre des étapes dans la création du graphe d'accessibilité. Quand nous générons ce dernier, nous pouvons vérifier parmi plusieurs transitions parallèles, une seule transition si elle est franchissable ou non. En plus, cette réduction permet la diminution du nombre des arcs du graphe.

Condition d'Application. Il existe deux transitions (t_1 et t_2) et une substitution σ tel que $\forall p \in P$:
$\lambda([p, t_1]) \equiv_I \lambda([p, t_2])\sigma$ et $\lambda([t_1, p]) \equiv_I \lambda([t_2, p])\sigma$.

Application. Supprimer t_1 et tous les arcs connectés à elle.

2.2.3. Règle 3 : Places Equivalentes

Deux places sont équivalentes si les transitions dépendantes de ces places ont le même effet sur le reste du réseau. Par conséquent, nous pouvons fusionner ces deux places. Les jetons produits dans une place ou dans l'autre des deux places équivalentes, sont regroupés dans la place résultante de la fusion.

Condition d'Application. Il existe deux places p_1 et p_2 et un nombre naturel strictement positif k :

1. $p_1F \cap p_2F = \phi$
2. Pour chaque $t \in p_1F$ il existe $t' \in p_2F$, une substitution σ sur $\xi(t')$ et un terme T tel que :

a) Pour chaque $p \in P$, $\lambda([t, p]) \equiv_I \lambda([t', p])\sigma$;

b) Pour chaque $p \in P\backslash\{p_1, p_2\}$ $\lambda([p, t]) \equiv_I \lambda([p, t'])\sigma$;

c) $\lambda([p_1, t]) \equiv_I \lambda([p_2, t'])\ \sigma$;

d) $\lambda([p_1, t]) = k.T$;

3. Pour chaque $t \in p_2F$ il existe $t' \in p_1F$, une substitution σ sur $\xi(t')$ et un terme T tel que :

a) Pour chaque $p \in P$, $\lambda([t, p]) \equiv_I \lambda([t', p])\sigma$;

b) Pour chaque $p \in P\backslash\{p_1, p_2\}$ $\lambda([p, t]) \equiv_I \lambda([p, t'])\sigma$;

c) $\lambda([p_2, t]) \equiv_I \lambda([p_1, t'])\sigma$;

d) $\lambda([p_2, t]) = k.T$;

4. Pour tout $t \in Fp_1 \cup Fp_2$ et pour tous les termes T, $k \mid \lambda([t, p_1])(T)$, $k \mid \lambda([t, p_2])(T)$, $k \mid m_0(p_1)(T)$ et $k \mid m_0(p_2)(T)$.

Application. Supprimer p_1, p_1F et tous les arcs commençant de p_1. Remplacer tous les arcs vers p_1 par les arcs vers p_2, inscrits de la même manière. Si tous les arcs existent déjà, ajouter l'inscription de cet arc vers p_1 par celui vers p_2. Ajouter $m_0(p_1)$ à $m_0(p_2)$.

3. Discussion et Adaptation des Règles de Réduction aux ECATNets

On considère p une place d'entrée (resp. de sortie) pour r si p figure dans la partie gauche (resp. droite) de la règle (transition) r. Soit M le nom du module décrivant le système ECATNet à traiter. Considérons que toutes les places de M sont non-bornées. La fonction MoreGeneralTerm(M, T1, T2, CD) utilisée par la suite dans la condition d'application, décide que le terme algébrique T2 n'est que l'une des valeurs possibles pour le terme algébrique T1 si une certaine condition CD et valide. Le terme nil indique qu'il n'y a pas de condition. L'implémentation de cette fonction dans la section 6 apporte une explication détaillée de la nature de cette fonction.

3.1. Règle 1 : Transitions sans Places d'Entrée

La forme d'une transition sans place d'entrée dans Maude est : rl [r] : m => m . Rg(r), tel que m est une variable de sorte Marking. Rg(r) est la partie droite de la règle r. C.-à-d., la règle r est franchissable dans n'importe quel marquage (représenté par la variable m) et son franchissement ajoute des jetons dans les places figurant dans (Rg(r)).

Condition d'Application. Soit R_0 qui n'est pas vide l'ensemble de toutes les règles sans places d'entrée dans M. Soit P_0 l'ensemble de toutes les places de sortie pour les règles dans R_0. Soit T1 (resp. T2) : le terme résultant de l'addition multi-ensembliste de tous les termes $< p ; T >$ ($p \in P_0$, T : terme) qui figurent dans la partie droite (resp. gauche) des règles de réécriture du module M. Nous avons MoreGeneralTerm(M, T1, T2, nil) == true.

Application. Supprimer toutes les règles de R_0, toutes les places $p \in P_0$ et tous les termes $< p ; T >$ pour $p \in P_0$ qui figurent dans les règles restantes de M après la suppression de R_0.

3.2. Règle 2 : Transitions Parallèles

Nous décrivons informellement, comment la condition d'application de la règle 2 peut être exprimée dans la logique de réécriture pour les ECATNets. La formule $\lambda([p, r_1]) \equiv_I \lambda([p, r_2])\sigma$ peut être implémentée dans la logique de réécriture en faisant une correspondance (matching) entre le côté gauche de r_1 et celui de r_2. Les transitions dans les ECATNets (et donc les règles dans Maude) peuvent être conditionnelles. Nous nous limitons aux règles inconditionnelles. Alors, r_1 et r_2 doivent être inconditionnelles pour être traitées.

Condition d'Application. Il existe r_1 et r_2 deux règles de réécriture inconditionnelles tel que :
metaMatch(M, RuleLeft(r_1), RuleLeft(r_2), nil, 0) =/= noMatch
et metaMatch(M, RuleRight(r_1), RuleRight(r_2), nil, 0) =/= noMatch.

Application. Supprimer r_1.

3.3. Règle 3: Places Equivalentes

Condition Application. Il existe deux places p_1 et p_2 et un nombre naturel strictement positif k tel que:

1. $p_1F \cap p_2F = \phi$

2. Pour chaque $r \in p_1F$ il existe $r' \in p_2F$ et un terme T tel que :

a) Pour chaque $p \in P$, metaMatch(M, RuleRight(r), RuleRight(r'), nil, 0) =/= noMatch.

b) metaMatch(M, GetSubTermExceptPlaces(RuleLeft(r), (p_1 . p_2)),
GetSubTermExceptPlaces(RuleLeft(r'), (p_1 . p_2)), nil, 0) =/= noMatch.

c) metaMatch(M, GetSubTermConcerningPlace(RuleLeft(r), p_1),
GetSubTermConcerningPlace(RuleLeft(r'), p_2), nil, 0) =/= noMatch.

d) GetSubTermConcerningPlace(RuleRight(r), p_1) = < p_1 ; T > .. < p_1 ; T > (k fois).

3. Pour chaque $r \in p_2F$ il existe $r' \in p_1F$ et un terme T tel que:

a) Pour chaque $p \in P$, metaMatch(M, RuleRight(r), RuleRight(r'), nil, 0) =/= noMatch.

b) metaMatch(M, GetSubTermExceptPlaces(RuleLeft(r), (p_1 . p_2)),
GetSubTermExceptPlaces(RuleLeft(r'), (p_1 . p_2)), nil, 0) =/= noMatch.

c) metaMatch(M, GetSubTermConcerningPlace(RuleLeft(r), p_1),
GetSubTermConcerningPlace(RuleLeft(r'), p_2), nil, 0) =/= noMatch.

d) GetSubTermConcerningPlace(RuleRight(r), p_2) = < p_2 ; T > .. < p_2 ; T > (k fois).

4. Pour tout $r \in Fp_1 \cup Fp_2$ et tous les termes T,

a. GetSubTermConcerningPlace(RuleLeft(r), p_1) = < p_1 ; T > .. < p_1 ; T > (n_1*k fois).

b. GetSubTermConcerningPlace(RuleLeft(r'), p_1) = < p_2 ; T > .. < p_2 ; T > (n_2*k fois).

c. GetSubTermConcerningPlace(T_0, p_1) = < p_1 ; T > .. < p_1 ; T > (n_3*k fois).

d. GetSubTermConcerningPlace(T_0, p_2) = < p_2 ; T > .. < p_2 ; T > (n_4*k fois).

Tel que n_1, n_2, n_3 et n_4 sont des nombres naturels strictement positifs.

Application. Pour chaque $r \in p_1F$ de la forme rl T => T' [r] . Soit T un terme de la forme T = < p_1 ; u_1 > .. < p_1 ; u_n > . T_1 (T_1 est le sous-terme de T qui soit indépendant de p_1). Soit T_2 un terme de la forme T_2 = < p_2 ; u_1 > .. < p_2 ; u_n >, Soit T'' = T_2 . T_1. Remplacer la règle r par la règle r' de la forme rl T'' => T' [r'] . For T_0 = < p_1 ; v_1 >.. < p_1 ; v_1 > . ST_0 (ST_0 est le sous-terme de T_0 qui soit indépendant de p_1), le nouveau marquage initial est T_0 = < p_2 ; v_1 > .. < p_2 ; v_1 > . ST_0.

4. Implémentation des Règles de Réduction pour les ECATNets dans Maude

Pour une raison de simplicité, nous présentons quelques fonctions principales de l'application qui sont suffisantes pour expliquer toute l'implémentation. Dans un module dit MODULE-OPERATIONS, nous proposons la fonction GetTransitionsWithoutInputPlaces(M) qui détecte et retourne 'les transitions sans places d'entrée' dans un module M modélisant un ECATNet :

```
op GetTransitionsWithoutInputPlaces :  Module -> RuleSet .
eq GetTransitionsWithoutInputPlaces(M) = SearchTransitionsWithoutInputPlaces(M, getRls(M)) .
```

Cette fonction appelle une autre fonction SearchTransitionsWithoutInputPlaces(M, getRls(M)) pour chercher de telles transitions dans toutes les règles de réécriture de M. Dans notre définition de la 'transition sans places d'entrée', le côté gauche de la transition (règle de réécriture) ne contient qu'une variable de type Marking (la condition

(IsVar(RuleLeft(Rl)) == true) and (getType(RuleLeft(Rl)) == 'Marking) et que cette variable doit figurer aussi dans le côté droit :

```
op SearchTransitionsWithoutInputPlaces : Module RuleSet -> RuleSet .
eq SearchTransitionsWithoutInputPlaces(M, none) = none .
eq SearchTransitionsWithoutInputPlaces(M, Rl Rls) =
 if (IsVar(RuleLeft(Rl)) == true) and (getType(RuleLeft(Rl)) == 'Marking)
    and  (SubTerm(RuleLeft(Rl),  RuleRight(Rl)))
  then Rl SearchTransitionsWithoutInputPlaces(M, Rls)
  else SearchTransitionsWithoutInputPlaces(M, Rls) fi .
```

La fonction RuleLeft(Rl) prend comme paramètre une règle de réécriture décrite dans le méta-niveau et retourne le terme qui figure dans le côté gauche de cette règle. Les règles dans Maude peuvent être conditionnelles ou non. Cette fonction traite les deux cas :

```
op RuleLeft : Rule -> Term .
eq RuleLeft(rl T1 => T2 [label(Q)] .) = T1 .
eq RuleLeft(crl T1 => T2 if CD [label(Q)] .) = T1 .
```

Dans un module dit COMPLETENESS-DECISION, nous proposons l'implémentation de la fonction MoreGeneralTerm. La condition (metaMatch(M, T1, T2, CD, 0) =/= noMatch) décide que le terme T2 n'est que le terme T1 après avoir subir une certaine substitution de ses variables si certaine condition CD est vraie :

```
op MoreGeneralTerm : Module Term Term Condition -> Bool .
eq MoreGeneralTerm(M, T1, T2, CD) =
 if (metaMatch(M, T1, T2, CD, 0) =/= noMatch)
  then true
  else if (CD == nil) and MoreGeneralTerms1(M, GetTermsInPlace(T1), GetTermsInPlace(T2))
        then true  else  false fi  fi .
```

Cependant, la condition (metaMatch(M, T1, T2, nil, 0) =/= noMatch) n'est pas suffisante pour implémenter toute la signification de MoreGeneralTerm. Comme il est décrit dans [Sch97], si on prend par exemple T1 = < p ; a > . < p ; b > (terme sans variable avec a et b des constantes de sorte S) et T2 = < p ; x > . < p ; succ(x) > (x est une variable de sorte S), nous avons dans ce cas metaMatch(M, T1, T2, nil, 0) == noMatch. Cependant, si a et b sont les seuls éléments de la sorte S avec a = succ(b) et b = succ(a), alors normalement la condition de l'application de la règle de réduction est tout à fait vérifiée parce que le multi-ensemble composé de x et succ(x) est toujours égal au multi-ensemble composé de a et b, quelque soit la substitution. Dans ce cas, T1 peut être considéré plus général que T2. Pour cela, nous proposons la fonction MoreGeneralTerms1 qui traite ce cas. Cette fonction prend trois paramètres, un de sorte Module et les deux autres de sorte TermList.

Dans l'expression MoreGeneralTerms1(M, GetTermsInPlace(T1), GetTermsInPlace(T2)), la fonction GetTermsInPlace(T) extrait les termes qui se trouvent dans les places de T, c.-à-d., si T = < p1 ; t1 > .. < pn ; tn >, GetTermsInPlace retourne la liste des termes t1,..,tn.

Sans entrer dans les détails des fonctions appelées par cette fonction, le comportement global de MoreGeneralTerms1 est de considérer pour chaque sorte S, une sous-liste des termes (soit GTL) dans GetTermsInPlace(T1) ayant tous la même sorte S et une sous-liste des termes (soit TL) dans GetTermsInPlace(T2) de la même sorte S. Ces deux sous-listes doivent avoir les mêmes longueurs. GTL ne doit contenir que des termes sans variables et TL peut contenir des termes quelconques, sinon la condition de la règle de réduction n'est pas valide. La fonction MoreGeneralTerms1 vérifie que l'ensemble des termes de GTL (sans répétition) constitue tous les éléments de sorte S (propriété 1). Ensuite cette fonction cherche toutes les substitutions possibles pour ti (i=1,.., n), bien sûr, les variables en commun entre les termes doivent avoir les même valeurs. Par la suite, la fonction montre que pour chaque substitution, GTL et TL sont identiques en termes de leurs contenus (propriété 2). S'il existe une substitution de toutes les variables des termes de TL, de telle manière que GTL et TL sont différentes en termes de leurs contenus, alors la condition de la règle n'est pas vérifiée et MoreGeneralTerms1(M, GTL, TL) retourne faux.

Soit GTL la liste contenant les termes gt1,..,gtn et TL la liste contenant t1,..,tn. Notons qu'après avoir montrer que gt1,..,gtn sont les seuls éléments de S, alors si nous faisons une substitution pour un terme ti,(i=1,..,n), il est forcément égal à un terme gtj (j =1,..,n).

L'implémentation dans Maude de la vérification de la propriété 1 est possible en ce moment grâce à l'outil 'Completeness Checker' proposé par Hendrix dans [Hen05]. Cet outil permet la vérification de la propriété 'complétude suffisante'. Cette propriété indique que la forme canonique de tout terme sans variables est basée uniquement sur les opérations constructrices. La déclaration de la fonction checkCompleteness dans Maude est :

```
op checkCompleteness : Module MembAxSet EquationSet MembAxSet -> ProofObligationSet .
```

Le premier paramètre de cette fonction représente un module, le second représente des sujets, le troisième représente des équations et le quatrième représente des 'membres constructeur' (constructor memberships). La fonction checkCompleteness(M, Subjects, Eqs, Mbs) écarte toutes les équations et les 'membres constructeurs' qui n'unifient pas avec un sujet, mais si un sujet à une correspondance non-conditionnelle, elle l'écarte. Si tous les sujets (subjects) ont une correspondance, alors cette fonction retourne qu'il n'y pas d'obligation de preuve (none.ProofObligationSet).

Pour utiliser cette fonction dans notre définition de MoreGeneralTerm1, nous procédons d'abord à la création d'un nouveau module New-M contenant l'ensemble des termes de sorte S comme les seuls éléments de S. Soit gt1,..,gtn des termes de sorte S, le module qui contient la définition de ses termes comme les seuls éléments de sorte S, doit contenir les définitions des opérations et les membres constructeurs de la manière suivante :

- Si gti (i=1,..,n) est une constante (opération non paramétrée) alors nous déclarons :

```
op gti : -> S .
```

- Si gti (i=1,..,n) n'est pas une constante (opération paramétrée), alors gti est de la forme f(r1,..,rm). Pour l'instant nous acceptons que le cas où r1,..,rm sont des constantes non paramétrées. La déclaration de ti a la forme suivante :

```
op f : S1 .. Sm -> S .
cmb f(x1:S1, …, xm :Sm) : S
if (x1:S1 == r1.S1) and … and (xm:Sm == rm.Sm) .
```

Pour vérifier la propriété précédente, nous définissons la fonction SortCompleteness :

```
op SortCompleteness : Module Module Type -> Bool .
eq SortCompleteness(New-M, M, S) =
if checkCompleteness(New-M, patterns(M, S), getsEqs(New-M), patterns(New-M, S))
== (none).ProofObligationSet  then true else false fi .
```

La fonction patterns(M, S) cherche les termes sans variables qui constitue l'algèbre initiale pour la sorte S. En cas d'infinité, cette fonction retourne des termes 'modèle' couvrant tous les éléments de la sorte S. la fonction SortCompleteness cherche pour chaque terme de sorte S dans M, s'il existe un terme de la même sorte dans New-M, de telle manière qu'il y'a une correspondance entre les deux termes. S'il existe un terme sans variables dans M et pour chaque terme dans New-M, il n'existe pas de correspondance entre les deux, alors la condition n'est pas vérifiée. Dans ce cas, les éléments définis dans New-M ne sont pas les seuls éléments de S. Le module qui implémente la règle de réduction 'transitions sans places d'entrée' est relativement petit, nous le présentons en entier :

```
mod TRANSITIONS-WITHOUT-INPUT-PLACES is
protecting COMPLETENESS-DECISION .
op IsReducedTransitionsWithoutInputPlaces :     Module -> Bool .
op ModuleAfetrTransitionsWithoutInputPlacesReduction : Module -> Module .
op RecursiveModuleAfterTransitionsWithoutInputPlacesReduction :  Module -> Module .
var M : Module .
eq IsReducedTransitionsWithoutInputPlaces(M) =
if GetTransitionsWithoutInputPlaces(M) =/= none
****** 1ère condition de la règle
and
MoreGeneralTerm(M,
RulesRightAddition(GetTransitionsWithoutInputPlaces(M)),  RulesLeftAddition(GetOutputTransitionsOfManyPlaces(
  GetOutputPlaceOfManyTransistion(M,    GetTransitionsWithoutInputPlaces(M)), M)), nil) == true
****** 2ème condition de la règle
and
RulesConditionsConcatenation(GetTransitionsWithoutInputPlaces(M)) == nil
****** Règles à retirer doivent être inconditionnelles
then true  else false   fi .

eq ModuleAfetrTransitionsWithoutInputPlacesReduction(M) =
 if IsReducedTransitionsWithoutInputPlaces(M) == true
  then DeleteTransitionsInModule(DeletePlacesAndConnectedArcsInModule(M,
        GetOutputPlaceOfManyTransistion(M,
        GetTransitionsWithoutInputPlaces(M))),  GetTransitionsWithoutInputPlaces(M))
  else M  fi .
```

```
eq RecursiveModuleAfterTransitionsWithoutInputPlacesReduction(M) =
if IsReducedTransitionsWithoutInputPlaces(M) == true
 then RecursiveModuleAfterTransitionsWithoutInputPlacesReduction(
        ModuleAfetrTransitionsWithoutInputPlacesReduction(M))
 else M  fi .
endm
```

La fonction ModuleAfetrTransitionsWithoutInputPlacesReduction(M) supprime toutes les règles de réécriture (transitions) qui vérifient la condition d'application de la règle de réduction du module. Après avoir appliquer cette fonction une fois, quelques règles de réécriture peuvent devenir sans places d'entrée et elles vérifient de nouveau la condition d'application de la règle de réduction. Pour cela, nous avons développé une autre fonction RecursiveModuleAfterTransitionsWithoutInputPlacesReduction(M). Cette fonction appelle la fonction précédente appliquée sur le module résultant chaque fois qu'il y a des transitions satisfaisant la condition d'application de la règle de réduction modélisée par la condition IsReducedTransitionsWithoutInputPlaces(M) == true.

5. Application de l'Outil sur un Exemple

Nous reprenons l'exemple du robot présenté au troisième chapitre. Cet exemple contient au départ une transition sans place d'entrée. Pour l'application de l'outil sur l'exemple, nous appelons la fonction RecursiveModuleAfterTransitionsWithoutInputPlacesReduction(META-ROBOT). Comme illustré dans la figure 1, L'outil élimine une seule transition (règle de réécriture) dans l'ECATNet. Nous remarquons que l'intérêt de l'application de cette règle décrit dans la section 3 est rempli pour cet exemple. Initialement, l'ECATNet est non-borné en raison de la présence de la transition sans places d'entrée E. Après l'application de cette règle, l'ECATNet obtenu est borné. Par conséquent, certaines méthodes d'analyse comme le Model Checking de Maude peuvent être appliquées sur ce réseau.

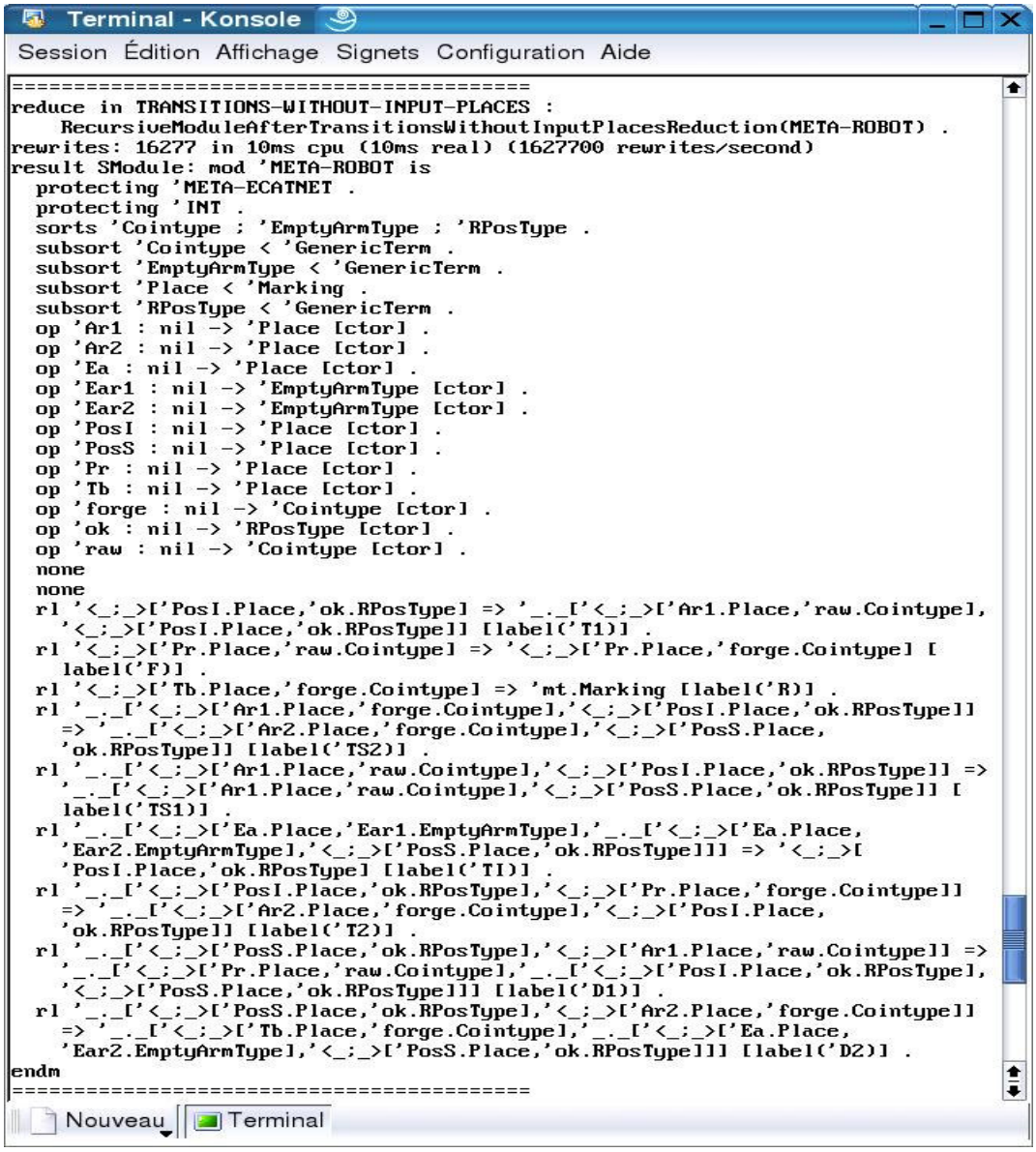

Figure 1. Réduction de META-ROBOT après l'application récursive de la règle 'transitions sans places d'entrée'

6. Conclusion

Dans ce chapitre, nous avons proposé un outil basé sur Maude qui met en application quelques règles de réduction pour les ECATNets. De telles règles de réduction sont définies au préalable dans [Sch97] pour les RPAs. Nous avons présenté trois règles et nous avons indiqué comment nous les avons adaptées pour les ECATNets. Pour la simplicité, nous avons montré l'implémentation dans Maude d'une seule règle. Grâce à l'intégration des ECATNets dans Maude et la réflectivité de ce langage, la création d'un outil implémentant les règles de réduction pour les ECATNets en utilisant Maude est largement simplifiée par rapport à sa création en utilisant un autre langage. Nous avons montré dans ce chapitre que l'implémentation en utilisant Maude des règles de réduction n'est pas compliquée. Peu de lignes de code sont suffisantes pour implémenter de telles règles de réduction. En fait, l'utilisation des services offerts par Maude pour gérer les méta-modules, nous a libéré de plusieurs détails.

Partie 3 : Analyse des Programmes Ada à l'aide des ECATNets

Cette dernière partie concerne la translation d'un programme Ada vers les ECATNets et l'application de certains outils des ECATNets pour la vérification de ce programme. Dans le premier chapitre, nous décrivons notre approche systématique de translation de certains concepts de langage Ada vers les ECATNets. Le chapitre 2 décrit une représentation compacte des Ada-ECATNets. Le chapitre 3 concerne notre implémentation de la translation de Ada-ECATNets en utilisant le langage Maude. Nous expliquons l'utilisation de quelques outils d'analyse des ECATNets pour l'analyse des programmes Ada dans le chapitre 4.

Nous utilisons le même exemple d'un programme Ada pour expliquer notre démarche de translater et d'analyser les programmes Ada à l'aide des ECATNets. Cet exemple est à propos du problème connu du producteur consommateur. Dans le premier chapitre de cette partie, nous traduisons cet exemple d'Ada à un ECATNet. Dans le second chapitre, nous montrons comment obtenir un Ada-ECATNet plus compact du problème producteur consommateur par rapport à Ada-ECATNet obtenu dans le premier chapitre. Ceci grâce aux règles de réduction que nous présentons dans le chapitre 2. Dans ce même chapitre, nous remarquons que l'Ada-ECATNet compact obtenu peut subir une autre réduction en appliquant la règle 'parallèles places' définie pour les réseaux de Petri algébriques et adaptée aux ECATNets. Pour confirmer les gains obtenus grâce à cette double réduction, nous appliquons certaines méthodes d'analyse avant et après l'application des règles du raffinement définies aux Ada-ECATNets et les règles de réduction adaptées aux ECATNets dans le chapitre 3. Les méthodes d'analyse prises en compte dans ce chapitre sont : la simulation, l'analyse d'accessibilité et le Model Checking. Cette application nous permet de voir les gains obtenus en termes d'efficacité de l'exécution. Nous concluons cette partie par un quatrième chapitre où nous décrivons notre compilateur Ada-ECATNet. Nous expliquons plus au moins en détail les différentes phases de ce compilateur qui est développé en grande partie en utilisant le paradigme fonctionnel du langage Maude.

Chapitre 1 :
Translation des Tâches Ada vers les ECATNets

1 Introduction

L'une des caractéristiques du langage de programmation Ada les plus attractives est la notion 'tâche', qui permet l'exécution concurrente dans des programmes de ce langage. La présence de la concurrence complique considérablement l'analyse, le test et le déboguage du code. L'expression de la concurrence est réalisée par les concepts de 'tâche' et de 'rendez-vous' dans Ada. Pour cela, beaucoup d'effort est concentré sur ces mécanismes. Les techniques de détection dans les tâches d'Ada s'étendent de l'analyse statique, qui tire des conclusions sur le comportement d'exécution du programme en regardant son code sans exiger n'importe quelle exécution du programme source, à l'analyse dynamique, qui consiste à choisir un ensemble de données d'entrée représentatives pour tester le programme. L'analyse dynamique a son point faible en évaluant la représentativité des essais et ainsi dans la généralité des conclusions tirées d'un ensemble de cas d'essai. L'analyse statique semble plus intéressante, malgré son incapacité d'explorer des caractéristiques des programmes dont le comportement dépend fortement des données d'entrée. On l'accepte bien que l'assurance de la qualité des systèmes concurrents exige les méthodes formelles [Bli00].

Dans ce chapitre, nous proposons un cadre de réseau de Petri basé sur les ECATNets pour traduire systématiquement le comportement des tâches Ada. A notre sens, les ECATNets présentent une puissance d'expression assez élevée pour décrire plusieurs concepts du langage Ada. D'un autre côté, les ECATNets ont une batterie intéressante des outils d'analyse, comme les règles de réduction [Bou06a], [Bou06b] et le Model Checking de Maude. Le cadre est illustré par un exemple.

Le reste de ce chapitre est organisé comme suit : La section 2 présente quelques travaux similaires à notre travail. La section 3 donne une vue sur le processus de traduire une 'tâche' d'Ada dans les ECATNets. La section 4 donne un exemple d'application. L'application du processus de translation d'Ada-ECATNet sur un exemple est donnée dans la section 5. La section 6 conclut le chapitre.

2. Travaux Similaires

Dans le contexte de l'analyse des programmes Ada, les approches proposées sont basées sur un certain flot ou réseau de Petri. L'objectif principal de ces travaux est la vérification des propriétés de concurrence comme la détection de deadlock ou la vivacité. Nous mentionnons dans cette section quelques uns de ces travaux. Dans [Tay83], Taylor présente un analyseur statique pour Ada. L'analyseur reçoit le code de programme Ada comme entrée et produit son "histoire de concurrence" ("concurrency history") (un graphe dont les noeuds représentent "les états concurrents" possibles et dont les arcs représentent des transitions entre les états). L'histoire de concurrence permet d'obtenir les informations sur des rendez-vous, des deadlocks, et des actions parallèles. La méthode d'analyse basée sur l'histoire de concurrence a une complexité exponentielle en termes de temps (en termes de nombre de tâches). À partir de l'observation que "la liste de des états concurrents et des états successeurs peuvent être interprétés comme graphe

d'accessibilité produit d'un modèle de réseau de Pétri ", Shatz et Cheng [Sha88] transforment le graphe de flot pour dériver un réseau de Petri (appelé Ada-net) équivalent au programme et pour produire le graphe d'accessibilité correspondant qui décrit le comportement potentiel du programme original. Puis, le graphe d'accessibilité est examiné pour détecter des propriétés du programme. En utilisant des réseaux de Petri, Shatz et Cheng gagnent la possibilité d'analyser le programme à l'aide de tous les méthodes et outils qui ont été déjà développés pour les réseaux de Petri. Pour faire face à ce problème, deux approches sont proposées. La première est la compositionalité (stratégie de diviser et conquérir) [Dwy96], [Ged99]: les propriétés appropriées du système entier sont obtenues à partir des propriétés correspondantes de ses plus petits et plus simples composants. La seconde approche est la théorie de structure ; elle consiste à obtenir des informations sur le comportement du modèle directement à partir de la structure de sa bipartite fondamentale et son digraphe évalué. Le marquage initial est considéré comme paramètre. Deux types d'analyse structurale peuvent être distingués : 1) L'analyse algébrique, où la structure du réseau est représentée par sa matrice d'incidence. 2) L'analyse théorique du graphe, où le comportement du réseau est lié à la relation de flot des sous-réseaux produits par les sous-ensembles remarquables de places, tels que les deadlocks et les verrous (traps). Dans [Mur89], une méthode de détection statique de deadlock pour le programme de tâches Ada est décrite. D'abord, un programme Ada est traduit en un Ada-net. Puis, le Ada-net est analysé pour détecter l'existence des deadlocks statiques basés sur l'analyse structurale (utilisation de T-invariants, S-invariants) et l'analyse dynamique (utilisation du graphe d'accessibilité). Dans [Bar97], [Bar98], les auteurs utilisent la structure des siphons à fin de préciser quelques conditions structurelles pour les propriétés comportementales comme l'absence de deadlock et la vivacité que les programmes Ada concurrents corrects doivent satisfaire. Dans la littérature, il existe aussi quelques méthodes d'analyse de dépendance des programmes d'Ada. Dans [Che96], une méthode en temps d'exécution de détection des deadlocks basée sur le graphe Task-Wait-For est présentée. Dans [Che97], les auteurs présentent un modèle de représentation des programmes Ada concurrents, appelé 'Task Dependence Net' (TDN). En réalité, TDN est un arc classifié di-graph pour représenter explicitement les programmes de dépendance primaires dans les programmes concurrents ou séquentiels. Dans [Bru99] un travail basé sur le formalisme de réseaux de Petri coloré qui automatise la vérification des propriétés concurrentes de programme Ada est présenté. Le réseau de Petri est automatiquement produit par une étape de traduction et la vérification est automatiquement effectuée sur le réseau avec des techniques relatives classiques. Dans [Bur99], les auteurs utilisent les techniques du Model Checking pour une vérification d'un programme Ada. Dans [Bli00] un contexte d'analyse de flux de données symboliques pour détecter des deadlocks dans des programmes Ada en ce qui concerne les tâches est présenté. Dans [Che02] une technique d'analyse (appelée l'analyse de dépendance de programme) pour identifier et déterminer de diverses dépendances de programme en codes sources de programme. Dans [Liu02], les auteurs utilisent l'algèbre de processus pour détecter le deadlock dans les rendez-vous d'Ada.

Vue l'importance de l'analyse des programmes concurrents et comme nous le remarquons, plusieurs approches ont été proposées depuis plus deux décennies. Chacune de ces approches possède certains points forts mais beaucoup de choses restent à faire dans l'avenir.

3. Traduction de Tâche Ada dans les ECATNets

Nous donnons ici une vue sur notre traduction systématique des tâches Ada dans les ECATNets.

Types. Ils sont traduits aux sortes dans les ECATNets.

Variables. Elles sont traduites aux termes algébriques (variables) des sortes représentant des types dans Ada.

Tâche. Dans Ada, une tâche est définie comme un type spécial : type actif. Le type 'tâche' est représenté par un terme algébrique et son aspect dynamique est représenté dans ce cas par un ECATNet. Nous choisissons un n-uplet contenant le terme algébrique 'tâche' et les termes algébriques 'variables locales' de cette tâche.

Instructions.

Instruction. Elle est représentée par une transition.

Instruction d'Affectation. Il est possible de représenter n'importe quelle affectation générale à l'aide d'un ECATNet. La transition dans la figure 1 est la translation de l'affectation x:= exp.

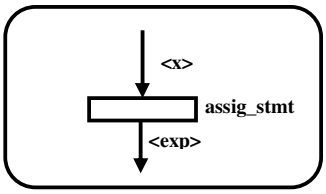

Figure 1. Représentation d'une instruction d'affectation dans Ada par un ECATNet

Instruction Conditionnelle. Cette instruction est représentée par deux transitions en conflit. Une transition pour orienter le contrôle vers la partie du code quand la condition est vraie et l'autre pour l'orienter vers la partie du code si la condition est fausse. La condition est une expression booléenne qui peut être décrite par la condition de la transition. La figure 2 montre un ECATNet résultant de la traduction de l'instruction if c then I1 else I2 endif.

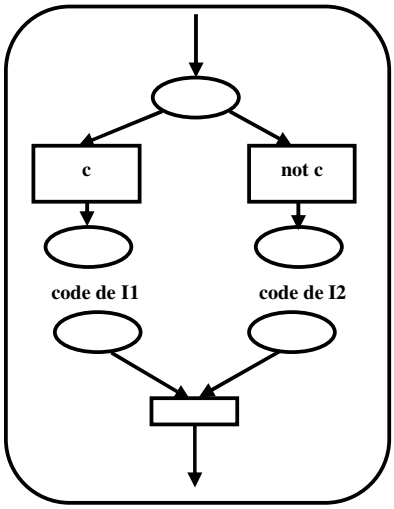

Figure 2. Représentation de l'instruction if-then-else dans Ada par un ECATNet

Boucle (Loop). À la fin du code de la boucle, nous ajoutons un arc vers le début de ce code.

Appel de Procédure. Nous représentons des procédures de type 'void' par des termes. Quand une procédure (ou tâche) appelle une autre, la procédure appelante ne peut pas passer ses variables locales à l'appelée. Les variables de chaque procédure (ou tâche) sont locales et ne peuvent pas être modifiées par une autre procédure (ou tâche). La transition représentant l'appel d'une procédure doit mettre au début de la procédure appelée le n-uplet composé de son nom, ses paramètres et ses variables locales et mettre dans une autre place le n-uplet représentant la procédure appelante. Pour faire cela, nous exprimons l'état d'attente de cette procédure. S'il y a un appel par valeur, il est nécessaire de mettre les noms des variables utilisées dans l'appel dans la position du paramètre correspondant dans le n-uplet au moment de l'appel. Si P appelle une procédure P1(in v1: t1, out v2,: t2,…) par P1(x,…), nous obtenons l'ECATNet de la figure 3.

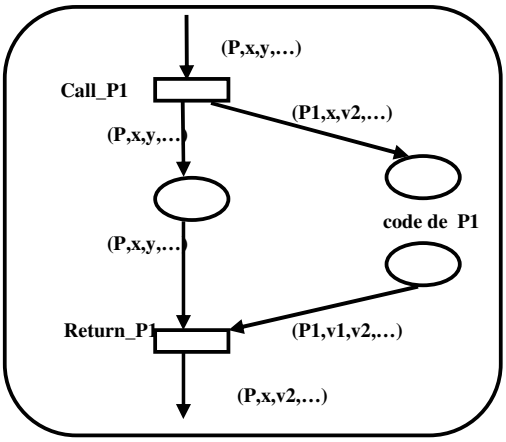

Figure 3. Représentation d'un appel de procédure dans Ada par un ECATNet

Tâches et Communications entre Tâches.

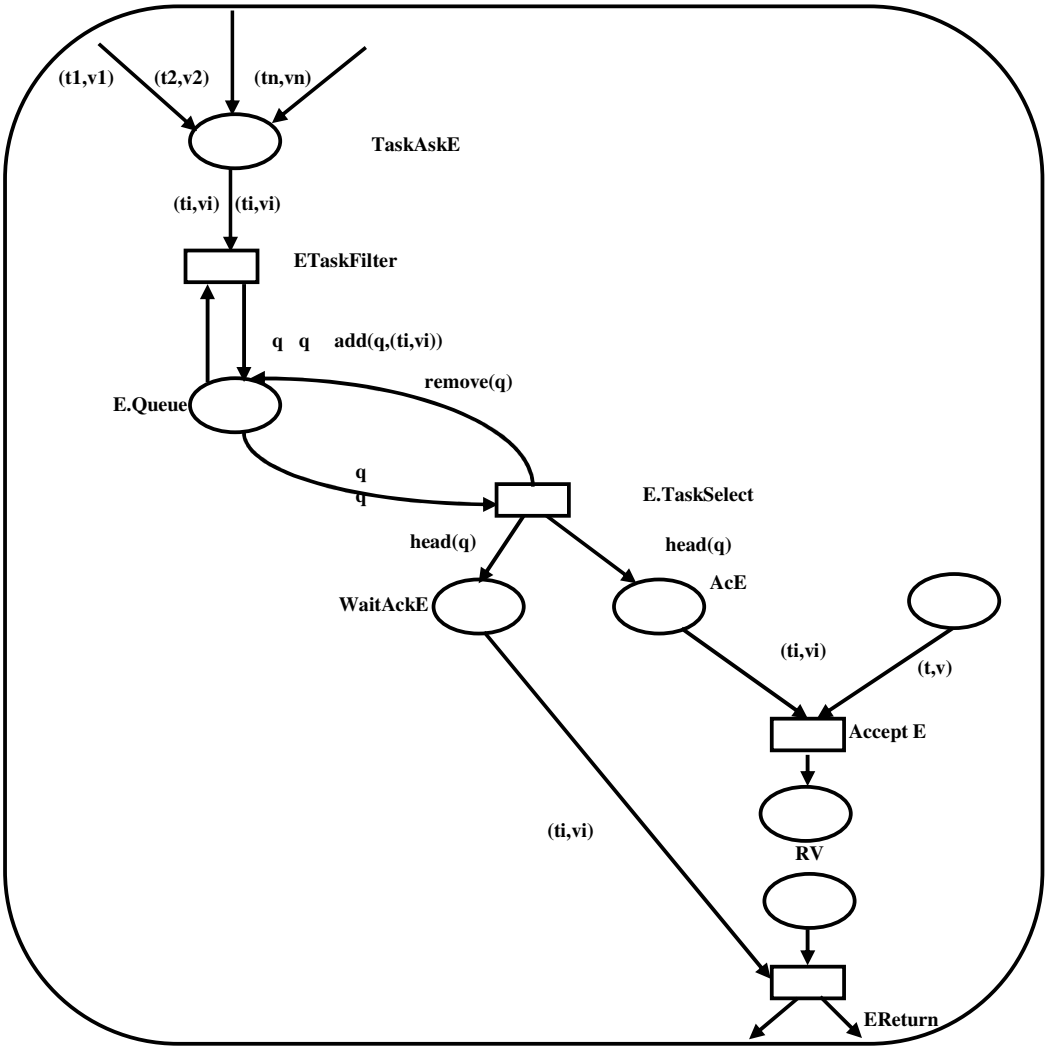

Figure 4. Représentation d'une entrée de tâche dans Ada par un ECATNet

Entrée (Entry). Pour une entrée de tâche, nous associons deux places pour contrôler la file d'attente qui contient les tâches en attente qui ont appelé cette entrée. Une place sert à contrôler l'ordre d'arrivée des tâches et elle doit avoir la taille maximale de supporter une seule tâche. Celle-ci doit être transférée à la file d'attente de l'entrée qui est dans l'autre place. Pour cela, nous faisons appel à la définition en termes de spécification algébrique de la file d'attente :

- q est une variable de type Queue.
- head(q) est la fonction qui retourne la tâche en tête de q.
- add(q, (ti, vi)) est la fonction qui ajoute la tâche (ti, vi) à q.
- remove(q) est la fonction qui retire la première tâche de q.
- l'attribut count correspond à la fonction qui retourne la taille de la file d'attente. Nous pouvons définir sa structure de la même manière: count(q).

Pour une entrée avec une garde, nous traduisons cette garde à une condition de la transition correspondante. La figure 4 est un ECATNet qui traduit une entrée E d'une tâche t. Les places TaskAskE et AcE doivent avoir une capacité maximale d'un seul jeton. Nous avons une condition supplémentaire isempty(q) = false pour la transition E.TaskSelect.

Types Protégés. Pour la traduction d'un type protégé, nous créons une place contenant ses variables. Ces dernières attendent pour être traités par une fonction, une procédure ou une entrée. Pour chacune de trois actions, nous associons une transition en conflit avec les autres transitions. Pour chaque fonction, procédure ou entrée, il est nécessaire d'avoir une place contenant les tâches demandant son exécution.

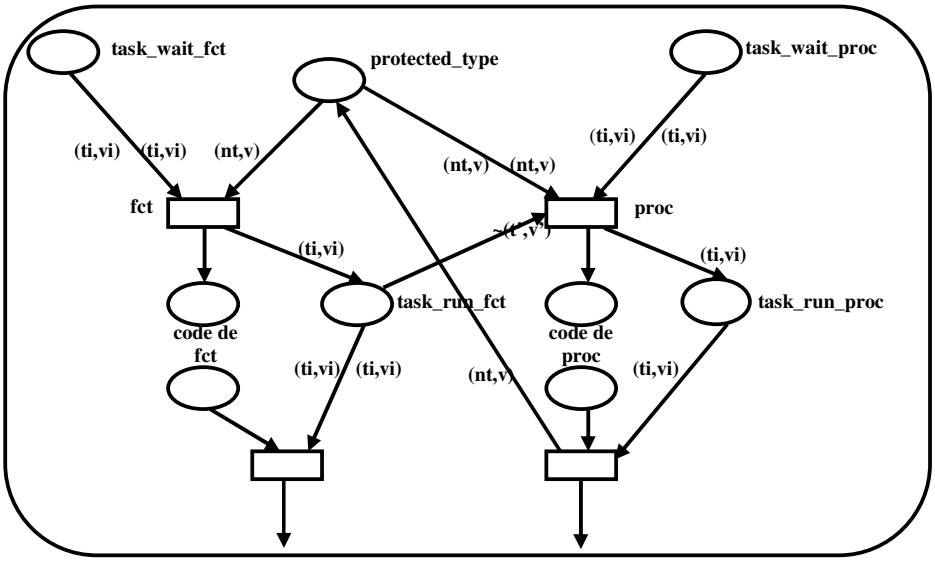

Figure 5. Représentation d'une fonction et d'une procédure de type protégé dans Ada par un ECATNet

L'exécution de la transition qui représente le début du code de fonction par une tâche doit retirer la tâche exécutrice, et elle doit examiner la présence des variables du type dans leurs places. Elle assure qu'aucune procédure ou une entrée n'est en évolution. La transition ne retire pas de telles variables pour permettre à une autre tâche l'exécution du code de la fonction. La transition permettant d'entrer dans le code de la fonction, doit mettre la tâche exécutrice dans une place particulière qui sert à savoir quelles sont les tâches en cours d'exécution de cette fonction. L'exécution de la transition représentant l'entrée vers le code d'une procédure doit retirer les variables du type, la tâche exécutrice et elle doit examiner s'il n'y a aucune fonction en cours d'exécution. En ce qui concerne une procédure, l'entrée gère une file d'attente et probablement une garde. La transition qui permet d'atteindre l'exécution du code d'une fonction, d'une procédure ou d'une entrée doit insérer les variables du type privé, les paramètres de la fonction, en outre, de la

procédure ou de l'entrée respectivement et l'identité de la tâche exécutrice. D'autre part, il est nécessaire de faire la correspondance entre les paramètres de tâche et ceux de la fonction, de la procédure ou de l'entrée. Les figures 5 et 6 montrent un ECATNet représentant l'exécution d'une fonction, d'une procédure ou d'une entrée par une tâche. La séparation est faite pour contrôler la complexité du diagramme obtenu.

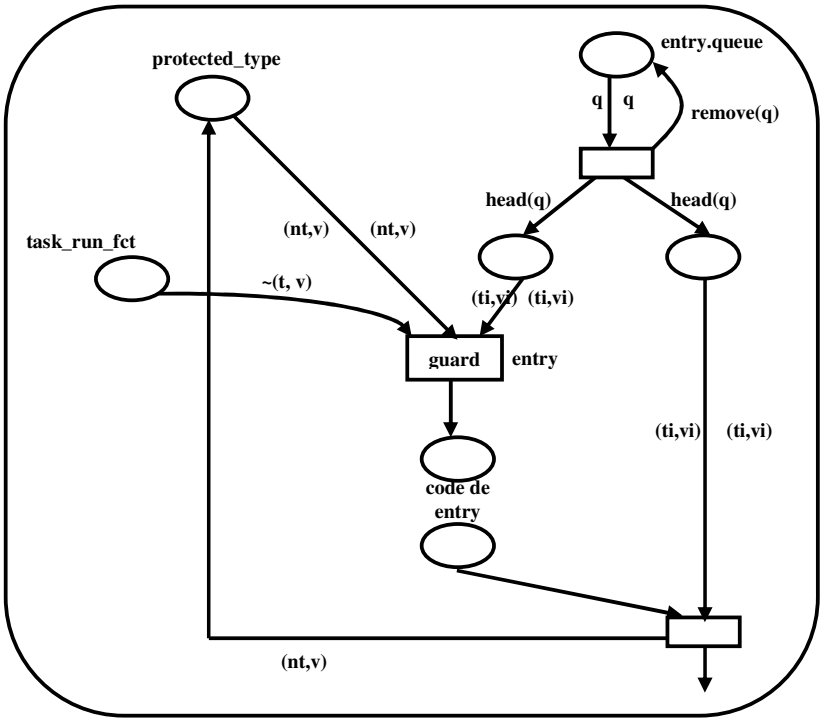

Figure 6. Représentation d'une entrée de type protégé dans Ada par un ECATNet

4. Exemple d'Application

L'exemple suivant définit un 'buffer' atteint par une tâche productrice et une autre consommatrice. La tâche productrice peut avoir la structure suivante :

```
task  Producer
task body Producer
Char : Character;
begin loop  … -- produire le prochain caractère Char
Buffer.Write(Char) ;  exit when Char = ASCII.EOT ;
end loop;  end Producer;
```

La tâche consommatrice peut avoir la structure suivante :

```
task Consumer
task body Consumer is
Char : Character;
begin loop Buffer.Read(Char); exit when Char = ASCII.EOT;  … -- Consommer le prochain caractère Char
end loop ; end Consumer ;
```

Le 'buffer' contient un 'Pool' interne des caractères contrôlés. Cet espace a deux indices, In_Index dénotant l'endroit du prochain caractère entrant et Out_Index dénotant l'endroit du prochain caractère sortant :

protected Buffer is entry Read(C : out character) ; entry Write(C : in character);
private Pool : Array[1..10] of character; Count : Natural := 0 ; In_Index, Out_Index : positive := 1; end Buffer;

protected body Buffer is
entry Write(C : in character) when Count < Pool'length is
begin Pool(In_Index) := C; In_Index := (In_Index mod Pool'length) + 1; Count := Count + 1; end write;
entry Read(C : out character) when Count > 0 is
begin C : = Pool(out_Index); Out_Index := (Out_Index mod Pool'length) + 1; Count := Count – 1; end Read;
end Buffer;

4.1. Translation de l'Exemple dans les ECATNets

Nous définissons les types algébriques tableau, naturel, et le type positive. Autour de ces types algébriques, nous définissons les fonctions suivantes :

- read(a, x): la fonction qui retourne l'indexe de l'élément x dans le tableau a.
- store(a, x, n): la fonction constructrice d'un tableau. L'élément x est dans la position indexée par n.
- set(a, x, n): la fonction qui insère l'élément x dans la position indexée par n dans le tableau a.
- get(a, n): la fonction qui retourne l'élément qui est dans la position indexée par n dans le tableau a.

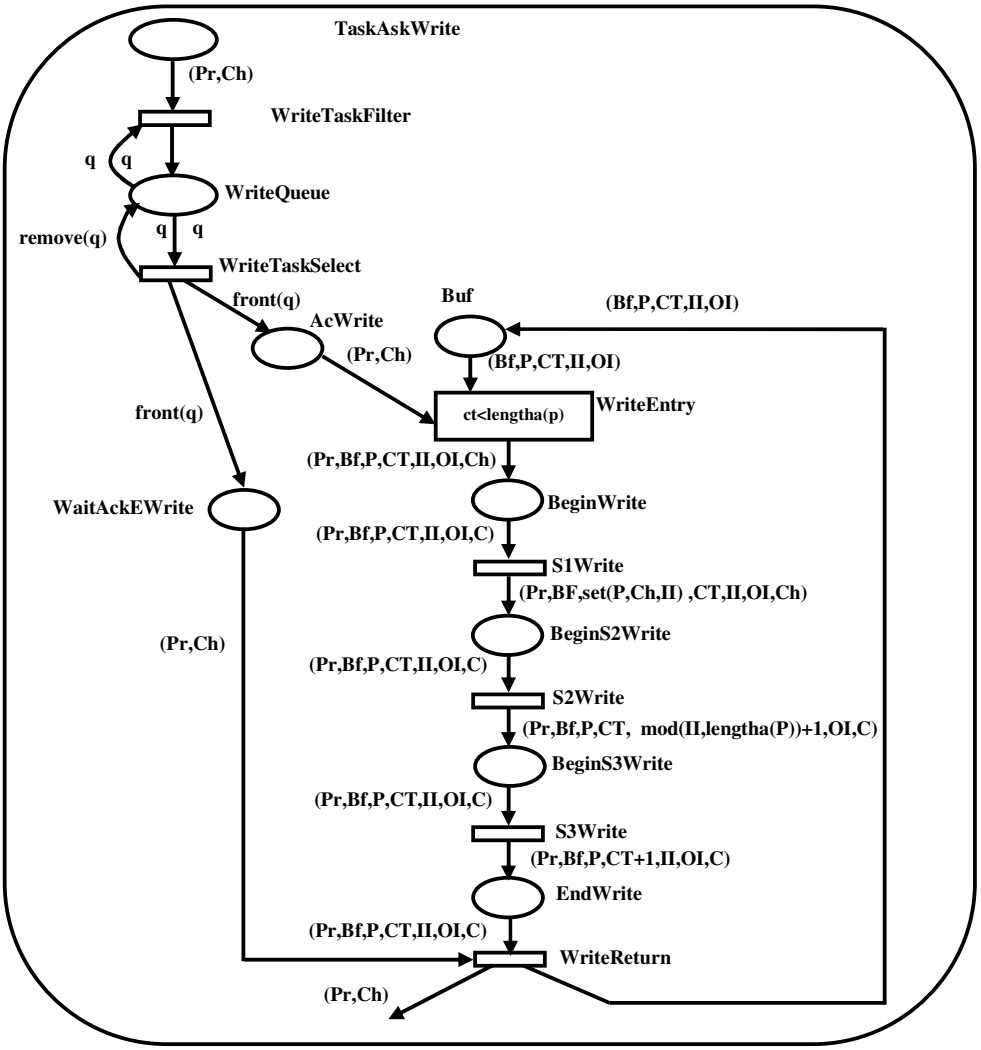

Figure 7. Représentation de l'entrée Write du type Buffer par un ECATNet

La translation de l'entrée Write donne l'ECATNet dans la figure 8. Les termes algébriques dans cette figure désignent les éléments du programme de la manière suivante : Pr : 'task Producer', BF: Buffer, P: Pool, CT: Count, II: In_Index, IO: Out_Index.

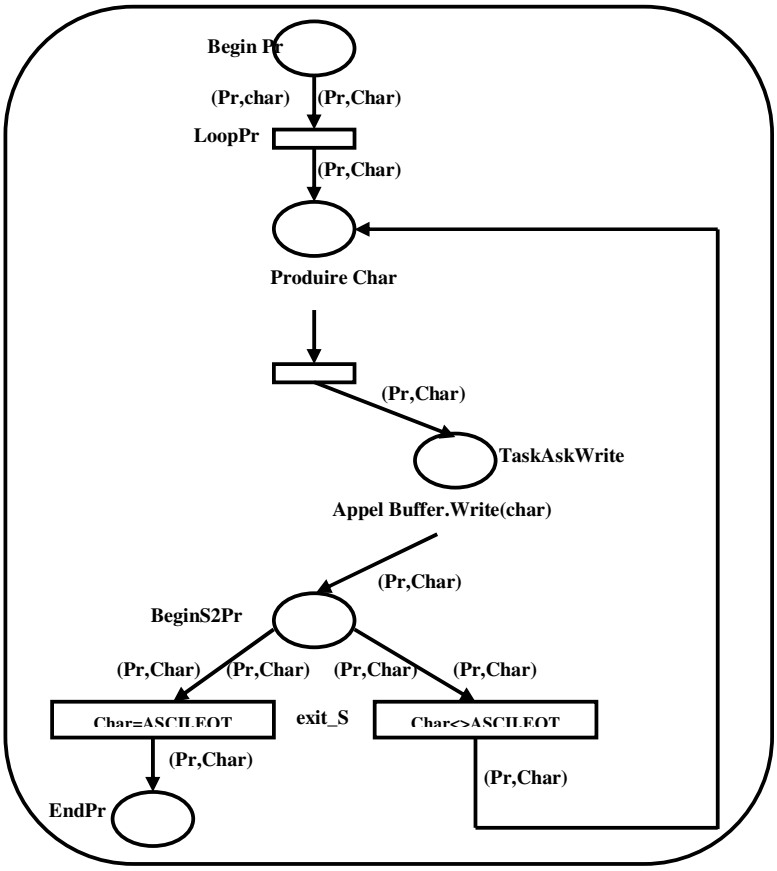

Figure 8. Représentation de la tâche productrice par un ECATNet

4.2. Modélisation de l'Exemple dans la Logique de Réécriture

Nous faisons appel à Maude pour décrire l'exemple précédent. Nous prenons l'ECATNet résultant de la traduction des comportements des tâches et l'objet protégé. Nous considérons la production comme création d'une chaîne de caractères et la consommation comme lecture d'une autre chaîne de caractères. Pour la simplicité, nous donnons seulement une partie de la description.

Dans un nouveau module dit TUPLE, nous définissons le type Tuple avec Exp est l'élément de base de Tuple. Ce type nous permet de créer le n-uplet composé de l'identificateur d'une tâche et ses variables. Le module générique TUPLE appelle le module ARRAY qui est le dernier module dans l'hiérarchie des modules fonctionnels translatant certains types de données du langage Ada. Notons que pr est une abréviation de protecting. Nous définissons une hiérarchie des modules fonctionnels implémentant des types de données algébriques translatant certains types de données du langage Ada. Le module fonctionnel ARRAY qui est le dernier dans cette hiérarchie, définit le type de données traduisant le type tableau du langage Ada.

```
fmod TUPLE is
pr ARRAY .
sorts Exp Tuple .
```

```
subsort Exp < Tuple .
op _,_ : Exp Tuple -> Tuple .
endfm
```

Nous définissons un module fonctionnel PRODUCER pour le type 'task Producer'. Un seul élément Pr de ce type est défini.

```
fmod PRODUCER is
  sort Producer .  op Pr : -> Producer .
endfm
```

Maintenant, nous donnons la description d'un module dit BUFFER-DATA. Ce dernier contient certaines opérations de base pour définir les données relatives au Buffer. Le module DATA-TYPES est le dernier module dans une hiérarchie contenant, entre autres, les modules TUPLE, QUEUE et GENERIC-ECATNET. L'expression subsort Buffer < Exp précise que le type Buffer est sous-sorte de type Exp. Ceci permet de mettre les éléments de type Buffer dans une file d'attente. L'opération _,,_ construit l'élément EltQ (Element Queue) comme un n-uplet pour pouvoir mettre une donnée de la forme d'un n-uplet dans une file d'attente. L'opération 1st retourne le premier élément dans un n-uplet construisant un élément de type EltQ.

```
fmod BUFFER-DATA is
  pr DATA-TYPES .  pr PRODUCER .  pr CONSUMER .
  sort Buffer .   sort Element .
  subsort Buffer < Exp .   subsort Producer < Exp .   subsort Consumer < Exp .
  subsort Array < Exp . subsort Int < Exp .
  vars E E1 E2 : Exp .   var q : Queue .
  op _,,_ : Exp EltQ -> EltQ .
  op 1st : EltQ -> Exp .
  eq 1st((E1 ,, E2)) = E1 .
...
endfm
```

Nous utilisons un module système pour décrire cet ECATNet et nous aurons le module décrit dessous. La formule subsort Queue < Tuple permet d'avoir un élément de type Queue comme un élément de sorte Tuple. WhereCharList est la place qui contient les caractères produits par un producteur et à être déposés dans le buffer. CharListResult est la place utilisée par le consommateur pour déposer les caractères consommés. Une expression de la forme < BeginPr ; (Pr, Char) > signifie que la place BeginPr contient le n-uplet (Pr, Char).

```
mod BUFFER is
 pr BUFFER-DATA .
 subsort Queue < Tuple . subsort List < Tuple . subsort Array < Tuple .
 op BF : -> Buffer . ops BeginPr BeginS1Pr BeginS2Pr EndPr InitialCharList : -> Place .
 ops TaskAskWrite WriteQueue AcWrite WaitAckEWrite
       BeginWrite BeginS2Write BeginS3Write EndWrite : -> Place .
 var P : Array .  var q : Queue .  vars CT II OI : Nat .
```

```
...
*** Règles de réécriture pour Producer
rl [loopPr] : < BeginPr ; (Pr , Ch) > => < BeginS1Pr ; (Pr , Ch) > .
rl [S1Pr] : < InitialCharList ; CharL > . < BeginS1Pr ; (Pr , Ch) > =>
            < InitialCharList ; tail(CharL) > . < TaskAskWrite ; (Pr , head(CharL)) > .
crl [S21Pr] : < BeginS2Pr ; (Pr , Ch) > => < EndPr ; (Pr , Ch) > if Ch == EOT .
crl [S22Pr] : < BeginS2Pr ; (Pr , Ch) > => < BeginS1Pr ; (Pr , Ch) > if Ch =/= EOT .

*** Règles de réécriture pour l'entrée Write
rl [WriteTaskFilter] : < TaskAskWrite ; (Pr , Ch) > . < WriteQueue ; q >  => < WriteQueue ; addq(q, (Pr ,, Ch)) > .

crl [WriteTaskSelect] : < WriteQueue ; q >
 => < WriteQueue ; remove(q) > . < AcWrite ; (front(q)) > . < WaitAckEWrite ; (front(q)) >     if isemptyq(q) == false .

crl [WriteEntry] : < Buf ; (BF , P , CT , II , OI) > . < AcWrite ; (Pr ,, Ch) >
 => < BeginWrite ; (Pr , BF , P , CT , II , OI , Ch) > if CT < lengtha(P) .

rl [S1Write] : < BeginWrite ; (Pr , BF , P , CT , II , OI , Ch) >
 => < BeginS2Write ; (Pr , BF , set(P, Ch, II) , CT , II , OI , Ch) > .

rl  [S2Write] : < BeginS2Write ; (Pr , BF , P , CT , II , OI , C) >
 => < BeginS3Write ; (Pr , BF , P , CT , ((II rem lengtha(P)) + 1) , OI , C) > .

rl [S3Write] : < BeginS3Write ; (Pr , BF , P , CT , II , OI , C) >
 => < EndWrite ; (Pr , BF , P , (CT + 1) , II , OI , C) > .

rl [WriteReturn] : < EndWrite ; (Pr , BF , P , CT , II , OI , C) > . < WaitAckEWrite ; (Pr ,, Ch) >
 => < Buf ; (BF , P , CT , II , OI) > . < BeginS2Pr ; (Pr , Ch) > .
…
endm
```

5. Conclusion

Nous avons vu dans ce chapitre, la faisabilité de la traduction des concepts du langage Ada aux ECATNets. Les ECATNets sont motivés par leur force pour décrire l'aspect statique et dynamique d'un système. Ils permettent de traduire directement et confortablement les programmes Ada. Intuitivement, les Ada-ECATNets obtenus ont un nombre réduit et minimal des places et des transitions. En général, chaque instruction est traduite à une transition. Par exemple, l'instruction *if-then* dans Ada correspond à une seule transition dans les ECATNets. La condition de cette instruction est supportée par la condition de la transition. Ceci signifie que nous n'ajoutons pas d'autres places ou d'autres transitions pour traduire la condition.

Chapitre 2 :
Une Représentation Compacte d'Ada-ECATNet
en utilisant la Logique de Réécriture

1. Introduction

Les programmes concurrents ainsi que les Ada-nets sont en général très complexes et toute proposition d'une réduction de la complexité des algorithmes d'analyse de ces Ada-nets est toujours la bienvenue. Dans toutes les approches existantes concernant l'utilisation des réseaux de Petri (simple ou de haut niveau) dans la description et la vérification des programmes Ada, nous avons noté que ces travaux visent d'abord à traduire des programmes Ada aux réseaux de Petri, puis appliquer des règles de réduction sur les Ada-nets obtenus. Cependant, dans notre travail, les règles de réduction proposées peuvent être faites pendant l'étape de traduction. L'idée de base de cette réduction consiste à 'sauter' certains états intermédiaires qui ne sont pas nécessaires pour la vérification des propriétés liées à la concurrence. Les Ada-ECATNets compacts obtenus sont équivalents aux Ada-ECATNets proposés dans le chapitre précédent et dans [Bou04a]. De telle traduction ne réduit pas seulement la taille du programme, mais elle réduit efficacement le nombre d'états de ce programme en exécution. Nous pouvons représenter deux ou plusieurs instructions dans un bloc séquentiel en utilisant une seule transition dans les ECATNets. Ceci permet de diminuer considérablement le nombre des étapes de réécriture dans le programme approprié dans Maude. Ainsi, l'espace mémoire et le temps d'exécution du programme de Maude sont réduits. Nous confirmerons une telle déduction à travers un exemple dans le chapitre suivant. Nous montrerons comment les règles de raffinement diminuent les étapes de réécriture en cas de vérification par simulation, par analyse dynamique ou par Model Checking.

La réduction proposée au niveau de ce chapitre et dans [Bou06c], [Bou06d] est spécifique aux Ada-ECATNets. Par conséquent, l'Ada-ECATNet réduit obtenu peut être soumis à une autre réduction comme celle proposée dans le chapitre 3 de la deuxième partie. Cette double réduction permet une diminution significative de la complexité de l'analyse de l'espace état. Avec précision et dans cette direction, nous montrerons dans la suite comment nous appliquons sur un Ada-ECATNet, la règle de réduction 'places parallèles' adaptée aux ECATNets. L'Ada-ECATNet réduit obtenu peut être soumis à tous les outils disponibles d'analyse des ECATNets. Dans ce chapitre, nous proposons quelques règles de raffinement pour traduire des Ada-instructions à un ECATNet. De telle manière, nous représentons de manière compacte beaucoup d'instructions dans une transition.

Le reste de ce chapitre est organisé comme suit : La section 2 présente les principales règles de réduction que nous avons défini pour les Ada-ECATNets. Dans la section 3, nous montrons comment nous appliquons sur l'exemple du producteur consommateur nos règles de réduction proposées ainsi que la règle de réduction les 'places parallèles'. Finalement, des remarques tirées du travail sont présentées dans la section 4.

2. Règles de Réduction pour les Ada-ECATNets

Nous présentons dans cette section quelques règles de raffinement menant à une réduction efficace de la taille de d'Ada-ECATNet obtenu.

Règle 1. Concernant une séquence d'instructions d'affectation. D'abord, nous étudions le cas de deux instructions. Ensuite, nous généralisons la règle à plusieurs instructions. Pour deux instructions d'affectation, $x:=e_1$, $y:=e_2$ représentées par un ECATNet dans la figure 1 (a), nous pouvons obtenir un ECATNet avec une seule transition au lieu de deux, en remplaçant l'occurrence de x par e_1 dans l'expression e_2 dans l'instruction $y:=e_2$. Nous mettons $y:= e_2[x/e_2]$. Nous avons dans ce cas deux instructions $x := e_1$ et $y:= e_3$. Ces deux sont représentées avec un ECATNet dans la figure 1 (b). En général, soit I_1 une séquence d'instructions $x_1:= e_1$, $x_2:= e_2$, …, $x_n:=e_n$, alors nous procédons comme suit : dans $x_2:=e_2$, nous remplaçons chaque occurrence de x_1 dans e_2 par e_1. Nous obtenons une nouvelle séquence d'instructions I_2. Récursivement, nous prenons l'instruction $x_i:=e_i$ et nous remplaçons dans e_i chaque occurrence de x_i,..., x_{n-1} par les côtés droits des affectations appropriées définies dans la séquence I_{i-1}. Nous obtenons dans ce cas une nouvelle séquence d'instructions I_i. L'opération se termine après avoir obtenu la dernière séquence d'instructions d'affectation I_n. C'est la séquence qui sera représentée par une transition dans les ECATNets.

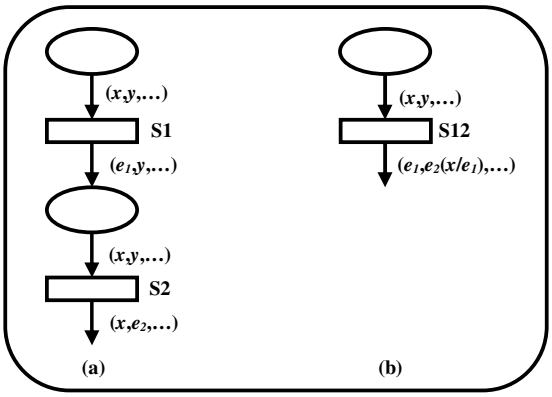

Figure 1. Ada-ECATNet avant (a) et après (b) l'application de la règle 1

Règle 2. Concernant une séquence d'affectations avec une instruction (if-then-else). Pour une instruction d'affectation $x:= e_1$ suivie par une instruction conditionnelle *if cond then* …, nous avons l'ECATNet correspondant dans la figure 2 (a). Chaque occurrence de x dans *cond* est remplacée par e_1 et nous obtenons l'ECATNet dans la figure 2 (b). Dans le cas général, *i.e.* une séquence des instructions d'affectation avec une instruction conditionnelle, nous transformons d'abord cette séquence dans la même manière décrite dans la règle 1. Puis, nous remplaçons chaque occurrence d'une variable dans *cond* par le côté droit des affectations appropriées.

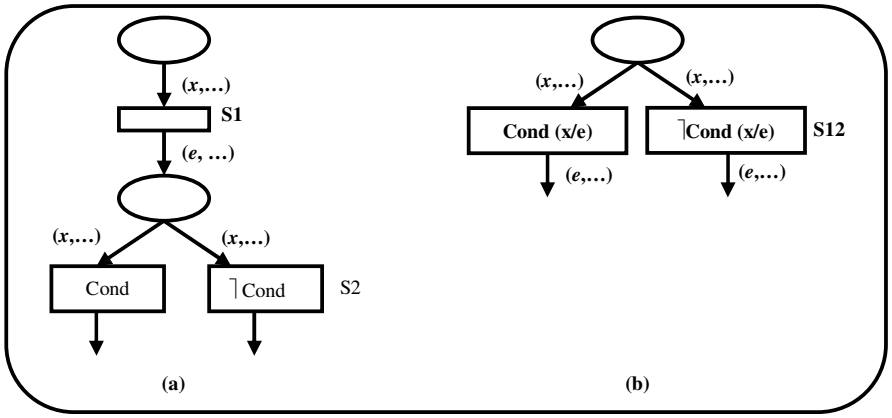

Figure 2. Ada-ECATNet avant (a) et après (b) l'application de la règle 2

Règle 3. Concernant une séquence d'instructions d'affectation suivie d'un appel à une procédure (respectivement, l'appel d'une fonction). La figure 3 (a) représente un ECATNet avec deux transitions : la première représente une instruction d'affectation et l'autre représente l'appel à une procédure. Nous pouvons les intégrer dans une seule transition comme il est décrit dans la figure 3 (b). Une telle intégration est obtenue par le raffinement de séquence d'affectation en utilisant la règle 1. Pour traiter les paramètres, nous ne mettons pas comme paramètres, les noms de variables, mais leurs équivalents côtés droits des instructions d'affectation trouvées dans la séquence raffinée.

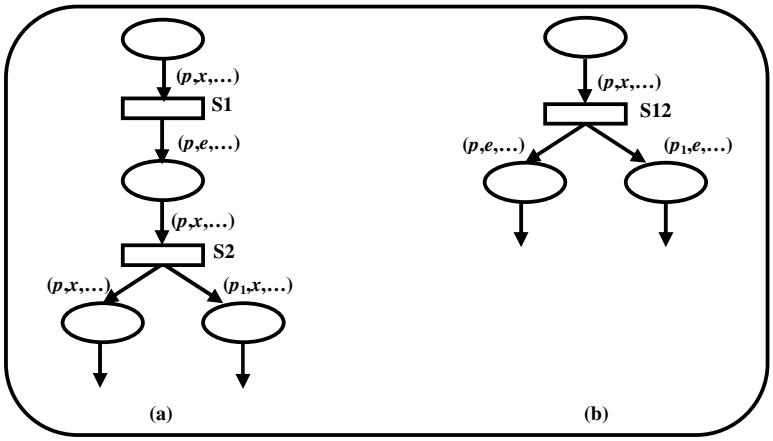

Figure 3. Ada-ECATNet avant (a) et après (b) l'application de la règle 3

Règle 4. Concernant le retour d'une procédure (respectivement, le retour d'une fonction) suivi d'une séquence d'instructions d'affectation. Un retour d'un appel de procédure contient implicitement une affectation. Cette situation peut être traitée d'une façon semblable décrite dans la règle 3.

3. Application des Règles de Réduction sur l'Exemple

Dans cette section, nous décrivons comment nous appliquons les règles de raffinement sur l'exemple de producteur consommateur. Après cela, nous montrons l'application de la règle de réduction 'places parallèles' adaptée aux ECATNets sur l'Ada-ECATNet obtenu.

3.1. Application des Règles de Réduction Ada-ECATNet sur l'Exemple

L'application des règles définies ci-dessus sur l'entrée Write du type Buffer donne la représentation compacte de la figure 4. Nous avons fusionné cinq transitions dans une seule transition. Nous avons fait la même chose avec l'entrée Read. Dans le programme Maude correspondant, nous avons maintenu les règles WriteTaskFilter et WriteTaskSelect sans aucun changement. Les cinq règles restantes sont fusionnées dans une seule transition comme suit :

crl [WriteS123EntryReturn] : < Buf ; (BF , P , CT , II , OI) > . < AcWrite ; (Pr ,, Ch) > . < WaitAckEWrite ; (Pr ,, Ch) >
=> < Buf ; (BF , set(P, Ch, II) , (CT + 1), ((II rem lengtha(P)) + 1) , OI) > . < BeginS2Pr ; (Pr , Ch) >
if CT < lengtha(P) .

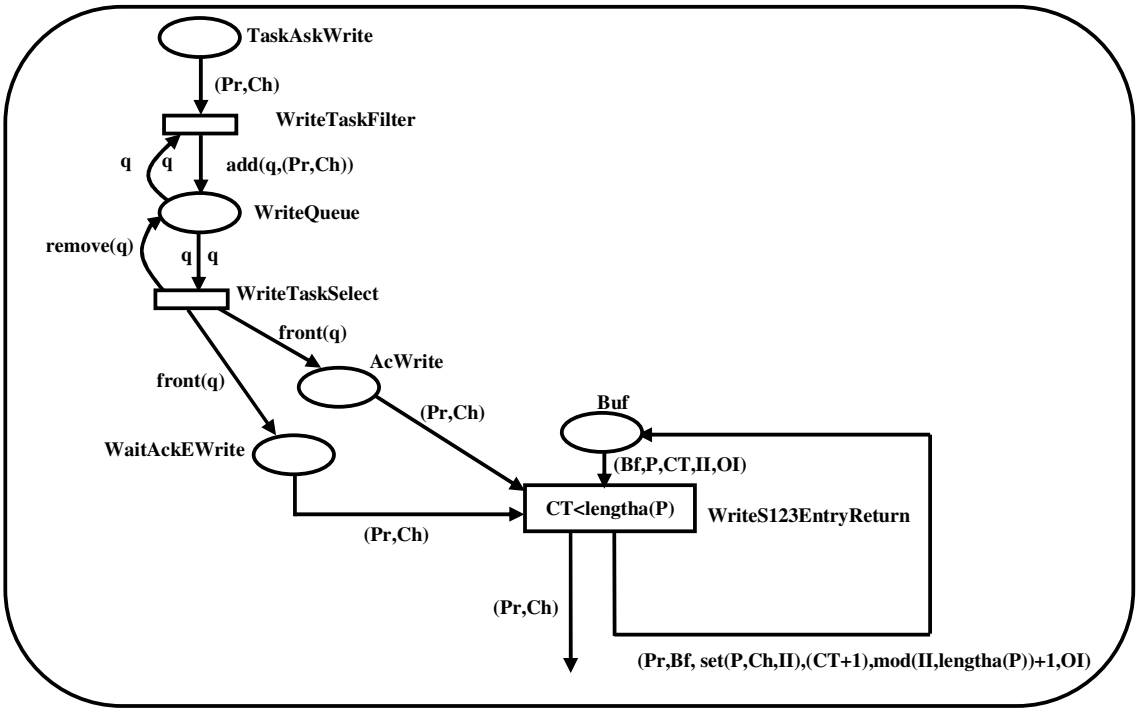

Figure 4. Représentation compacte de l'entrée Write du type Buffer après application des règles de raffinement

3.2. Application des Règles de Réduction des RPAs Adaptées aux ECATNets sur l'Exemple

Après l'application des règles de réduction proposées dans ce chapitre sur l'exemple précédent, nous notons que l'ECATNet donné dans la figure 4 peut être encore réduit. Nous pouvons appliquer la règle de réduction 'places parallèles' adaptée aux ECATNets. Nous notons que WaitAckEWrite et AcWrite sont des places parallèles. Elles vérifient la condition d'application de cette règle. Ces deux places sont vides dans l'état initial. Nous pouvons éliminer l'une d'entre eux. Nous supprimons la place WaitAckEWrite. Nous verrons comment cette double réduction diminue les étapes de quelques méthodes d'analyse dans le chapitre suivant.

4. Conclusion

Les ECATNets offre une représentation compacte des programmes Ada. Dans ce chapitre, nous avons proposé comment obtenir une représentation plus compacte d'Ada-ECATNet des programmes d'Ada pendant l'étape de translation. L'obtention de telle réduction consiste à intégrer plusieurs transitions (qui représentent quelques instructions séquentielles d'Ada) dans une seule transition. D'une part, une telle opération réduit la taille des ECATNets, et d'autre part, cette opération minimise considérablement le nombre des règles de réécriture dans le programme Maude équivalent. L'Ada-ECATNet réduit aura une petite taille en termes de transitions, de places et d'arcs et ainsi de petit nombre d'états par rapport à son équivalent avant la réduction. Par conséquent, l'application de tout outil de vérification devient plus efficace. Nous avons montré à travers un exemple comment il est possible d'appliquer une règle de réduction des RPAs sur l'Ada-ECATNet après l'application des règles de raffinement proposées dans ce chapitre. Cette double réduction diminue efficacement le temps d'exécution et l'espace mémoire de quelques méthodes d'analyse. Le chapitre suivant contient les tests pratiques confirmant de tels déductions. Nous faisons appel au simulateur, à l'analyseur dynamique et le Model Checker pour comparer le nombre des étapes de réécriture avant et après les réductions sur l'Ada-ECATNet.

Chapitre 3 :
Analyse des Ada-ECATNets

1. Introduction

Les techniques de vérification connues des différentes catégories des réseaux de Petri se résument dans la simulation, l'analyse statique, l'analyse dynamique et le Model Checking. En plus, certaines catégories de réseaux de Petri présentent des règles de réduction pour rendre ces méthodes d'analyse plus efficaces. De même, les ECATNets présentent actuellement ces différentes techniques. L'intégration des ECATNets dans la logique de réécriture donne de nombreuses occasions de validation et de vérification de point de vue théorique et pratique. Dans ce chapitre et à travers un exemple, nous montrons la possibilité de simuler un programme Ada à l'aide des ECATNets. Nous faisons appel aux outils offerts par Maude pour l'analyse d'accessibilité. Ensuite, nous vérifions des propriétés sémantiques à l'aide du Model Checker de Maude. Nous montrons aussi dans ce chapitre l'impact de l'utilisation des règles de réduction proposées dans le cadre de cette thèse.

Le reste de ce chapitre est organisé comme suit : Nous présentons dans la section 2 à travers un exemple, notre démarche à propos de la simulation d'un programme Ada à l'aide des ECATNets. Dans la section 2, nous découvrons ce que les ECATNets offrent aux programmes Ada comme outils en ce qui concerne l'analyse de l'accessibilité. Dans la section 4, nous expliquons comment appliquer le Model Checker de Maude sur l'exemple du programme Ada-ECATNet modélisant le problème de producteur consommateur. Nous montrons dans la section 5, les gains obtenus en appliquant la représentation compacte d'Ada-ECATNet et les règles de réduction adaptées aux ECATNets. La section 6 est une conclusion de ce chapitre.

2. Simulation

La logique de réécriture offre une grande flexibilité en ce qui concerne la simulation. Sans la moindre modification dans le programme d'origine, nous pouvons valider des parties de code séparément à d'autres parties. Ceci est possible grâce au choix libre du marquage initial par l'utilisateur. Par exemple, si nous voulons observer uniquement le comportement des tâches productrices indépendamment de comportement de consommateurs, nous pouvons introduire le marquage suivant :

< InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; (Pr, AnyThing) > . < WriteQueue ; newq >
. < Buf ; BF,newa,0,1,1 > .

Nous appliquons la commande 'rew' de système Maude sur ce marquage :

rew in BUFFER : < InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; (Pr, AnyThing) > . < WriteQueue ; newq >
. < Buf ; BF,newa,0,1,1 > .

Ce qui donne le résultat suivant :

result Marking: < InitialCharList ; empty > . < WriteQueue ; newq > . < AcWrite ; Pr,,'y >

. < WaitAckEWrite ; Pr,,'y >
. < Buf ; BF,store(store(store(store(store(newa, 't, 5), 'x, 4), 'b, 3), 'v, 2), 'r, 1),5,1,1 >

Le producteur est en état d'attente dans la file de l'entrée Write parce que le Buffer et rempli (5 caractères au maximum). Si nous reprenons un autre exemple avec une liste de départ contenant un nombre de caractères moins que la taille maximale du Buffer :
rew in BUFFER :
rew in BUFFER : < InitialCharList ; 'r 'v 'b > . < BeginPr ; Pr,AnyThing > . < WriteQueue ; newq >
. < Buf ; BF,newa,0,1,1 > .

Dans ce cas, le producteur arrive à sa fin comme le montre le résultat de la récriture de ce marquage :
result Marking: < EndPr ; Pr,endoflist > . < InitialCharList ; empty > . < WriteQueue ; newq >
. < Buf ; BF,store(store(store(store(newa, endoflist, 4), 'b, 3), 'v, 2), 'r, 1),4,5,1 >

Nous pouvons aussi contrôler pas à pas la simulation de la réécriture. Nous pouvons voir l'état du système après un ou plusieurs pas de réécriture. Par exemple, nous demandons au système Maude de nous donner l'état intermédiaire du système après 10 pas de réécriture :
rew [10] in BUFFER : < InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; Pr,AnyThing > . < WriteQueue ; newq >
. < Buf ; BF,newa,0,1,1 > .

Le résultat de telle exécution retourne le marquage intermédiaire suivant :
result Marking: < BeginS1Pr ; Pr,'r > . < InitialCharList ; 'v 'b 'x 't 'y > . < WriteQueue ; newq >
. < Buf ; BF,store(newa, 'r, 1),1,2,1 >

Nous pouvons continuer pas à pas notre exécution en utilisant la commande 'continue' si nous voulons contrôler de près les états intermédiaires. Par exemple, la commande 'continue 3 . ' retourne l'état du système après trois étapes de réécriture.
result Marking: < InitialCharList ; 'b 'x 't 'y > . < WriteQueue ; newq > . < AcWrite ; Pr,,'v >
. < WaitAckEWrite ; Pr,,'v > . < Buf ; BF,store(newa, 'r, 1),1,2,1 >

Maintenant, nous procédons à la simulation de tout le programme, voici un marquage initial :
rew in BUFFER : < InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; Pr, AnyThing > . < WriteQueue ; newq >
. < BeginCs ; Cs,AnyThing > . < ReadQueue ; newq > . < CharListResult ; empty > . < Buf ; BF,newa,0,1,1 > .

Le résultat de telle simulation est le marquage :
result Marking: < EndPr ; Pr,endoflist > . < InitialCharList ; empty > . < EndCs ; Cs,endoflist >
. < CharListResult ; 'r 'v 'b 'x 't 'y > . < ReadQueue ; newq > . < WriteQueue ; newq >
. < Buf ; BF,store(store(store(store(newa, 't, 5), 'x, 4), 'b, 3), endoflist, 2), 'y, 1),0,3,3 >

Note. Par la suite, nous allons donner des noms pour les états initiaux. Le nom initial6 (une constante) sera un état initial avec InitialCharList contenant 6 caractères. Nous considérons :
eq initial6 = < InitialCharList ; 'r 'v 'b 'x 't 'y > . < BeginPr ; (Pr , AnyThing) >

. < WriteQueue ; newq > . < BeginCs ; (Cs , AnyThing) > . < ReadQueue ; newq >
. < CharListResult ; empty > . < Buf ; (BF , newa , 0 , 1 , 1) > .

3. Analyse d'Accessibilité

D'autres styles d'analyse sont possibles grâce aux nombreux services offerts par Maude. Grâce à la commande 'search', nous pouvons savoir si un certain état est accessible ou non à partir d'un état initial. En général, nous connaissons à l'avance l'état final auquel le système doit aboutir à partir d'un état initial. Nous voulons savoir si le système aboutit à cet état final ou non. Dans les systèmes non-déterministes, une exécution par simulation ne peut pas montrer si le système aboutit ou non à un état final spécifique à cause de la non-détermination de comportement qu'on ne peut pas forcer à suivre un chemin déterminé. Pour savoir s'il y'a un chemin possible dans l'exécution du système et qui lui permet d'aboutir à cet état final, nous faisons appel à la commande 'search'. Dans notre exemple, nous voulons être sûrs que l'état final précédent est abouti par l'état initial initial6. La commande suivante permet de lancer la recherche de tous les états accessibles de l'état initial jusqu'à l'arrivée à l'état final. Dans l'absence d'une solution, le système affiche 'no solution' :

search in BUFFER : initial6 =>*
< EndPr ; Pr,endoflist > . < InitialCharList ; empty > . < EndCs ; Cs,endoflist >
. < CharListResult ; 'r 'v 'b 'x 't 'y > . < ReadQueue ; newq > . < WriteQueue ; newq >
. < Buf ; BF,store(store(store(store(store(newa, 't, 5), 'x, 4), 'b, 3), endoflist, 2), 'y, 1),0,3,3 > .

Cette commande 'search' nous permet d'avoir le graphe d'accessibilité de tout l'ECATNet à partir d'un état initial. Pour cela, il faut préciser d'abord un état final général qui est dans le cas des ECATNets M:Marking. Dans ce cas 'search' retourne tout état accessible de l'état initial parce que tout état d'un ECATNet est une instanciation de l'état M:Marking.

search in BUFFER : initial6 =>* M:Marking .

Ensuite, pour avoir tout le graphe d'accessibilité dans ce cas, il faut écrire après cette requête la formule suivante :
show search graph .

Une partie du graphe d'accessibilité de cet exemple se trouve dans la figure 1. Nous remarquons dans cette figure que l'état initial est dénoté par 0. L'exécution précise qu'à partir de cet état on aboutit à l'état 1 (arc 0 ===>), le système Maude précise aussi la règle de réécriture appliquée pour passer de l'état 0 à l'état 1. L'obtention de graphe d'accessibilité permet de décider aisément les propriétés statiques comme la vivacité et la présence de deadlock.

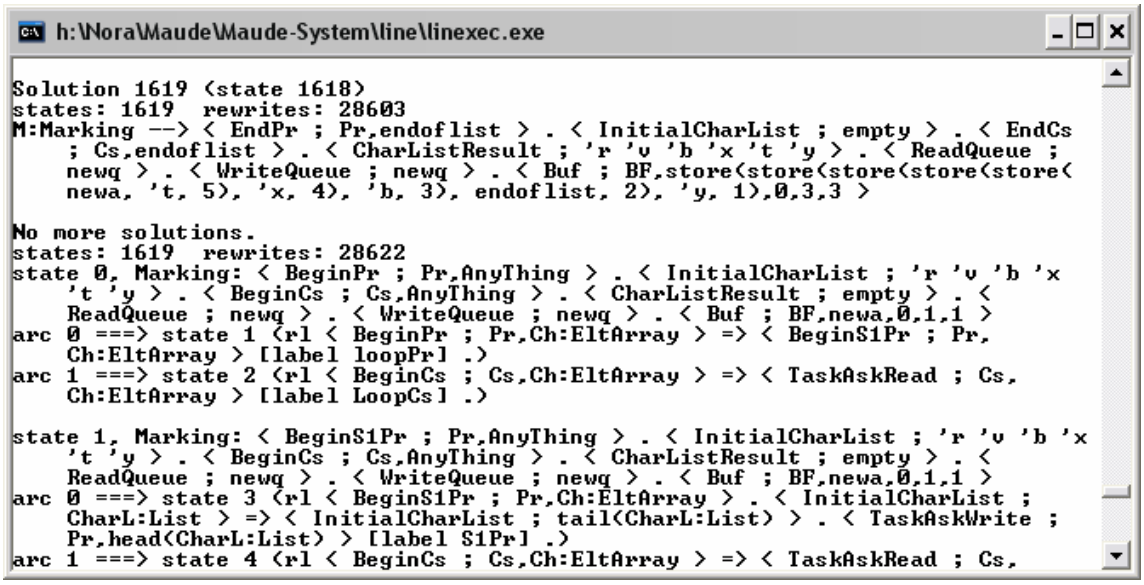

Figure 1. Exemple de graphe d'accessibilité dans Maude

4. Model Checking

Dans cette section, nous étudions l'applicabilité de Model Checking de Maude pour la vérification de quelques exemples de propriétés de concurrence d'Ada-ECATNet. Six propriétés sont définies et prouvées qu'elles sont correctes en utilisant le Model Checker de Maude.

Propriété 1. Selon la spécification, le producteur Pr n'accède qu'à l'entrée Write et jamais à l'entrée Read. Nous voulons développer une formule pour prouver que cette propriété est toujours vraie. D'abord, nous définissons les propositions Pr-In-TaskAskRead(Pr), Pr-In-AcRead(Pr), Pr-In-WaitAckERead(Pr), Pr-In-ReadQueue(Pr) Pr-In-BeginRead(Pr), Pr-In-BeginS2Read(Pr), Pr-In-BeginS3Read(Pr) et Pr-In-EndRead(Pr). La première proposition est vraie si Pr est dans la place TaskAskRead. La seconde est vraie si Pr est dans la place AcRead et ainsi de suite pour les autres propositions. Nous expliquons comment réaliser Pr-In-TaskAskRead(Pr) et Pr-In-ReadQueue(Pr) dans Maude. Pr-In-TaskAskRead(Pr) est valide dans la configuration < TaskAskRead ; (Pr , Ch) > . M, tel que M est une variable de type 'Marking':

op Pr-In-TaskAskRead : Producer -> Prop .

eq < TaskAskRead ; (Pr , Ch) > . M |= Pr-In-TaskAskRead(Pr) = true .

Pr-In-ReadQueue(Pr) est valide dans la configuration de la forme < ReadQueue ; q > . M, de telle façon que Pr se trouve dans la file d'attente q. Cette dernière contient des pairs de la forme (Pr', Ch). La fonction find-1st-Queue(Pr, q) retourne vrai si Pr est le premier élément dans un pair qui se trouve dans la file q :

op Pr-In-ReadQueue : Producer -> Prop .

ceq < ReadQueue ; q > . M |= Pr-In-ReadQueue(Pr) = true if find-1st-Queue(Pr, q) == true .

Nous montrons en utilisant le Model Checker de Maude la formule suivante qui exprime notre propriété :

red in BUFFER-CHECK : modelCheck(initial6,

[] ~(Pr-In-TaskAskRead(Pr)) /\ [] ~(Pr-In-AcRead(Pr)) /\ [] ~(Pr-In-WaitAckERead(Pr))

/\ [] ~(Pr-In-ReadQueue(Pr)) /\ [] ~(Pr-In-BeginRead(Pr)) /\ [] ~(Pr-In-BeginS2Read(Pr))

/\ [] ~(Pr-In-BeginS3Read(Pr)) /\ [] ~(Pr-In-EndRead(Pr))) .

La formule [] ~(Pr-In-TaskAskRead(Pr)) signifie pour le producteur Pr n'est jamais dans la place TaskAskRead dans n'importe quel état accessible à partir du marquage initial 'initial6'.

Propriété 2. De même pour un consommateur Cs qui accède uniquement à l'entrée Read mais jamais à l'entrée Write. Les propositions Cs-In-TaskAskWrite(Cs), Cs-In-AcWrite(Cs), Cs-In-WaitAckEWrite(Cs), Cs-In-WriteQueue(Cs), Cs-In-BeginWrite(Cs), Cs-In-BeginS2Write(Cs), Cs-In-BeginS3Write(Cs) et Cs-In-EndWrite(Cs) sont définies pour de tel besoin. La formule suivante exprimant notre propriété est montrée valide en utilisant le Model Checking de Maude :

```
red in BUFFER-CHECK : modelCheck(initial6,
[] ~(Cs-In-TaskAskWrite(Cs)) /\ [] ~(Cs-In-AcWrite(Cs)) /\ [] ~(Cs-In-WaitAckEWrite(Cs))
/\ [] ~(Cs-In-WriteQueue(Cs)) /\ [] ~(Cs-In-BeginWrite(Cs)) /\ [] ~(Cs-In-BeginS2Write(Cs))
/\ [] ~(Cs-In-BeginS3Write(Cs)) /\ [] ~(Cs-In-EndWrite(Cs)) ) .
```

Propriété 3. À chaque fois une tâche productrice Pr demande l'entrée Write, elle accède à cette entrée. Une formule qui exprime cette propriété pour le producteur Pr et l'état initial 'initial6' est :

```
red modelCheck(initial6, (<> Pr-In-TaskAskWrite(Pr)) => ( <> Pr-In-WriteEntry(Pr))) .
```

La vérification de cette formule retourne 'true'.

Propriété 4. De même pour la tâche consommatrice, si Cs demande l'entrée Read, elle accède à cette entrée. La formule qui exprime cette propriété est la suivante :

```
red modelCheck(initial6, (<> Cs-In-TaskAskRead(Cs)) => ( <> Cs-In-ReadEntry(Cs))) .
```

L'évaluation de cette propriété retourne aussi 'true'.

Propriété 5. Quand une tâche accède à une entrée du type protégé, aucune autre tâche n'accède à une autre entrée de ce type de protégé. La tâche Pr (resp. Cs) est dans l'entrée Write (resp. Read) si elle est dans l'une des places possibles de cette entrée BeginWrite, BeginS2Write, BeginS3Write et EndWrite et vice-versa. Pour le producteur Pr et le consommateur Cs et l'état initial 'initial6', l'évaluation de cette propriété est vraie :

```
red in BUFFER-CHECK : modelCheck(initial6,
<> ((Pr-In-BeginWrite(Pr) \/ Pr-In-BeginS2Write(Pr) \/ Pr-In-BeginS3Write(Pr) \/ Pr-In-EndWrite(Pr))
=> ~ ( Cs-In-BeginRead(Cs) \/ Cs-In-BeginS2Read(Cs) \/ Cs-In-BeginS3Read(Cs) \/ Cs-In-EndRead(Cs)))
/\ <> ( Cs-In-BeginRead(Cs) \/ Cs-In-BeginS2Read(Cs) \/ Cs-In-BeginS3Read(Cs) \/ Cs-In-EndRead(Cs)
=> ~ ( (Pr-In-BeginWrite(Pr) \/ Pr-In-BeginS2Write(Pr) \/ Pr-In-BeginS3Write(Pr) \/ Pr-In-EndWrite(Pr))))) .
```

Propriété 6. Nous pouvons prouver que le contenu de la liste InitialCharList à l'état initial égal au contenu de CharListResult à un état final. Nous décrivons la proposition InitialCharList-In-InitialState(CharL) qui est valide exactement au marquage initial :

```
op InitialCharList-In-InitialState : List -> Prop .
```

eq < InitialCharList ; CharL > . < BeginPr ; (Pr , AnyThing) >

. < WriteQueue ; newq > . < BeginCs ; (Cs , AnyThing) > . < ReadQueue ; newq >

. < CharListResult ; empty > . < Buf ; (BF , newa , 0 , 1 , 1) >

|= InitialCharList-In-InitialState(CharL) = true .

Dans cette direction, CharListResult-In-FinalState(CharL) est une proposition valide à l'état final. Nous exprimons et nous faisons appel au Model Checker de Maude pour prouver notre propriété :

red in BUFFER-CHECK : modelCheck(initial6,

<> (InitialCharList-In-InitialState('r 'v 'b 'x 't 'y)) /\ <> (CharListResult-In-FinalState('r 'v 'b 'x 't 'y))) .

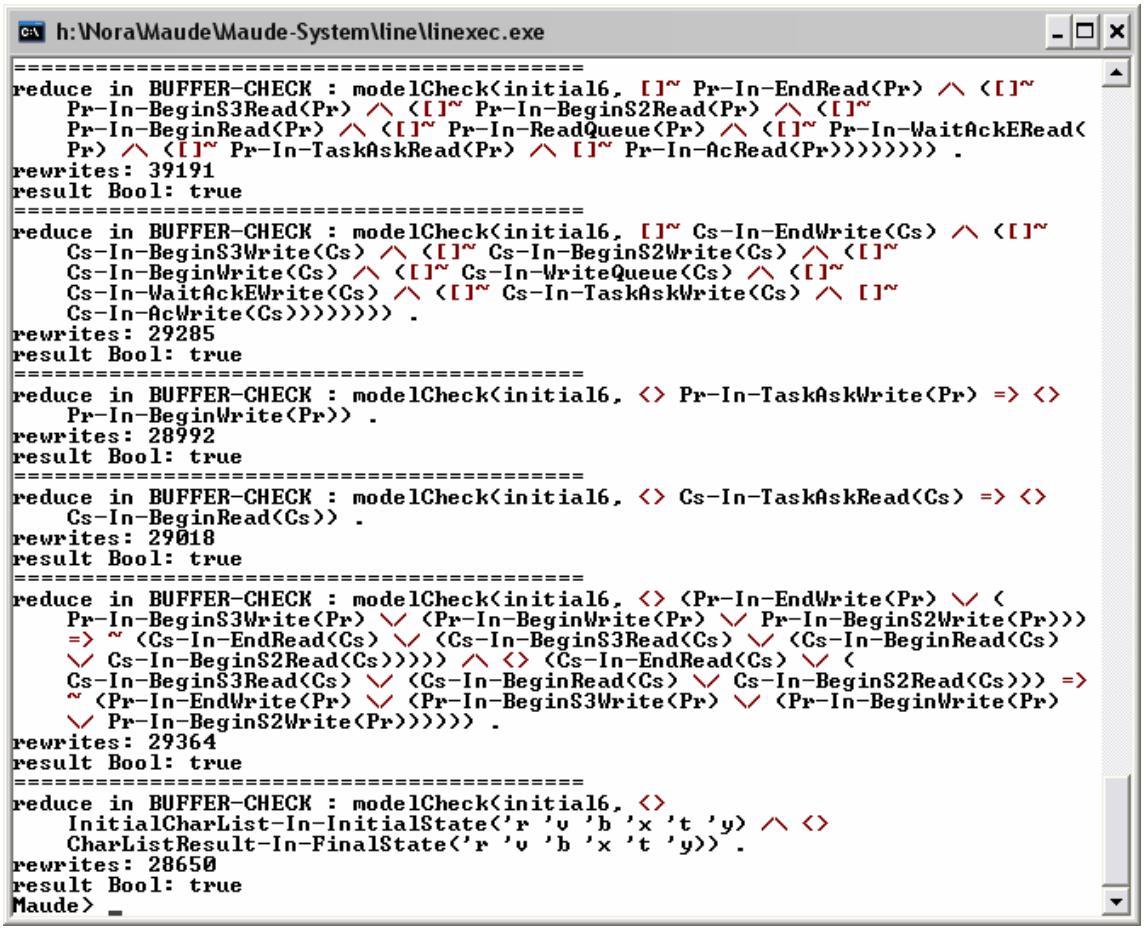

Figure 2. Vérification de toutes les propriétés en utilisant le Model Checker de Maude

5. Application des Règles de Réduction : Evaluation des Performances

Pour montrer comment les règles proposées ont réduit d'une manière efficace la taille d'Ada-ECATNet, nous avons appliqué la simulation, l'analyse dynamique et le Model Checker sous le système de Maude. Dans la suite, nous considérons :

Cas1 : Ada-ECATNet avant l'application de n'importe quelle réduction.

Cas2 : Ada-ECATNet après l'application des règles de réduction proposées dans le chapitre précédent.

Cas3 : Ada-ECATNet après l'application des règles de réduction proposées dans le chapitre précédent et dans le chapitre 3 de la deuxième partie.

Note. L'analyse de l'accessibilité et le Model Checker de Maude ne sont pas décidables en cas des programmes à états-infinis. Cependant, de telles techniques sont intéressantes vu que les Ada-nets obtenus après translation des programmes Ada aux réseaux de Petri sont des systèmes à états-finis [Bar98].

5.1. Simulation

Pour la simulation des trois cas en utilisant le même marquage initial, nous trouvons naturellement le même marquage final. Pour une entrée InitialCharList contenant chaque fois de 6 à 10 caractères, nous avons fait une simulation pour les trois cas Cas1, Cas2 et Cas3. Pour chaque n (n = 6,..,10) le nombre de caractères, nous avons calculé le nombre des étapes de réécriture exigés pour obtenir le marquage final pour le même état initial dans les trois cas. D'abord, nous notons que le nombre des étapes de réécriture dans Cas1 est toujours plus élevé que celui dans Cas2. D'ailleurs, l'espace entre les deux nombres des étapes de réécriture dans Cas1 et Cas2 augmente chaque fois que le nombre de caractères dans InitialCharList augmente. Le résultat de la simulation est présenté dans la table comparative suivante :

	6 C.	**7 C.**	**8 C.**	**9 C.**	**10 C.**
Cas1	1098	1298	1531	1797	1966
Cas2	995	1180	1398	1649	1803
Cas3	994	1179	1397	1648	1802
Diff.	**104**	**119**	**134**	**149**	**164**

Notons que Diff. (Différence) dans le tableau est entre Cas1 et Cas3. La simulation de Cas1, Cas2 et Cas3 qui concernent une donnée d'entrée InitialCharList contenant 10 caractères se trouve dans la figure 3.

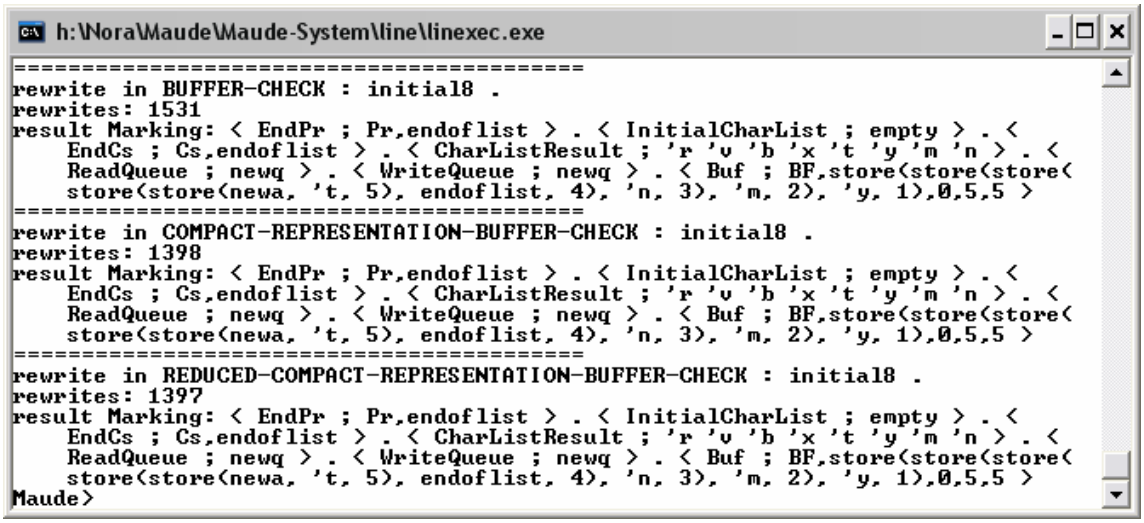

Figure 3. Le marquage résultant de la simulation sur l'état initial avant et après l'application des règles de réduction

5.2. Analyse d'Accessibilité

Nous avons obtenu un résultat intéressant quand nous avons construit le graphe d'accessibilité pour chaque cas. Nous avons considéré deux critères de comparaison : les nombres des états et les nombres des pas de réécriture. La différence est significative. La réduction proposée dans ce chapitre a un grand impact dans la diminution de nombre des pas de réécriture nécessaires pour la construction du graphe de marquage. L'application de la règle de réduction 'places parallèles' diminue également ce nombre. A titre d'exemple, quand InitialCharList contient 10 caractères, la construction du graphe d'accessibilité exige dans Cas3 un nombre d'étapes de réécriture égal approximativement à 33

% moins que dans Cas1. La différence entre Cas1 et Cas2 est très intéressante. La différence entre Cas2 et Cas3 est légère. Comme il est décrit ci-dessus, l'avantage de la règle de réduction 'places parallèles' consiste à diminuer les pas de réécriture dans la construction du graphe de marquage. Cet avantage est réalisé dans notre situation entre Cas2 et Cas3. Le tableau suivant résume l'évolution de nombre des étapes de réécriture demandées dans Cas1, Cas2 et Cas3 pour construire le graphe de marquage. La différence indiquée dans les tableaux est entre Cas1 et Cas3.

Pas de réécriture					
	6 C.	**7 C.**	**8 C.**	**9 C.**	**10 C.**
Cas1	28622	36047	44147	52922	60747
Cas2	19143	24162	29697	35767	40747
Cas3	19130	24130	29665	35735	40715
Diff.	**8292**	**10097**	**11902**	**13707**	**15512**

Le tableau suivant résume l'évolution de nombre des états générés dans Cas1, Cas2 et Cas3 dans le graphe de marquage pour InitialCharList contenant de 6 à 10 caractères. Le nombre des états dans Cas3 est moins de 57 % par rapport au Cas1.

Nombre des états					
	6 C.	**7 C.**	**8 C.**	**9 C.**	**10 C.**
Cas1	1619	1969	2319	2669	3019
Cas2	699	849	999	1149	1299
Cas3	695	845	995	1145	1295
Diff.	**924**	**1124**	**1324**	**1524**	**1724**

5.3. Model Checking

Nous obtenons le même résultat quand nous faisons appel au Model Checker de Maude. Pour InitialCharList contenant chaque fois de 8 à 10 caractères, le tableau suivant résume l'évolution de nombres des étapes de réécriture requises dans Cas1, Cas2 et Cas3 pour vérifier la correction des trois propriétés définies au-dessus.

	Propriété 1.				**Propriété 2.**				**Propriété 3.**		
	8 C.	**9 C.**	**10 C.**		**8 C.**	**9 C.**	**10 C.**		**8 C.**	**9 C.**	**10 C.**
Cas1	58916	69792	79717		45090	53965	61890		44176	52951	60776
Cas2	37226	44317	50317		29960	36030	41010		29726	35796	40776
Cas3	37164	44255	50255		29928	35998	40978		29694	35764	40744
Diff.	**21752**	**25537**	**29462**		**15162**	**17967**	**20912**		**14482**	**17187**	**20032**

Une partie de vérification de la propriété 1 en utilisant le Model Checker de Maude est présentée dans la figure 4. Les règles de réduction proposées dans cette thèse diminue le nombre des pas de réécriture de 58916 dans Cas1 à 37226 dans Cas2. L'application de la règle de réduction 'places parallèles' diminue encore ce nombre à 37164 dans Cas3.

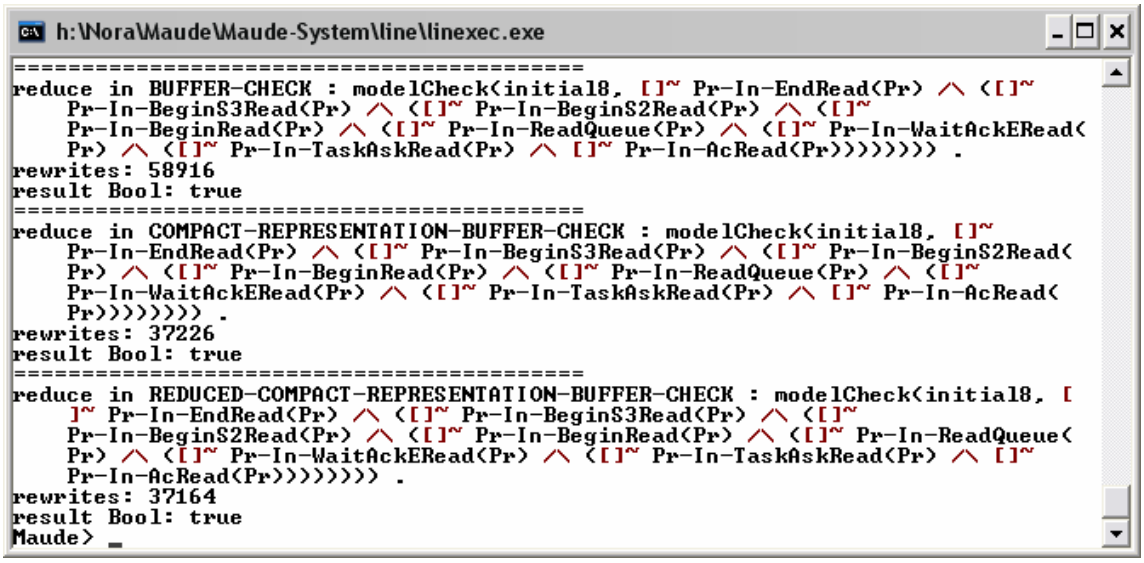

Figure 4. Vérification de propriété en utilisant le Model Checker de Maude avant et après l'application des règles de réduction

6. Conclusion

Dans ce chapitre, nous avons montré comment utiliser les différents outils des ECATNets dans l'analyse d'un programme Ada. Nous avons appliqué le simulateur de Maude sur un exemple. Ce simulateur présente quelques avantages comme la liberté de choisir le marquage initial et le contrôle des états intermédiaires. En second lieu, nous avons utilisé les techniques offertes par Maude pour l'analyse de l'accessibilité. L'analyse de l'accessibilité de Maude ne fonctionne pas en cas des programmes à états-infinis. Cependant, de telles techniques sont intéressantes vu que les Ada-nets obtenus après translation des programmes Ada aux réseaux de Petri sont des systèmes à états-finis. Enfin, nous avons montré l'utilisation de la technique du Model Checking dans la vérification d'un programme Ada. Nous avons développé six propriétés et nous avons montré leur correction en utilisant le Model Checker de Maude.

Les règles de réduction adaptées aux Ada-ECATNets apportent leurs fruits en ce qui concerne l'efficacité des méthodes d'analyse comme la simulation, l'analyse dynamique et le Model Checking. Dans notre proposition, l'Ada-ECATNet compact obtenu peut être réduit encore en appliquant des règles de réduction proposées pour les RPAs et adaptées aux ECATNets. A travers un exemple, nous avons montré que cette double réduction diminue efficacement le temps d'exécution et l'espace mémoire des trois méthodes d'analyse précédemment décrits.

Chapitre 4 :
Réalisation du Translateur Ada-ECATNets

1. Introduction

Dans ce chapitre, nous décrivons l'implémentation de notre translation des programmes Ada dans les ECATNets. Vu la complexité de Ada, notre implémentation touche seulement un sous-ensemble des concepts de ce langage. Nous avons concentré nos efforts sur les concepts relatifs à la concurrence et à la définition de tâche. Notre objectif est de montrer la faisabilité de translater d'une manière pratique un programme Ada vers un ECATNet. Nous considérons dans ce travail la validation de la translation de quelques concepts de base de la concurrence comme, entre autres, le rendez-vous, l'accès à un type protégé, etc. Nous trouvons dans l'annexe, la partie de grammaire du langage Ada pour lequel nous avons généré le code ECATNet.

Le translateur Ada-ECATNet est basé sur l'intégration des trois étapes classiques d'un compilateur : une analyse lexicale, une analyse syntaxique et une étape de génération de code ECATNet. Nous excluons le traitement des erreurs de notre traitement. A cause d'une faiblesse de Maude dans le traitement des fichiers, nous avons décidé de développer l'étape de l'analyse lexicale dans le langage Delphi. Le reste des étapes a été développé en utilisant le langage Maude.

Le reste de ce chapitre est organisé comme suit : la section 2 explique en détail les différentes phases de notre translateur. L'application des différentes étapes du translateur sur un exemple d'un programme Ada est présentée dans la section 3. Finalement, la section 4 est une conclusion de ce chapitre.

2. Translateur Ada-ECATNet

Nous développons le translateur Ada-ECATNet en suivant les mêmes étapes classiques définies en général dans la réalisation des compilateurs. L'ensemble de ces étapes inclut : une analyse lexicale, une analyse syntaxique et une étape de génération de code. Nous excluons de notre application le traitement des erreurs. Le programme Ada à translater ne doit pas contenir des erreurs de compilation ou d'exécution. Notons que si le programme Ada ne contient pas d'erreurs, notre application retourne le code correspondant, sinon notre application retourne 'ErrorUple'. Nous expliquons par la suite les détails de la réalisation des étapes de notre translateur.

2.1. Analyse Lexicale

Cette phase prend un programme Ada comme entrée et génère tous les lexèmes constituant ce programme. Cette phase est implémentée en utilisant le langage impératif Delphi. Cette étape transforme le programme Ada à une liste de ses lexèmes.

2.2. Analyse Syntaxique

Cette étape transforme la liste d'entrée en une représentation intermédiaire pour nous faciliter la génération du code ECATNet. Pour le développement de cette phase, nous avons crée certains types de données spécifiques. Cette phase prend comme entrée un type de données 'Uple' composé d'une liste 'List' et d'une pile 'Stack'. La liste contient

les chaînes de caractères dénotant les lexèmes du programme Ada (opérateur, identificateur, mot clé, etc.). La pile contient les informations sur le programme Ada nécessaires pour la génération du code ECATNet correspondant. A chaque fois que l'analyseur syntaxique évolue dans la liste, il collectionne les informations sur le programme Ada et les met dans la pile. L'élément de base de la pile est un type de donnée appelé 'Code'. Ce dernier contient aussi des chaînes de caractères. L'opération suivante permet la construction d'un 'Uple'.
op (_;_) : List Stack -> Uple .

Dans le cas où la liste à analyser contient une chaîne de caractères fausse (le programme Ada est erroné), l'analyseur retourne 'ErrorUple' :
op ErrorUple : -> Uple .

L'opération 1st extrait le premier paramètre de (_;_), qui est une liste et la fonction 2nd extrait le deuxième paramètre qui est de sorte 'Stack' :
op 1st : Uple -> List . eq 1st((L ; SK)) = L . op 2nd : Uple -> Code . eq 2nd((L ; SK)) = SK .

Maintenant, les opérations pour la construction d'un code (un élément de sorte 'Code') et la construction d'un code vide : op _,_ : Code Code -> Code [assoc id : nullCode] . op nullCode : -> Code .

La fonction 1stC((Id , Cd)) retourne le premier élément dans un code. Tel que Id est de type 'String' et Cd est de type 'Code' : op 1stC : Code -> String . eq 1stC((Id , Cd)) = Id .

D'autres fonctions comme, entre autres, 2ndC, 3rdC, 4rtC sont définies pour retourner dans l'ordre le deuxième élément, le troisième élément et le quatrième élément dans un code.

Nous définissons deux constantes BeginCode et EndCode de type code. Ces deux constantes servent à délimiter dans une pile, les informations nécessaires concernant une construction syntaxique spécifique d'un programme Ada : op BeginCode : -> Code . op EndCode : -> Code .

D'autres types de données sont définis pour Ada-ECATNet et qui sont importés par les modules ECATNets crées pour traduire le programme Ada. Nous considérons le type 'Tuple' qui supporte le jeton constitué par l'identificateur de la tâche et ses variables. L'opération _,_ permet de construite un n-uplet extensible :
sorts Elt Tuple . subsort Elt < Tuple . op _,_ : Elt Tuple -> Tuple .

Nous définissons encore une autre sorte 'ExtTuple' qui est composée de plusieurs 'Tuple'. L'opération _;_ permet de construire la sorte 'ExtTuple'. Ce dernier est utile dans les cas de passage des paramètres au moment de l'appel d'une entrée ou de procédure. Selon notre translation Ada-ECATNet, quand la tâche appelle une entrée, le n-uplet représentant l'état de la tâche entre dans la place où il se trouve la file d'attente de l'entrée. Dans ce cas, nous n'avons pas un mécanisme de distinguer les paramètres actuels utilisés dans l'appel. Alors, pour ne pas perdre de telle information au moment de l'appel, le n-uplet se réécrit en un terme de sorte 'ExTuple'. Ce terme est composé de deux n-uplets : celui qui représente l'état de la tâche et l'autre contient les paramètres actuels à passer plus tard au moment de la concrétisation du rendez-vous entre les deux tâches :

sort ExtTuple . subsort Tuple < ExtTuple . op _;_ : Tuple ExtTuple -> ExtTuple .

Nous expliquons maintenant quelques fonctions d'analyse syntaxique. Nous prenons les fonctions concernant l'analyse des instructions. D'abord, la règle de production de l'instruction d'affectation est :
assignment_statement ::= variable_name **:=** expression;

Intuitivement, après l'analyse de cette instruction, nous devons sauvegarder le nom de variable et le contenu de l'expression. Soit assignment-statement-Analysis(L ; (Id, Ex)), la fonction qui analyse l'affectation et sauvegarde les parties qui composent une affectation. Id est la partie gauche de l'affectation et Ex est la partie droite de l'affectation :

```
op assignment-statement-Analysis : Uple -> Uple .
eq assignment-statement-Analysis(L ; (Id, Ex)) =
if IsIdentifier(head(L)) == true
  and head(tail(L)) == ":="
  and expression-Analysis(tail(tail(L)) ; Ex) =/= ErrorUple
  and head(1st(expression-Analysis(tail(tail(L)) ; Ex))) == ";"
then tail(1st(expression-Analysis(tail(tail(L)) ; Ex))) ; (head(L), 2nd(expression-Analysis(tail(tail(L)) ; Ex)))
else ErrorUple fi .
```

Dans ce code, nous trouvons quatre conditions :

- IsIdentifier(head(L)) == true : cette condition est vraie si head(L) est un identificateur.
- head(tail(L)) == ":=" : cette condition est vraie si l'élément qui se trouve après cet identificateur est égal à ":=".
- expression-Analysis(tail(tail(L)) ; Ex) : cette condition est correcte si les éléments qui se trouvent dans la liste après l'identificateur et ":=" constitue une expression correcte.
- head(1st(expression-Analysis(tail(tail(L)) ; Ex))) == ";" : cette dernière condition est vraie si l'élément qui se trouve dans la liste après les éléments constituant l'expression est égal à ";".

Si les quatre conditions sont valides, la fonction assignment-statement-Analysis(L ; (Id, Ex)) retourne un 'Uple' tail(1st(expression-Analysis(tail(tail(L)) ; Ex))) ; (head(L), 2nd(expression-Analysis(tail(tail(L)) ; Ex))). Ce dernier est constitué du reste de la liste à analyser : tail(1st(expression-Analysis(tail(tail(L)) ; Ex))) après avoir éliminer les éléments décrits au-dessus concernant l'affectation, et un code (head(L), 2nd(expression-Analysis(tail(tail(L)) ; Ex))) contenant les informations nécessaires pour la génération ultérieure du code ECATNet pour cette affectation. Ce code contient la partie droite de l'affectation head(L) et sa partie gauche 2nd(expression-Analysis(tail(tail(L)) ; Ex))). Cette partie, elle-même est retournée par la fonction expression-Analysis(tail(tail(L)) ; Ex)) qui est responsable de l'analyse des expressions.

La fonction assignment-statement-Analysis(L ; (Id, Ex)) est appelée au cours de l'analyse par la fonction simple-statement-Analysis(L ; (Kind, Id, IdL, Ex)) qui analyse le code Ada généré par simple_statement. Nous avons besoin de quatre paramètres pour le code : Kind, Id, IdL et Ex. D'abord, les règles de production de simple_statement sont :

```
simple_statement ::= null_statement | assignment_statement  | exit_statement  | return_statement
                   | entry_call_statement  | abort_statement
null_statement ::= null;
```

exit_statement ::= **exit** [loop_name] [**when** condition]**;**

return_statement ::= **return** [expression]**;**

entry_call_statement ::= entry_name [actual_parameter_part]**;**

abort_statement ::= **abort** task_name {**,** task_name}**;**

Aucune information nécessaire à sauvegarder après l'analyse de l'instruction null-statement. L'analyse de exit_statement retourne comme code le nom de la boucle Id et la condition Ex. Soit exit-statement-Analysis(L ; (Id, Ex)) la fonction qui analyse cette instruction. S'il n' y a pas de nom de boucle parce que c'est optionnel, Id retourné est une chaîne vide. Idem pour la condition. Dans l'analyse de return_statement, l'expression retournée peut être vide aussi. La fonction responsable de cette analyse est dite return-statement-Analysis(L ; (Ex)). L'analyse de return_statement renvoie un code composé de nom de l'entrée Id et une liste IdL des paramètres actuels. S'il n' y a pas de paramètres actuels, la liste IdL est vide. Soit entry-call-statement-Analysis(L ; (Id, IdL)) la fonction qui analyse cette instruction. Enfin, l'analyse de abort_statement retourne un code composé des noms des tâches abandonnées IdL. La fonction qui analyse de telle instruction est dite abort-statement-Analysis(L ; IdL). Nous avons justifié pour l'instant les trois paramètres Id, IdL et Ex. le dernier paramètre Kind sert à enregistrer le type de l'instruction, le code suivant constitue une partie de la fonction simple-statement-Analysis(L ; (Kind, Id, IdL, Ex)) :

```
op simple-statement-Analysis : Uple -> Uple .
eq simple-statement-Analysis(L ; (Kind, Id, IdL, Ex)) =
if IsIdentifier(head(L)) == true
then if assignment-statement-Analysis(L ; (Id, Ex)) =/= ErrorUple
       then 1st(assignment-statement-Analysis(L ; (Id, Ex))) ;
            ("assg", 1stC(2nd(assignment-statement-Analysis(L ; (Id, Ex)))),
            empty, 2ndC(2nd(assignment-statement-Analysis(L ; (Id, Ex)))))
     else if entry-call-statement-Analysis(L ; (Id, IdL)) =/= ErrorUple
            then 1st(entry-call-statement-Analysis(L ; (Id, IdL))) ;
                 ("entry call", 2nd(entry-call-statement-Analysis(L ; (Id, IdL))), "")
          else ErrorUple  fi
     fi
else …
**** L'analyse  des autres types d'instructions   fi .
```

Dans ce code, si head(L) est un identificateur, alors nous testons si assignment-statement-Analysis(L ; (Id, Ex)) =/= ErrorUple est valide. Si cette condition est vraie, nous sommes donc devant une instruction d'affectation dans le code. Le résultat retourné dans ce cas par la fonction est composé par le reste de la liste d'entrée après skipper les éléments constituants l'affectation soit : 1st(assignment-statement-Analysis(L ; (Id, Ex))). Le code est constitué par quatre éléments : "assg", 1stC(2nd(assignment-statement-Analysis(L ; (Id, Ex)))), empty, et 2ndC(2nd(assignment-statement-Analysis(L ; (Id, Ex))))). Le premier élément dans ce code qui est la valeur de Kind indique tout simplement que les trois autres éléments expriment les composants d'une affectation. Le deuxième élément est la partie gauche de l'affectation, le troisième est vide (il est indépendant de l'affectation, mais il n'est pas vide pour d'autres instructions). Le quatrième élément est la partie droite de l'affectation.

La fonction simple-statement-Analysis(L ; (Kind, Id, IdL, Ex)) elle-même est appelée par la fonction statement-Analysis(L ; SK) qui analyse l'instruction. Les règles de production de l'instruction sont :
statement ::= simple_statement | compound_statement

La fonction statement-Analysis(L ; SK) analyse simple_statement et compound_statement. Dans le cas de simple_statement, la fonction crée une pile délimitée par BeginCode et EndCode vu que simple-statement-Analysis(L ; (Kind, Id, IdL, Ex)) renvoie un code simple (une pile avec un seul élément) pour l'isoler des codes des autres instructions. Ceci permet d'enregistrer clairement les informations pour les bien lire par la suite pendant la génération de code ECATNet. Cette fonction n'ajoute pas BeginCode et EndCode dans le cas de compound_statement parce que son analyse par la fonction compound-statement-Analysis(L ; SK) génère déjà un code entre BeginCode et EndCode vu la complexité du code des instructions composées :

```
op statement-Analysis : Uple -> Uple .
eq statement-Analysis(L ; SK) =
if  IsIdentifier(head(L)) == true
then if simple-statement-Analysis(L ; ("", "", empty, "")) =/= ErrorUple
       then 1st(simple-statement-Analysis(L ; ("", "", empty, ""))) ;
            push(BeginCode, push(2nd(simple-statement-Analysis(L ; ("", "", empty, ""))), EndCode))
        else if compound-statement-Analysis(L ; SK) =/= ErrorUple
            then compound-statement-Analysis(L ; SK)
            else ErrorUple     fi
       fi
else if FindInFirsts(head(L), simple-statement-Firsts) == true
        and simple-statement-Analysis(L ; ("", "", empty, "")) =/= ErrorUple
      then 1st(simple-statement-Analysis(L ; ("", "", empty, ""))) ;
           push(BeginCode, push(2nd(simple-statement-Analysis(L ; ("", "", empty, ""))), EndCode))
      else if FindInFirsts(head(L), compound-statement-Firsts) == true
            and compound-statement-Analysis(L ; SK) =/= ErrorUple
          then compound-statement-Analysis(L ; SK)
          else ErrorUple fi
   fi    fi .
```

Finalement, la fonction statement-Analysis(L ; SK) est appelée par sequence-of-statements-Analysis(L ; SK) pour l'analyse d'une séquence d'instructions. La fonction sequence-of-statements-Analysis(L ; SK) renvoie le code retourné par statement-Analysis(L ; SK) entouré par BeginCode et EndCode en intégrant juste après BeginCode et avant le code de statement-Analysis(L ; SK), une chaîne de caractères "sequence of statements" pour confirmer que ce code concerne une séquence d'instructions :

```
sequence_of_statements ::= statement {statement}
op sequence-of-statements-Analysis :  Uple -> Uple .
eq sequence-of-statements-Analysis(L ; SK) =
if statement-Analysis(L ; SK) =/= ErrorUple
 then if optional-Statement-sequence-of-statements(statement-Analysis(L ; SK)) =/= ErrorUple
```

```
        then 1st(optional-Statement-sequence-of-statements(statement-Analysis(L ; SK))) ;
                push(BeginCode,  push(("sequence of statements"),
                StackAddition(StackAddition(2nd(statement-Analysis(L ; SK)),
                2nd(optional-Statement-sequence-of-statements(statement-Analysis(L ; SK)))),
                EndCode)))
      else ErrorUple  fi
 else ErrorUple fi .
```

2.3. Génération du Code

Cette phase prend la représentation intermédiaire précédente et génère le code ECATNet équivalent au programme Ada. Nous générons le code ECATNet équivalent au code Ada en se basant sur la représentation intermédiaire. Le code généré est sous forme d'une hiérarchie des modules fonctionnels et puis systèmes. L'hiérarchie des modules fonctionnels implémente les types de données. Le module système implémente le comportement concurrent des tâches ainsi que leurs communications. Le module système importe le dernier module fonctionnel dans l'hiérarchie des modules fonctionnels. Nous expliquons une partie de code qui sert à générer le code ECATNet relatif à l'instruction de l'assignation. Nous avons déclaré dans notre fichier concernant la génération du code les variables suivantes :

```
vars Id RuleOrder Place1  : String .  vars L : List .  var SK  : Stack .
```

La fonction suivante Create-RulesFor-assg-Stmts(L, SK, Id, RuleOrder, Place1) génère le code ECATNet d'une instruction d'affectation. Tel que L est la liste contenant l'identificateur et les variables locales de la tâche, SK est la pile contenant la partie de code concernant l'affectation pour laquelle, nous allons générer son code. Id est le nom de l'unité (tâche, package, entrée …), RuleOrder est l'ordre de la prochaine la règle de réécriture à générer. Place1 sauvegarde l'ordre de la prochaine place à générer. Create-RulesFor-assg-Stmts(L, SK, Id, RuleOrder, Place1) commence par enlever BeginCode et EndCode en appelant StackSubstraction(pop(SK), EndCode). Puis, elle appelle Create-RulesFor-assg-Stmts-1(L, StackSubstraction(pop(SK), EndCode), Id, RuleOrder, Place1). StackSubstraction(pop(SK), EndCode) permet de retourner dans ce cas un code :

```
op Create-RulesFor-assg-Stmts : List Stack String String String  -> String .
eq Create-RulesFor-assg-Stmts(L, SK, Id, RuleOrder, Place1) =
Create-RulesFor-assg-Stmts-1(L, StackSubstraction(pop(SK), EndCode), Id, RuleOrder, Place1) .
```

Ce code retourné soit Cd est composé de quatre paramètres sauvegardant les parties de l'affectation, le premier est "assg", le second est la partie gauche de l'affectation soit 2ndC(Cd) et la troisième partie de l'affectation soit 4rthC(Cd). La fonction Create-Tuple(L) transforme cette liste à un n-uplet. Si L = a1 . a2 . .. .an, alors le n-uplet obtenu est de la forme (a1, a2, .., an). La fonction ReplaceEltinList(2ndC(Cd), 4rthC(Cd), L) permet de remplacer l'occurrence de la variable 2ndC(Cd) dans la liste L par l'expression 4rthC(Cd) :

```
op Create-RulesFor-assg-Stmts-1  : List Stack String String String -> String .
eq Create-RulesFor-assg-Stmts-1(L, Cd, Id, RuleOrder, Place1) =
" rl [" + NewRule(RuleOrder, Id) + "] : < " + NewPlace(Place1, Id) + " ;  " + Create-Tuple(L) + " > "
+ " => " + " < " + NewPlace(SuccNumber(Place1), Id) + " ; " + Create-Tuple(ReplaceEltinList(2ndC(Cd), 4rthC(Cd),
L)) + " > . "  .
```

La fonction SuccNumber(Place1) permet de calculer le successeur de la valeur Place1. En occurrence, Place1 est une chaîne de caractères, mais elle simule une valeur numérique. Par exemple, SuccNumber("12") retourne la valeur "13". La fonction NewPlace(Place1, Id) génère un nom de place contenant le nom de l'unité (tâche, d'entrée, package,…) et possède un numéro sauvegardé par Place1 :

```
op NewPlace : String String  -> String .
eq NewPlace(Place1, Id) =  Id + "-"  + "Place"  + "-" + Place1 .
```

Les variables de la tâche sont stockées dans une liste L sauvegardée au niveau du code. La fonction Create-Tuple(L) transforme cette liste à un n-uplet. Si L = a1 . a2 . .. .an, alors le n-uplet obtenu est de la forme (a1, a2, .., an) :

```
op Create-Tuple : List -> String .
eq Create-Tuple(L) =  if L == empty   then ""   else "(" + head(L) + Create-Tuple-1(tail(L))  fi .

op Create-Tuple-1 : List -> String .
eq Create-Tuple-1(L) = if L == empty    then ")"    else  ", " + head(L) + Create-Tuple-1(tail(L))  fi .
```

Maintenant, nous expliquons une autre fonction Create-RulesFor-entry-call-Stmt(L, SK, Id, RuleOrder, Place1) qui sert à générer le code ECATNet d'une instruction d'un appel d'une entrée. Tel que SK est la pile contenant la partie de code concernant l'appel de l'entrée pour laquelle, nous allons générer son code. L, Id, RuleOrder et Place1 ont la même signification que dans la fonction Create-RulesFor-assg-Stmts(L, SK, Id, RuleOrder, Place1). Dans le code crée par Create-RulesFor-entry-call-Stmt(L, SK, Id, RuleOrder, Place1), CreateTaskAsk-Place(entry-call-name(SK)) crée un nom de place contenant le nom de l'entrée appelée. En fait, la fonction entry-call-name(SK) retourne le nom de l'entrée appelée. La première règle de réécriture construite, permet au jeton de la tâche qui Create-Tuple(L) de se mettre dans la place de l'entrée entry-call-name(SK). Cette place qui s'appelle CreateTaskAsk-Place(entry-call-name(SK)) se trouve aussi dans le code de l'entrée appropriée et sert à recevoir les tâches qui appellent cette entrée. La fonction entry-call-list(SK) retourne les paramètres actuels passés à l'entrée appelée :

```
op Create-RulesFor-entry-call-Stmt : List Stack String String String -> String .
eq Create-RulesFor-entry-call-Stmt(L, SK, Id, RuleOrder, Place1) =

" rl [" + NewRule(RuleOrder, Id) + "] : "
+ " < " + NewPlace(Place1, Id) +  " ; " + Create-Tuple(L) + " > "
+ " => " + " < " + CreateTaskAsk-Place(entry-call-name(SK)) +  " ; "
+ "(" + Create-Tuple(L) + " ; " + Create-Tuple(entry-call-list(SK)) + ") >  . "

+ " rl [" + NewRule(SuccNumber(RuleOrder), Id) + "] : "
+ " < " + CreateReturnFromCall-Place(entry-call-name(SK)) + " ; "
+ "(" + Create-Tuple(L) + " ; " + Create-Tuple(entry-call-list(SK)) + ") >  "
+ " => " + " < " + NewPlace(SuccNumber(Place1), Id) +  " ; " + Create-Tuple(L) + " > . "  .
```

La deuxième règle de réécriture construite par cette fonction représente le retour de l'appel de l'entrée. Normalement, au niveau du code de l'entrée, nous avons une place dite CreateReturnFromCall-Place(entry-call-name(SK)). Cette règle de réécriture permet à la tâche de quitter cette place vers une nouvelle place pour suivre son activité.

La fonction entry-call-name(SK) retourne le nom de l'entrée appelée :

```
op entry-call-name : Stack -> String .
eq entry-call-name(SK) =
if SK == emptystack  then ""
else if 1stC(top(SK)) == "entry call"
      then 2ndC(top(SK))  else entry-call-name(pop(SK)) fi  fi .
```

La fonction entry-call-list(SK) retourne les paramètres actuels passés à l'entrée appelée :

```
op entry-call-list : Stack -> String .
eq entry-call-list(SK) =
if SK == emptystack
then ""   else if 1stC(top(SK)) == "entry call"
            then 3rdC(top(SK)) else entry-call-list(pop(SK)) fi   fi .
```

3. Exemple

Nous montrons dans cette section, le comportement des différentes phases de notre translateur sur un exemple simple. Considérons un code Ada composé d'une seule instruction : Buffer.Write(Char) ;

L'analyseur lexical transforme ce code à un 'Uple' composé d'une liste des chaînes de caractères et une pile vide : "Write" . "(" . "Char" . ")" . ";" ; emptystack

L'analyse syntaxique transforme cet appel d'une entrée à une représentation sous forme de pile contenant trois éléments BeginCode, ("entry call", "Write", "Char") et EndCode. Les deux constantes BeginCode et EndCode servent à délimiter le code de l'appel de l'entrée. Le code au milieu contient trois chaînes de caractères : "entry call", "Write" et "Char". En fait, la première chaîne de caractères permet d'indiquer que ce code concerne l'appel d'une entrée. La deuxième chaîne de caractères porte le nom de l'entrée appelée et la troisième chaîne peut être une liste des paramètres de l'entrée appelée. Le résultat obtenu après l'analyse de cette instruction est le 'Uple' composé par la liste d'entrée qui est vide dans notre cas, et la représentation sous de forme de pile de l'appel de l'entrée : empty ; push(BeginCode, push(("entry call", "Write", "Char"), EndCode))

L'analyse syntaxique génère une représentation intermédiaire sous forme de pile (retournée par la fonction qui analyse une séquence d'instructions. Cette dernière contient une seule instruction dans notre exemple :

```
empty ;  push(BeginCode, push("sequence of statements",
          push(BeginCode, push( "entry call", "Write", "Char", "", EndCode ))))
```

Nous donnons uniquement une petite partie de code généré par notre application (générateur). Le générateur de code génère une déclaration de variable var Generic-producer : producer . Ceci permet d'écrire un seul code ECATNet pour tous les producteurs. La règle de réécriture producer-Rule-4 traduit en partie l'appel de l'entrée Write. Cet appel est

exprimé en terme de mettre le n-uplet (Generic-producer, Char) dans la place Task-Ask-write-Place. Notons que les places utilisées dans cette partie de code sont toutes créées avant leur utilisation ainsi que le code exprimant l'entrée Write. Après l'exécution de cette règle de réécriture, le jeton (Generic-producer, Char) se trouve au niveau de l'entrée Write qui n'est pas expliqué ici pour la simplicité. Normalement, à la fin de l'exécution de cette entrée, ce jeton se retrouve à une place dite Return-From-Call-Entry-Write-Place. La règle de réécriture producer-Rule-5 reprend le jeton et le met dans une place producer-Place-4 pour que la tâche continue son exécution :

```
var Generic-producer : producer .
rl [producer-Rule-4] :  < producer-Place-3 ; (Generic-producer, Char) >
=>  < Task-Ask- Write-Place ; ((Generic-producer, Char) ; (Char)) >  .
rl [producer-Rule-5] :  < Return-From-Call-Entry- Write-Place ; ((Generic-producer, Char) ; (Char)) >
=>  < producer-Place-4 ; (Generic-producer, Char) > .
```

4. Conclusion

Dans ce chapitre, nous avons présenté notre démarche dans la réalisation d'un translateur Ada-ECATNet. Nous avons expliqué les différentes étapes de ce translateur. Nous avons détaillé les types de données utilisées dans ces différentes phases. Le translateur a été réalisé dans le langage Maude. Dans le cadre de l'implémentation de ce translateur, nous avons utilisé le mode fonctionnel du langage Maude. Uniquement l'étape de l'analyse lexicale qui a été réalisée dans le langage Delphi.

Conclusion

Dans cette thèse, nous avons proposé une étude théorique suivie d'une réalisation pratique de certains outils basés sur le langage de la logique de réécriture Maude pour une catégorie des réseaux ECATNets. Nous avons adopté l'utilisation des ECATNets pour translater les programmes concurrents Ada pour leur vérification. A notre sens, les ECATNets présentent une puissance d'expression assez élevée pour décrire plusieurs concepts du langage Ada. D'un autre côté, les ECATNets ont une batterie intéressante des outils d'analyse, comme les règles de réduction et le Model Checking de Maude.

Dans cette thèse, nous avons décrit notre outil d'édition et de simulation pour les ECATNets. Notre objectif à travers cet outil est de faciliter à l'utilisateur la spécification de son système avec les ECATNets sans perdre l'avantage de l'aspect graphique. Nous avons montré à travers un exemple d'ECATNet, le gain obtenu en utilisant l'outil proposé par rapport à une utilisation directe de la version textuelle des ECATNets dans Maude.

Ensuite, nous avons proposé un outil basé sur la logique de réécriture pour une analyse dynamique des ECATNets. Cet outil nous permet d'énumérer l'espace d'états (construction de graphe de couverture) et de vérifier quelques propriétés de base. De tel algorithme est motivé par le fait que le Model Checking ne peut pas vérifier les modèles à états infinis. En plus, les ECATNets ne disposent d'aucun algorithme d'analyse dynamique par rapport à d'autres types de réseaux de Petri ordinaires ou de haut niveau. En plus, l'étude des places à capacité infinie est importante en ce qui concerne la décidabilité.

De même, nous avons proposé un outil basé sur Maude qui met en application quelques règles de réduction pour les ECATNets. De telles règles de réduction sont définies au préalable dans [Sch97] pour les RPAs. Nous avons présenté trois règles et nous avons indiqué comment nous les avons adaptées pour les ECATNets. Pour la simplicité, nous avons montré l'implémentation dans Maude pour une seule règle.

Grâce à l'intégration des ECATNets dans Maude et la réflectivité de ce langage, la création d'un outil implémentant l'analyseur dynamique ou les règles de réduction pour les ECATNets en utilisant Maude est largement simplifiée par rapport à sa création en utilisant un autre langage. Nous avons montré dans cette thèse que l'implémentation en utilisant Maude des règles de réduction n'est pas compliquée. Peu de lignes de code sont suffisantes pour de telle implémentation. En fait, l'utilisation des services offerts par Maude pour gérer les méta-modules, nous a libéré de plusieurs détails.

Dans notre travail concernant la translation d'un programme Ada à un ECATNet, nous avons proposé une méthode systématique permettant l'expression de certains concepts relatifs en particulier au concept de 'multi-tâches' dans les ECATNets. En plus, nous avons implémenté cette translation. Notre compilateur Ada-ECATNet a été réalisé en grande partie à l'aide du langage Maude.

Nous avons concentré nos efforts à développer une représentation compactée des Ada-ECATNet rendant le processus (simulation, statique, dynamique ou Model Checking) plus efficace. En plus, cette représentation permet l'application des outils spécifiques des ECATNets ou des autres réseaux de Petri de haut niveau (algébrique ou colorés) après les avoir adapter aux ECATNets. Nous avons montré à travers des exemples la possibilité d'appliquer des règles de réduction des RPAs adaptées aux ECATNets sur la représentation compactée Ada-ECATNets. Cette double réduction sur un Ada-ECATNet permet une diminution considérable sur le temps d'exécution et sur la consommation de mémoire de quelques méthodes d'analyse. Ceci a été montré à travers un exemple.

Le choix de Maude est justifié de point de vue théorique et pratique pour ce travail. La puissance de Maude de remplir les besoins des ECATNets en termes de spécification, programmation, simulation et vérification implique qu'il n'est pas nécessaire de traduire les ECATNets dans plusieurs langages pour de telles exigences. Par conséquent, nous évitons n'importe quelle traduction erronée des ECATNets dans plusieurs langages ainsi que tous les risques relatifs à une perte sémantique.

Comme perspectives, nous envisageons à l'avenir la finalisation de certains points. Dans le cadre de l'adaptation des règles de réductions des RPAs pour les ECATNets, nous avons terminé l'adaptation et l'implémentation des sept règles de réduction pour les ECATNets et il reste encore deux règles. L'adaptation et l'implémentation pour les ECATNets de ces deux règles restantes, sont en cours de développement. Ces règles présentent des besoins d'implémentation différents par rapport à ceux des autres règles. En plus, nous comptons démontrer que les propriétés préservées dans les RPAs par les règles de réduction sont aussi préservées dans les ECATNets après l'application des règles de réduction. L'algorithme de l'analyse dynamique proposé ainsi que son implémentation dans le cadre de ce travail ne tient pas compte d'aucune réduction de la taille de l'espace-état. Nous comptons l'étude d'une réduction de cet espace-état. Une approche envisagée est basée sur la concurrence de la logique de réécriture, c'est-à-dire la possibilité de franchir plusieurs transitions en même temps au lieu de franchir une seule transition pour calculer le marquage accessible. Dans ce cas, nous pouvons 'sauter' plusieurs marquages intermédiaires qui n'affectent pas la vérification des propriétés statiques comme la détection des deadlock. En dernier point, nous planifions l'implémentation des règles de réduction que nous avons proposé pour les Ada-ECATNets.

Références

[Bar97] K. Barkaoui and J-F Pradat-Peyre. "Petri Nets Based Proofs of Ada 95 Solution for Preference Control". In Proceedings APSEC '97 / ICSC '97, 1997.

[Bar98] K. Barkaoui and J-F Pradat-Peyre. "Verification in Concurrent Programming with Petri nets Structural Techniques". In Proceedings Third IEEE International High-Assurance Systems Engineering Symposium November 13 - 14, 1998 Washi, 1998.

[Bea01] M. Beaudouin-Lafon and al.. "CPN/Tools: A Tool for Editing and Simulating Coloured Petri Nets - ETAPS Tool Demonstration Related to TACAS". In: LNCS 2031: Tools and Algorithms for the Construction and Analysis of Systems, pages 574-pp. Springer Verlag, 2001.

[Bel00] F. Belala, M. Bettaz, L. Petruci. "Concurrent Systems Analysis Using ECATNets". Logic Journal of the IGPL, pp. 149-164, 2000.

[Ber92] G. Berthelot, C. Johnen, and L. Petrucci. "PAPETRI : environment for the analysis of Petri nets". Volume 3 of Series in Discrete Mathematics and Theoretical Computer Science (DIMACS), p. 43-55. American Mathematical Society, 1992.

[Bet93] M. Bettaz, M. Maouche. "How to specify Non Determinism and True Concurrency with Algebraic Term Nets". Volume 655 of LNCS, Spring-Verlag, p. 11-30, 1993.

[Bli00] J. Blieberger, B. Bugstaller, B. Scholz. "Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs". In Proc. Of the Ada-Europe International Conference on Reliable Software Technologies, Postdam, Germany, June 2000, Springer, 2000.

[Bou04a] N. Boudiaf, A. Chaoui. "Towards Automated Analysis of Ada-95 Tasking Behavior By Using ECATNets". In Proceedings ISIIT'04, Jordan, 2004.

[Bou04b] N. Boudiaf, A. Chaoui. "Dynamic Analysis Algorithm for ECATNets", Proceedings of Moca'04, October 11-13, 2004, Aarhus, Denmark, pp 47-63, Daniel Moldt Editor, ISSN 0105-8517, 2004.

[Bou04c] N. Boudiaf, A. Chaoui. "A Rewriting Logic based Tool for Automated Support of ECATNets Reachability Analysis". Proceedings of CSITeA-04, Cairo, 27-29, 2004, Egypt (Organized by Winona State University, USA), ISBN 0.9742059.1.5.

[Bou05] N. Boudiaf, A. Chaoui. "A Rewriting Logic Based Tool for ECATNets' Analysis : Edition and Simulation Steps Description". European Journal of Scientific Research. Vol 5. No.1, AMS Publishing inc. (Austria), 2005.

[Bou06a] N. Boudiaf, K. Barkaoui, Allaoua Chaoui. "Implémentation Des Règles de Réduction des ECATNets dans Maude". Proceedings de la Conférence Mosim'06, 2-5 avril, Rabat, Maroc, pp. 1505-514, 2006.

[Bou06b] N. Boudiaf, K. Barkaoui and Allaoua Chaoui. "Applying Reduction Rules to ECATNets". Proceedings of AVIS'06 Workshop (Co-located with the conferences ETAPS'06), 1st April, Vienna, Austria, R. Bharadwaj and S. Mukhopadhyay (Eds.), 2006, to appear in Electronic Notes in Theoretical Computer Science (ENTCS), 2006.

[Bou06c] N. Boudiaf, A. Chaoui. "A Compact Representation of Ada95-ECATNets Using Rewriting Logic". Abstract accepted in workshop wadt'06 (La Roche, Belgium), 2006.

[Bou06d] N. Boudiaf, A. Chaoui. “Double Reduction of Ada-ECATNet Representation Using Rewriting Logic”. Enformatika Journal (Transactions on Engineering, Computing and Technology), Volume 15, ISSN 1305-5313, pp. 278-284, October 2006.

[Bra93] T. Bräunl. "Parallel Pogramming". Prentice-Hall, Englewood Cliffs NJ, 1993, pp. (X, 270).

[Bru99] E. Bruneton and J-F. Pradat-Peyre. “Automatic Verification of Concurrent Ada Programs”, In Proceedings Reliable Software Technologies-Ada-Europe'99, 1999.

[Bul91] B. Bütler, R. Esser, R. Mattmann. “A Distributed Simulator for High Order Petri Nets”. In: Rozenberg, G.: Lecture Notes in Computer Science, Vol. 483; Advances in Petri Nets 1990, pages 47-63. Berlin, Germany: Springer-Verlag, 1991.

[Bur99] A. Burns and A. J. Wellings. "How to Verify Concurrent Ada Programs : The Application of Model Checking". The Application of Model Checking, Ada Letters, Vol XIX, pp78-83, June 1999.

[Bur92] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill and L. H. Hwang. “Symbolic model checking”. 1020 states and beyond. Information & Computation, 92(2) : 142-170, 1992.

[Cha90] W. Chanat, C. Yaobin. "A Real-Time Petri Nets Simulator for Automated Manufacturing Systems". In: Proceedings of the 5th IEEE International Symposium on Intelligent Control, 1990, Philadelphia, PA, USA, pages 999-1004. Piscataway, NJ, USA: IEEE Service Center, 1990.

[Che96] J. Cheng and K. Ushijima. "Tasking Deadlocks in Ada 95 Programs and Their Detection". Reprinted from A. Strohmeir (Ed.), Reliable Software Technologies – Ada-Europe '96, Proc. 1996 Ada-Europe Annual International Conference Montreux, Switzerland, june 10-14, 1996, LNCS 1088, Springer-Verlag, 1996.

[Che97] J. Cheng. “Task Dependence Nets for Concurrent Systems with Ada 95 and Its Applications”. ACM TRI-Ada International Conference, pp. 67-78, 1997.

[Che02] Z. Chen, B. Xu, J. Zhao and H. Yang. "Static Dependency Analysis for Concurrent Ada 95 Programs".

[Cla03] M. Clavel, F. Duran, S. Ecker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott. “The Maude 2.0 System”. In Proc. Rewriting Techniques and Applications (RTA), Volume 2706 of LNCS, Spring-Verlag , p. 76-87., 2003.

[Cla05] M. Clavel, F. Dur´an, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, C. Talcott. “Maude Manual (Version 2.2) ”. December 2005.

[Cor02] L. A. Cortés, G. Jervan. “SimPRES A Simulator for Time Petri Nets”. Dept. of Computer and Information Science, Linköpings universitet, SWEDEN, 2002. http://www.ida.liu.se/~luico/SimPRES/

[Gor04] Goran Center “Petri .NET Simulator 2.0 – User’s manual” Bigeneric Technologie, 2004. http://www.bigeneric.com/

[Dwy96] M.B. Dwyer and L.A. Clarke. A compact petri net representation and its implications for analysis. IEEE Transactions of Software Engineering, 22(11), November 1996.

[Fin93] A. Finkel. “The Minimal Coverability Graph for Petri Nets”. In: Rozenberg, G.: Volume 674 of LNCS,; Advances in Petri Nets 1993, Springer-Verlag, p. 210-243, 1993.

[Ged99] Ravi K. Gedela, Sol M. Shatz and Haiping Xu. "Compositional Petri Net Models of Advanced Tasking in Ada-95". Comput. Lang. 25(2): 55-87, 1999.

[Gir93] C. Girault and P. Estraillier, “CPN-AMI”. MASI Lab, University Paris VI, France.

[Gor04] Goran Center “Petri .NET Simulator 2.0 – User’s manual” Bigeneric Technologie, 2004. http://www.bigeneric.com/

[Hau98] J. Hauptmann, B. Hohberg, E. Kindler, I. Schwenzer, M. Weber. “The Petri Net kernel – Documentation of the Application Interface”. Petri Net Kernel Team Humboldt-University Berlin, Germany, 1998.

[Eng97] J. English. "Ada 95: The Craft of Object-Oriented Programming". Prentice Hall, 1997.

[Hen05] J. Hendrix, M. Clavel and J. Meseguer. "A Sufficient Completeness Reasoning Tool for Partial Specifications". 16th International Conference on Term Rewriting and Applications, Volume 3467 of LNCS, Spring-Verlag, p. 165–174, 2005.

[Hul96] J. Hulaas. "An Evolutive Distributed Algebraic Petri Nets Simulator". In A. Javor, A. Lehmann, and I. Molnar, editors, Modelling and Simulation 1996, pages 348--352, Budapest, Hungary, June 2-6 1996. 10th European Simulation Multiconference ESM96, Society for Computer Simulation International. http://citeseer.ist.psu.edu/hulaas96evolutive.html

[Iso95] ISO/IEC 8652. "Information Technology – Programming Languages – Ada", 1995.

[Iso01] ISO/IEC 8652:1995(E) with with Technical Corrigendum 1 :2000 — Ada Reference Manual, 2001.

[Kel95] C. Kelling. "TimeNET–Sim–a parallel simulator for stochastic Petri nets". 28th Annual Simulation Symposium, April 25–28, Santa Barbara, 1995.

[Lin95] T. Lindner. "Formal Development of Reactive Systems : Case Study Production Cell". *Volume 891 of LNCS,* Spring-Verlag, p. 7-15, 1995.

[Liu02] Y. Liu, B. Xu and Z. Chen. "Detecting Deadlock in Ada Rendezvous Flow Structure Based on Process Algebra". C. George and H. Mia (Eds.): ICFEM 2002, LNCS 2495, pp. 262-274, Springer-Verlag Berlin Heidelberg 2002.

[Mao97] M. Maouche, M. Bettaz, G. Berthelot and L. Petrucci. "Du vrai Parallélisme dans les Réseaux Algébriques et de son Application dans les Systèmes de Production". Conférence Francophone de Modélisation et Simulation (MOSIM'97), Hermes, p. 417-424., 1997.

[Mes92] J. Meseguer, "Conditional Rewriting Logic as an Unified Model of concurrency". Theoretical Computer Science, 1992.

[Mes96] J. Meseguer. "Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report". Seventh International Conference on Concurrency Theory (CONCUR'96), Volume 1119 of LNCS, Springer Verlag, p. 331-372, 1996.

[Mes00] J. Meseguer. "Rewriting logic and Maude: a Wide-Spectrum Semantic Framework for Object-based Distributed Systems". In S. Smith and C.L. Talcott, editors, Formal Methods for Open Object-based Distributed Systems, (FMOODS'2000), p. 89-117. Kluwer, 2000.

[Mat05] MathTools. "Petri Nets Simulator 1.0". MathTools Company, Cairo, Egypt, 2005. http://www.eurodownload.com/download-software/5249/Petri-Nets-Simulator.html

[Met96] Meta Software Corporation. "Design/CPN Tutorial for X–Windows : Version 2.0". Cambridge, England, 1993. http://www.daimi.au.dk/designCPN/man/

[Mur89] T. Murata, B. Shenker, S. M. Shatz. "Detection of Ada Static Deadlocks Using Petri Nets Invariants". IEEE trans. On Software Engineering, vol. 15, No. 3, pp 314-326, 1989.

[Nic95] D. M. Nicol, A.S Miner. "The fluid Stochastic Petri Net Simulator". Sixth International Workshop on Petri Nets and Performance Models –PNPM'95, Durham, North Carolina, USA, October 03–06, 1995.

[Pin93] V. Pinci and L. Zand. "DESIGN/CPN". USA, 1993.

[Roc99] S. Roch and P.H. Strake. "INA (Integrated Net Analyzer) : Version 2.2". Manual, Humboldt-Universität zu Berlin Institut für Informatik, Lehrstuhl für Automaten- und Systemtheorie, 1999.

[Sch97] K. Schmidt. "Applying Reduction Rules to Algebraic Petri Nets". *TKK Monoistamo; Otaniemi 1997*, ISSN 0783 5396, ISBN 951-22-3495-5, 1997.

[Sha88] S.M. Shatz, W.K. Cheng. "A Petri Net Framwork For Automated Static Analysis of Ada Tasking Behavior", J.\ Syst. Software, Vol. No 8, pp. 343-359, Dec 1988.

[Tay83] R. N. Taylor. "Complexity of Analyzing the Synchronization Structure of Concurrent Programs". Acta Informatica, 19:57-84, 1983.

[Tou99] E. Tourtelot. "Atelier de Simulation de Réseaux de Petri & Interactive Petri Net Simulator". http://etourtelot.free.fr/download/petri/petri.html, 03 Novembre 1999.

[Tri99] K. S. Trivedi. "SPNP User's Manual Version 6.0". CACC, Department of Electrical and Computer Engineering, Duke University, Durham, USA, 1999.

[Var95] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. "PROD reference manual". Technical Report B13, Helsinki University of Technology, Digital Systems Labora tory, Espoo, Finland, August 1995.

[Wel02] L. Wells, "CPN–Tools : Computer Tool for Coloured Petri Nets" , Version 2, Fri 06 Dec 2002 Department of Computer Science University of Aarhus, Denmark, 2002. http://www.daimi.au.dk/CPNTools/

[Zbi95] S. Zbigniew. "PN-tools: Environment for the Design and Analysis of Petri Nets". In: Control and Cybernetics, Vol. 24, No. 5, pages 200-222. 1995.

Annexe :

Résumé d'une Partie de Syntaxe d'Ada

graphic_character ::= identifier_letter | digit | space_character | special_character

identifier ::= identifier_letter {[underline] letter_or_digit}

letter_or_digit ::= identifier_letter | digit

numeric_literal ::= decimal_literal

decimal_literal ::= numeral [.numeral] [exponent]

numeral ::= digit {[underline] digit}

exponent ::= **E** [**+**] numeral | **E -** numeral

base ::= numeral

character_literal ::= 'graphic_character'

string_literal ::= **"**{string_element}**"**

string_element ::= **""** | non_quotation_mark_graphic_character

basic_declaration ::= type_declaration | subtype_declaration | object_declaration

defining_identifier ::= identifier

type_declaration ::= full_type_declaration

full_type_declaration ::= **type** defining_identifier **is** type_definition;

| task_type_declaration | protected_type_declaration

type_definition ::= enumeration_type_definition | integer_type_definition | array_type_definition

subtype_declaration ::= subtype defining_identifier **is** subtype_indication**;**

subtype_indication ::= subtype_mark [constraint]

subtype_mark ::= subtype_name

constraint ::= scalar_constraint

scalar_constraint ::= range_constraint | digits_constraint

object_declaration ::= defining_identifier_list **:** [constant] subtype_indication [**:=** expression]**;**

| single_task_declaration | single_protected_declaration

defining_identifier_list ::= defining_identifier {**,** defining_identifier}

range_constraint ::= **range** range

range ::= simple_expression **..** simple_expression

enumeration_type_definition ::= (enumeration_literal_specification {, enumeration_literal_specification})

enumeration_literal_specification ::= defining_identifier

integer_type_definition ::= signed_integer_type_definition

signed_integer_type_definition ::= **range** static_simple_expression1 **..** static_simple_expression2

digits_constraint ::= **digits** static_expression

array_type_definition ::= constrained_array_definition

constrained_array_definition ::= **array (**discrete_subtype_definition**) of** component_definition

discrete_subtype_definition ::= range

component_definition ::= subtype_indication

component_declaration ::= defining_identifier_list : component_definition [:= default_expression];

declarative_part ::= {declarative_item}

declarative_item ::= basic_declarative_item | body

basic_declarative_item ::= basic_declaration

body ::= proper_body

proper_body ::= task_body | protected_body

expression ::= relation {**and** relation} | relation {**or** relation} | relation {**xor** relation}

relation ::= simple_expression [relational_operator simple_expression]

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | **abs** primary | **not** primary

primary ::= numeric_literal | (expression**)**

logical_operator ::= **and** | **or** | **xor**

relational_operator ::= **=** | **/=** | **<** | **<=** | **>** | **>=**

binary_adding_operator ::= **+** | **-** | **&**

unary_adding_operator ::= **+** | **--**

multiplying_operator ::= ***** | **/** | **mod** | **rem**

highest_precedence_operator ::= ****** | **abs** | **not**

sequence_of_statements ::= statement {statement}

statement ::= simple_statement | compound_statement

simple_statement ::= null_statement | assignment_statement | exit_statement | return_statement

| entry_call_statement | abort_statement

compound_statement ::= if_statement | loop_statement | accept_statement | select_statement

null_statement ::= **null;**

assignment_statement ::= variable_name **:=** expression;

if_statement ::= **if** condition **then** sequence_of_statements

[**else** sequence_of_statements] **end if;**

condition ::= boolean_expression

loop_statement ::= [loop_statement_identifier**:**] [iteration_scheme]

loop sequence_of_statements **end loop** [loop_identifier]**;**

iteration_scheme ::= **while** condition

loop_parameter_specification ::= defining_identifier **in** discrete_subtype_definition

exit_statement ::= **exit** [loop_name] [**when** condition]**;**

defining_program_unit_name ::= defining_identifier

parameter_profile ::= [formal_part]

formal_part ::= (parameter_specification {; parameter_specification})

parameter_specification ::= defining_identifier_list : mode subtype_mark [:= default_expression]

mode ::= [**in**] | **in out** | **out**

actual_parameter_part ::= (parameter_association {**,** parameter_association})

parameter_association ::= explicit_actual_parameter

explicit_actual_parameter ::= expression | variable_name

return_statement ::= **return** [expression]**;**

package_body ::= **package body** defining_program_unit_name **is**

declarative_part **end** [[parent_unit_name.]identifier]**;**

task_type_declaration ::= **task type** defining_identifier [**is** task_definition]**;**

single_task_declaration ::= **task** defining_identifier [**is** task_definition]**;**

task_definition ::= {task_item} **end** [task_identifier]

task_item ::= entry_declaration

task_body ::= **task body** defining_identifier **is**

declarative_part1 **begin** handled_sequence_of_statements **end** [task_identifier];

declarative_part1 ::= defining_identifier_list **:** [constant] subtype_indication [**:=** expression]**;**

protected_type_declaration ::= **protected type** defining_identifier **is** protected_definition;

single_protected_declaration ::= **protected** defining_identifier **is** protected_definition;

protected_definition ::= { protected_operation_declaration }

[**private** { protected_element_declaration }] **end** [protected_identifier]

protected_operation_declaration ::= entry_declaration

protected_element_declaration ::= component_declaration

protected_body ::= **protected** body defining_identifier **is**

{ protected_operation_item } **end** [protected_identifier]**;**

protected_operation_item ::= entry_body

entry_declaration ::= entry defining_identifier parameter_profile**;**

accept_statement ::= **accept** entry_direct_name parameter_profile [**do**

handled_sequence_of_statements **end** [entry_identifier]]**;**

entry_body ::= **entry** defining_identifier entry_body_formal_part entry_barrier **is**

declarative_part **begin** handled_sequence_of_statements **end** [entry_identifier]**;**

entry_body_formal_part ::= parameter_profile

entry_barrier ::= **when** condition

entry_call_statement ::= entry_name [actual_parameter_part]**;**

select_statement ::= selective_accept | conditional_entry_call | asynchronous_select

selective_accept ::= **select** [guard] select_alternative { **or** [guard] select_alternative }

[**else** sequence_of_statements] **end select;**

guard ::= **when** condition **=>**

select_alternative ::= accept_alternative | terminate_alternative

accept_alternative ::= accept_statement

terminate_alternative ::= **terminate;**

entry_call_alternative ::= entry_call_statement

conditional_entry_call ::= **select** entry_call_alternative **else** sequence_of_statements **end select;**

abortable_part ::= sequence_of_statements

abort_statement ::= **abort** task_name {**,** task_name}**;**