DEA EN INFORMATIQUE

Sémantique et vérification de langages de coordination par les réseaux de Petri

Mayala Lusilabo mfumua'Nsi, Frumence

Award date:
2005

Sémantique et Vérification de Langages de Coordination par les Réseaux de Petri

Frumence MAYALA LUSILABO MFUMU A'NSI

Mémoire présenté en vue de l'obtention du Diplôme d'Etudes Approfondies en Informatique

Année académique 2004-2005

Remerciements

Nous tenons à exprimer nos sincères remerciements à M. Jean-Marie Jacquet qui a dirigé ce travail. Ses indications, ses remarques, son approche des problèmes rencontrés nous ont été très instructives.

Nos vifs remerciements s'adressent également Isabelle Linden dont la disponibilité, les remarques et les suggestions nous ont été d'un grand apport dans la réalisation du présent travail.

A vous nos amis et membres de famille, toute notre gratitude pour votre présence, votre confiance et notre espérance partagée. A toi ma maman chérie, merci pour tout. A toi Iza, merci.

Résumé

Nous utilisons les réseaux de Petri contextuels avec arcs inhibiteurs pour exprimer la sémantique opérationnelle d'un langage de coordination de type Linda, $\mathcal{L}^r_\Psi$. Ce langage utilisant la composition alternative, la sémantique que nous proposons est orientée-location. Nous adoptons une vue statique du comportement d'un agent afin de limiter le nombre des places dans les réseaux. Nous montrons ensuite que la sémantique ainsi exprimée est équivalente à celle décrite dans le style de Plotkin. Nous proposons également une application qui génère le fichier de réseaux qui est traité par un outil d'analyse et de vérification des propriétés de réseaux de Petri, INA.

Abstract

In this thesis, we use contextual Petri nets with inhibitor arcs to express the operationnal semantics of a coordination language like Linda, $\mathcal{L}^r_\Psi$. Because this language uses the alternative composition, this net semantics is location-oriented. We consider the behaviour of an agent in the static manner in order to produce smaller nets. We show that this semantics is sound with respect to the semantics given in Plotkin's style. We develop an application which generates nets file from an agent expressed in $\mathcal{L}^r_\Psi$. The generated net file can be processed by a Petri nets analyzer called INA.

Table des figures

Table des matières

Introduction

L'évolution phénoménale des systèmes informatiques a imposé l'utilisation des composants de plus en plus disparates, de plus en plus hétérogènes. La question de leur coopération et de leur communication s'est alors logiquement posée avec acuité. Il s'est peu à peu imposé la nécessité d'avoir des nouveaux outils, des nouveaux langages, des nouvelles approches pour faire face à ce genre de problèmes. Longtemps laissés en dehors des principaux courants de recherche, les langages de coordination peuvent être considérés aujourd'hui comme une partie des solutions apportées à ces problèmes, particulièrement dans le domaine des systèmes parallèles et des systèmes distribués complexes.

Linda est un des modèles les plus connus et, historiquement, le plus ancien des langages de coordination. Créé par D. Gelernter au milieu des années 1980, il n'est pas un langage de programmation, à proprement parlé, mais un langage qui définit un modèle de communication et de synchronisation entre les processus. Et comparativement aux champs qui avaient l'attention de la communauté des chercheurs dans le domaine de la programmation parallèle, notamment la *programmation orientée objet concurrente*, les *langages logiques concurrents* ou encore les *systèmes de programmation fonctionnelle*, Linda est *plus simple, plus puissant et plus élégant* [17]. Il a été conçu pour être indépendant de tous les langages de programmation mais pouvant *coopérer* avec chacun d'eux. En pratique, il est associé à un langage de programmation sous forme d'une librairie.

Par ailleurs, les réseaux de Petri sont quelques fois présentés dans la littérature comme l'un des formalismes à même de présenter "la concurrence de manière plus appropriée et naturelle" [6], contrairement aux langages algébriques. Dans le présent travail, nous proposons une lecture du comportement des agents exprimés dans un langage de coordination, en l'occurrence $\mathcal{L}^r_\Psi$, au moyen du formalisme des réseaux de Petri.

En effet, nous partons d'un langage de coordination (de type *Linda)*, $\mathcal{L}_{\Psi}^{r}$, ayant la particularité de permettre l'utilisation des données qui sont porteuses d'une information partielle, dans le but de définir sa sémantique opérationnelle au moyen du formalisme des réseaux de Petri. Le type particulier des instructions utilisées dans certains langages de coordination, dont celui qui nous intéresse dans cette rédaction, induit une certaine *adaptation* des réseaux de Petri pour exprimer de manière adéquate la sémantique opérationnelle dont nous avons parlé plus haut. Ainsi, à la place des réseaux de Petri que l'on peut appeler *classiques*, nous utilisons les réseaux de Petri dits *contextuels*. La sémantique opérationnelle basée sur les réseaux de Petri que nous définissons est équivalente à la sémantique opérationnelle définie au moyen des règles de transitions.

En dehors de l'introduction, ce travail est divisé en six chapitres. Dans le premier chapitre, nous présentons brièvement les langages de coordination, en nous attardant spécialement sur Linda. Nous détaillons, dans le deuxième chapitre, de manière particulière le langage de coordination, $\mathcal{L}_{\Psi}^{r}$, qui est un langage de type Linda. C'est sur ce langage que se base la suite de ce travail. Dans le troisième chapitre, nous parlons des réseaux de Petri *contextuels*. Nous y abordons certains de leurs traits caractéristiques, et nous y parlons également d'une approche pour transformer un réseau de Petri *contextuel* en un réseau de Petri *classique*. Dans le quatrième chapitre, nous présentons la sémantique opérationnelle du langage de coordination $\mathcal{L}_{\Psi}^{r}$ en utilisant les réseaux de Petri *contextuels*. Dans le cinquième chapitre, nous présenterons l'outil d'analyse des réseaux de Petri : Integrated Net Analyzer (INA, en sigle). Cet outil, comme son nom l'indique, permet d'analyser un réseau de Petri et d'étudier ses différentes propriétés. Dans le sixième chapitre, nous présentons l'application que nous avons conçue pour valider nos résultats. Cette application permet de transformer un agent décrit dans le langage $\mathcal{L}_{\Psi}^{r}$ sous forme d'un réseau de Petri décrit dans un format précis, propre à l'outil INA, avant de pouvoir être analysé. En dernier lieu, nous concluons en parlant de ce que nous pensons être notre apport tout au long de ce travail et, enfin, nous envisagerons des pistes éventuelles que ce travail peut avoir ouvert et qui peuvent constituer des objets de prochaines recherches.

Chapitre 1

Les langages de coordination

Dans ce chapitre, nous présentons brièvement les langages de coordination. Comme nous l'avons dit dans l'introduction, les langages de coordination peuvent être considérés comme une réponse aux multiples questions que posent la communication, la coopération, la coordination entre des composants de plus en plus hétérogènes des systèmes informatiques. Ce qu'il convient aujourd'hui d'appeler le *paradigme de coordination* offre une voie pour alléger les problèmes liés aux systèmes parallèles et distribués complexes. Ainsi, par exemple, la programmation d'un système parallèle peut être vue comme une combinaison de deux activités distinctes. La première portant sur le calcul et comprenant un certain nombre de processus impliqués dans la manipulation des données ; et la deuxième activité portant sur la communication et le coopération entre les processus [3].

Dans les paragraphes qui suivent, nous épinglons la différence qui existe entre les termes de "modèle" et de "langage" de coordination. Nous expliquons notre choix pour les langages de type Linda, et nous donnons certaines caractéristiques de ce type de langages.

1.1 Modèles et langages de coordination

Dans la littérature, il arrive assez souvent que les termes de "modèle" et de "langage" de coordination soient indifféremment utilisés. Et il est un fait que, généralement, cela n'entraîne pas de confusion si l'on sait en quoi s'en tenir. Il existe pourtant une nuance entre les deux termes. Selon N. Carriero et D. Gelernter, le modèle de coordination peut être défini comme "*la colle qui relie des activités séparées dans un ensemble*", tandis qu'un langage de

coordination est, lui, défini comme "*une incarnation linguistique d'un modèle de coordination*" [17]. Il apparaît donc que le modèle de coordination fournit la sémantique alors qu'un langage de coordination correspondant propose une syntaxe pour utiliser le modèle dans une implémentation [3].

Il existe plusieurs classifications des modèles et langages de coordination. Certaines se basent sur le type d'entités à coordonner, d'autres se basent sur les architectures supportées par le modèle, et d'autres encore sur les sémantiques auxquelles le modèle adhère. D'après Arbab et Papadopoulos [9], les modèles et les langages de coordination peuvent être divisés en deux grandes familles dont la première comprend les modèles et langages de coordination *orientés données* (*data-driven*) ; et la seconde famille est constituée des modèles et langages *orientés événements* (*control-driven*). Les premiers ont comme dénominateur commun le fait d'utiliser un espace partagé (TS) pour la communication des données. Et, l'activité essentielle tourne autour cet espace partagé des données. Quant aux seconds, ils utilisent les canaux et les ports pour leurs communications. Et leur activité tourne autour des traitements ou des flux de contrôle. Généralement, la notion des données y est occultée. Cette dernière famille contient les langages tels que Manifold [7], qui selon [3], est le plus représentatif et le plus évolué de cette catégorie. Il y a aussi MICADO [10], STL [38], COOL [41], CFL [5], CoLas [18], etc. Et concernant la première classe, Linda est le modèle le plus représentatif, mais il existe aussi d'autres langages et modèles tels que Objective Linda, AproCo [33], etc.

1.2 Présentation de Linda

Le modèle Linda, premier né de la famille des langages et modèles de coordination, propose une manière simple et élégante de séparer les responsabilités entre le calcul et la coordination. En outre, Linda est basé sur la *communication générative* [17]. C'est à dire que si deux processus veulent communiquer, ils n'ont besoin pas d'échanger directement des messages ou de partager une variable, le processus *émetteur* produit un nouvel objet qu'il place sous forme d'un *tuple* dans l'espace partagé, TS, d'où le récepteur pourra le lire ou l'extraire. Le *tuple* ainsi émis est également accessible à tous les processus en présence. Ce paradigme sépare les processus dans l'espace et le temps [3]. Ainsi, aucun processus n'aura besoin de connaître l'identité de l'autre, et il ne sera pas exigé à tous les processus impliqués dans un calcul d'être présents au même moment.

Notre choix de parler plus en détail des langages de type Linda est motivé, au-délà du fait que c'est le modèle de coordination le plus connu, par le fait que le langage $\mathcal{L}_{\Psi}^{r}$, qui fait l'objet du chapitre suivant, appartient à cette catégorie. Et, c'est le langage pour lequel nous allons construire la sémantique opérationnelle au moyen des réseaux de Petri dans la suite de ce travail.

Définition 1.2.1 *Linda est un modèle qui est composé :*

1. *d'une collection de* tuples, *appelé* espace des tuples *ou encore* espace des données partagé, *noté* TS *(comme* Tuple Space*);*
2. *d'un ensemble limité d'opérations, notamment l'ajout, le retrait et la lecture d'un* tuple, *appelées aussi* primitives, *permettant de manipuler les* tuples *contenus dans* TS *;*
3. *d'un mécanisme de matching qui permet aux primitives d'accéder aux contenus des* tuples *présents dans l'espace partagé.*

Définition 1.2.2 *Un* tuple *est une suite finie et ordonnée des champs typés et contenant chacun une valeur typée (de type correspondant au champ associé) ou un processus.*

Définition 1.2.3 *Un tuple qui ne contient que des valeurs typées est appelé un* tuple de données.

Exemple 1.2.1
Voici un tuple de données qui contient trois champs :

$$(\text{“le premier champ est une chaîne de caractères”}, \mathit{true}, 1000)$$

dont le premier est une chaîne de caractères, le deuxième est une valeur booléenne et le troisième une valeur entière.

Définition 1.2.4 *Un tuple qui contient au moins un processus est appelé un* tuple de processus.

Exemple 1.2.2
Voici un exemple de tuple de processus ayant deux champs :

$$(\text{“ceci est un tuple de processus”}, \mathit{sin}(x)).$$

Le premier champ contient une valeur de type chaîne de caractères et le second est un processus (de calcul).

Remarque 1.2.1
Généralement, un tuple de données est considéré comme une entité passive, *tandis qu'un tuple de processus est considéré comme une entité* active *qui échange les données en générant, lisant et consommant des tuples de données [3].*

Définition 1.2.5 *Un* anti-tuple *est une suite finie et ordonnée de champs typés, chacun contenant soit une valeur typée, soit une place libre typée susceptible d'accueillir une valeur du même type.*

Remarque 1.2.2
- *Dans la définition ci-dessus, un champ du premier type est appelé* paramètre valeur *et un champ du deuxième type est dit* paramètre formel.
- *Le mécanisme d'*matching, *dont parle la définition 1.2.1, utilise les paramètres formels pour essayer d'unifier un anti-tuple à un tuple.*
- *D'une façon générale, les opérations de lecture et de retrait imposent une restriction sur les anti-tuples : ils ne peuvent pas contenir de champs de processus.*

Exemple 1.2.3
Voici un anti-tuple

("ceci est un anti-tuple", 1112, ?x)

composé de trois champs dont les deux premiers sont des paramètres valeurs. *D'une façon générale, dans la littérature, on préfixe un* paramètre formel *par un point d'interrogation " ?". C'est le cas du troisième champ ci-dessus.*

Remarque 1.2.3
Les primitives de Linda sont complètement indépendantes de tout langage d'accueil. Il est dit que Linda est orthogonal *à ce langage d'accueil [17], ce qui rend possible l'*intégration *de Linda dans de nombreux langages de programmation. Les exemples de C-Linda, Fortran-Linda, Pascal-Linda, Lisp-Linda, et beaucoup d'autres encore, illustrent cette approche. Il peut être utile de souligner qu'un langage qui* intègre *Linda se comporte exactement comme avant : ce langage prend en charge tout l'aspect* calcul, *qui n'est pas l'affaire de Linda, et ce dernier s'occupe de l'aspect* création et coordination *des processus. Les deux langages agissent donc dans des secteurs différents et leurs sémantiques respectives restent indépendantes.*

1.3 Le fonctionnement du modèle Linda

Rappelons que dans le modèle Linda, les processus utilisent les primitives, appelées aussi *primitives de coordination*, pour :

1. *placer*, ou *insérer*, ou encore *émettre* un tuple dans l'espace des tuples, TS. Cette primitive, généralement notée par `out`(t), avec t le tuple émis, est dite *non-bloquante*, puisqu'un processus n'a besoin d'aucun préalable pour *produire* un tuple à placer dans TS.

   Exemple 1.3.1
   Si un processus p émet

   `out`(*"un tuple à placer dans le TS"*, 130905, *"mémoire"*),

   il place dans le TS ce tuple de trois champs dont deux strings, aux premier et troisième champs, et un entier au deuxième champ.

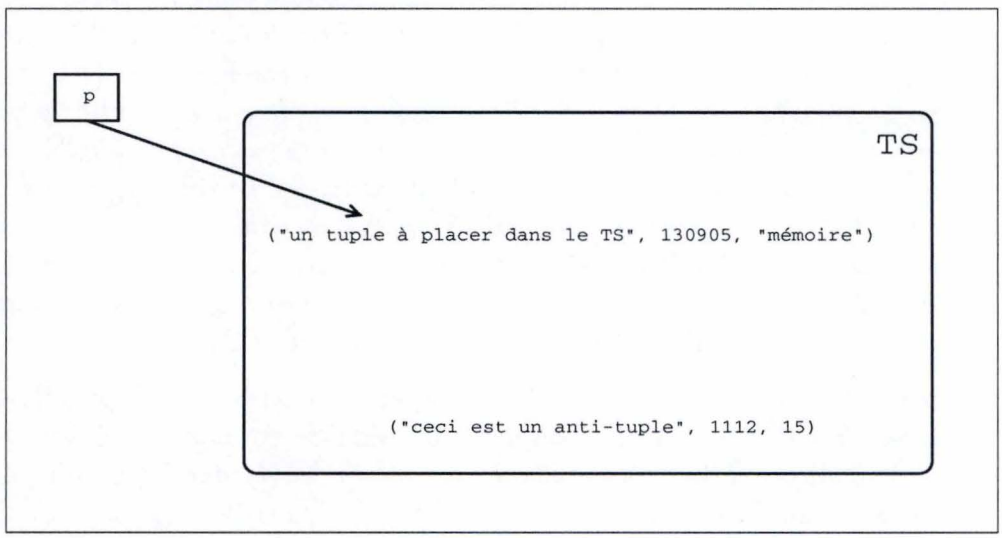

FIG. 1.1 – Le placement d'un tuple dans l'espace des tuples par le processus p.

2. *lire* un tuple dans TS. Cette primitive, notée `rd`(a), avec a un anti-tuple. **Si** il existe un tuple de données t dans l'espace des tuples qui matche avec l'anti-tuple a, **alors** les paramètres valeurs de t sont assignés aux paramètres formels de a et le processus continue son exécution **sinon** le processus appelant est *suspendu.* Dans ce cas, le matching et, par conséquent, l'activation du processus appelant interviennent au

bout d'un temps fini et équitable dès lors que le matching devient possible.

Clairement, on peut bien constater que, contrairement à la première primitive, celle-ci est *bloquante*, puisque la poursuite de l'exécution du processus *lecteur* est conditionnée par l'existence dans le TS d'un tuple qui *matche* l'anti-tuple de départ.

Exemple 1.3.2

Si un processus p exécute la primitive suivante

$$\texttt{rd}(\text{“}\textit{voici une primitive de lecture}\text{”}, ?i, ?j, 1234),$$

une vérification est faite dans l'espace des tuples pour trouver un tuple qui s'unifie avec l'anti-tuple ci-dessus. C'est à dire, cette vérification consistera à trouver un tuple dans TS ayant quatre champs dont le premier et le dernier portent respectivement le string « voici une primitive de lecture »et 1234, et dont les troisième et quatrième champs portent des valeurs de mêmes types que les paramètres formels i et j respectivement. Si un tel tuple est trouvé dans TS, alors le processus p assigne à i et j respectivement les valeurs du troisième et quatrième champs de ce tuple, lit le tuple (sans le supprimer) et poursuit son exécution. Il se peut qu'il y ait plusieurs tuples qui remplissent cette condition, le système en choisit un au hasard parmi eux, assigne à i et j les valeurs que portent respectivement les troisième et quatrième champs dans ce tuple choisi *et poursuit son exécution. S'il n'y a aucun tuple dans TS qui* matche *l'anti-tuple, alors le processus p est suspendu jusqu'à ce qu'il y ait présence d'un tel tuple dans TS ;*

3. *extraire* ou *consommer* un tuple de l'espace des tuples. Cette primitive, notée $\texttt{in}(a)$, avec a un anti-tuple, est *bloquante* comme la précédente. C'est à dire que le processus *consommateur* est *bloqué* tant qu'il n'y a pas un tuple t dans TS qui s'unifie avec l'anti-tuple a. Le fait de *consommer* un tuple t entraîne, naturellement, sa suppression de l'espace des tuples. Le processus poursuit son exécution dès lors qu'un tuple t aura été trouvé et effacé de l'espace des tuples.

Exemple 1.3.3

Si un processus p exécute la primitive suivante

$$\texttt{in}(\text{“}\textit{voici une primitive d'extraction}\text{”}, ?i, ?j, 1234),$$

une vérification exactement comme pour le cas d'une opération de lecture, sauf qu'ici le tuple trouvé sera, en plus, supprimé de l'espace des tuples ;

4. *créer* un processus dans le système. Cette primitive, notée `eval`(t), crée un processus p qui permet d'évaluer le tuple t. Lorsque ce processus est créé, le processus appelant poursuit immédiatement son exécution. Cette primitive est *non-bloquante* comme la première.
   Le processus créé par cette opération s'exécute en parallèle avec tous les autres processus présents dans le système.

Remarque 1.3.1
- *La dernière primitive ne figure pas toujours dans les différentes versions de Linda que l'on peut trouver dans la littérature, généralement pour la complexité qu'elle induit dans la définition de la sémantique du langage.*
- *Il peut être utile de rappeler que l'espace partagé est une collection et non un* ensemble, *au sens mathématique du terme. Ainsi, il peut s'y trouver plusieurs* copies *d'un même tuple t. Par conséquent, si un tuple t est présent en n exemplaires dans TS, par l'effet de la primitive* `in`(t), *il en restera $n-1$.*
- *Dans la littérature, notamment dans [16], il arrive que les primitives de Linda s'appliquent sur les* messages, *qui représentent un concept équivalent aux tuples. Nous allons présenter une notion beaucoup plus englobante dans le chapitre suivant pour définir le type d'éléments sur lesquels s'appliquent les primitives.*

Hormis les principales primitives définies ci-dessus, il y a des versions de Linda, notamment celle présentée dans [16, 24], qui ajoutent deux prédicats conditionnels, lesquels prédicats retournent une valeur booléenne selon l'état courant de l'espace des tuples :
- un prédicat d'*insertion*, noté `inp`(t), qui vérifie l'état courant de TS ; si le tuple t y est présent, le prédicat agit exactement comme la primitive `in`(t) mais retourne la valeur *vrai*, et si le tuple t est absent, la valeur *faux* est retournée ;
- un prédicat de *lecture*, noté `rdp`(t) qui agit exactement comme la primitive `rd`(t) si le tuple est présent dans le TS. Dans ce cas, il retourne la valeur *vrai*, et si le tuple t est absent du TS, le prédicat retourne la valeur *faux*.

Il est clair que ces prédicats ne sont pas *bloquants* puisqu'un résultat (booléen) est retourné quel que soit l'état de l'espace partagé, avec la présence ou l'absence du tuple cherché.
En fait, l'utilisation de ces prédicats fait penser à la structure conditionnelle **if** A **then** B **else** C. Voici une illustration :
soit `inp`(a)$?P\_Q$; c'est à dire le processus extrait du TS un tuple qui s'unifie

à a avant de passer à P ou Q. S'il existe un tel tuple existe, alors le processus l'extrait (le tuple en question est supprimé du TS) et poursuit avec l'exécution de P ; sinon, le processus exécute directement Q. Ceci est rendu possible par le fait que le prédicat `inp` est non-bloquant. On a donc un résultat quel que soit l'état de l'espace des tuples.

1.4 Une autre sémantique pour la primitive d'insertion

Dans les lignes qui ont précédé, nous avons parlé assez rapidement de la primitive de coordination d'insertion, $\texttt{out}(t)$, avec t un tuple ou encore, plus largement, un *message* (selon une autre terminologie dans la littérature notamment dans [15]). Et pourtant, s'il faut donner une sémantique formelle à cette opération d'insertion (non bloquante, du reste), il existe au moins deux manières naturelles de l'interpréter selon [15]. En effet, l'exécution de cette opération peut être scindée en deux phases : une première phase d'*émission* du tuple t, i.e. son envoi vers l'espace des tuples ; et une seconde phase d'*accessibilisation* du tuple t, i.e. sa présence concrète dans l'espace des tuples. Dans [15], N. Busi et Al. proposent, pour ce faire, deux sémantiques :

- une sémantique dite *ordonnée* dans laquelle l'émission et l'accessibilité du tuple forment un tout. Ces deux phases forment donc une action atomique. Ainsi, par exemple, l'agent $\texttt{out}(t);P$, c'est à dire l'opération d'insertion suivie d'une opération P, se transforme en une seule étape interne en l'agent $\langle t \rangle \| P$, où $\|$ est la composition parallèle. C'est à dire que l'on a le tuple t accessible dans le TS et le sous-agent P pouvant s'exécuter parallèlement.
  Dans cette sémantique, l'ordre d'accessibilisation des tuples dans le TS suit leur ordre l'ordre d'émission ;
- une sémantique dite *non-ordonnée* dans laquelle l'émission et l'accessibilisation d'un tuple sont deux actions distinctes. Ainsi, par exemple, le même agent que précédemment $\texttt{out}(t).P$ se transforme dans un premier temps en l'agent $\langle\langle t \rangle\rangle \| P$, c'est à dire le tuple t n'est pas encore accessible dans le TS mais le sous-agent P peut déjà s'exécuter. Dans cette sémantique, il faut une étape interne en plus pour passer de $\langle\langle t \rangle\rangle$ à $\langle t \rangle$. Par conséquent, dans ce cas, l'ordre d'accessibilisation peut ne pas respecter l'ordre d'émission des tuples. C'est à dire que si l'on a, par exemple, un agent $\texttt{out}(t).\texttt{out}(t')$, il peut arriver que le tuple t soit présent dans le TS avant le tuple t', mais il peut aussi arriver que ce soit le contraire.

Par ailleurs, on peut remarquer que l'utilisation des prédicats définis dans la section précédente peut, pour un processus donné, changer radicalement le résultat à obtenir selon que l'on utilise la sémantique ordonnée ou non-ordonnée.
Si l'on considère l'algèbre de processus, L_o, contruite à partir de la sémantique ordonnée, et L_u celle construite à partir de la sémantique non-ordonnée, il est établi, notamment dans [15], que L_o est *Turing complète* tandis que L_u ne l'est pas.

Chapitre 2

Le langage $\mathcal{L}_{\Psi}^{r}$

Le langage $\mathcal{L}_{\Psi}^{r}$, comme nous l'avons dit dans l'introduction, est un langage de coordination de type Linda. Bien que fidèle à la simplicité de Linda, il présente néanmoins certaines particularités intéressantes. Une de ses particularités est le fait qu'il autorise aux tuples de porter une information partielle. Une autre particularité est qu'il possède une primitive qui, à l'opposé de la traditionnelle primitive de lecture des langages de type Linda vérifiant la présence d'un tuple dans l'espace partagé, vérifie l'absence d'un tuple dans le TS. En outre, il est muni, en plus des opérations de composition séquentielle et de composition parallèle, d'une opération de *choix*, selon Milner [35]. Et contrairement à certains langages de même type qui ne proposent qu'une compréhension intuitive de la notion de l'espace partagé, $\mathcal{L}_{\Psi}^{r}$ définit de manière rigoureuse l'espace des tuples ainsi que les autres concepts associés.

Le présent chapitre est essentiellement construit sur base de [13].

2.1 Présentation de $\mathcal{L}_{\Psi}^{r}$

Le langage $\mathcal{L}_{\Psi}^{r}$ propose, dans la lignée des langages de type Linda, des primitives d'insertion, d'extraction et de lecture qui seront notées respectivement `tell`, `get` et `ask` au lieu de `out`, `in` et `rd` respectivement.
Une particularité de $\mathcal{L}_{\Psi}^{r}$ est de permettre aux tuples de porter une information partielle. Concrètement, cela s'exprime par le fait qu'il autorise la *recherche*, dans l'espace partagé, des tuples dont on ne détient que l'information dans quelques champs. Rappelons-nous, dans le chapitre précédent, avec les primitives de lecture et d'extraction, il était possible de chercher un tuple qui matche l'anti-tuple de départ, mais il était nécessaire de préciser, dans le bon ordre, tous les champs de l'anti-tuple, soit avec des paramètres

valeurs, soit avec des paramètres formels. Le langage $\mathcal{L}^r_\Psi$ s'affranchit, quant à lui, de cette contrainte. Il est possible, avec $\mathcal{L}^r_\Psi$, non seulement de ne donner que l'information concernant quelques champs, mais aussi de présenter ces champs dans n'importe quel ordre, à condition de bien les identifier. Pour réaliser ces nouveautés, $\mathcal{L}^r_\Psi$ introduit la notion des Ψ-termes.

Définition 2.1.1 *Soit Scvar un ensemble dénombrablement infini des variables de communication et Sf un ensemble dénombrablement infini des noms de foncteurs, chacun associé à une arité. Une association de ce type est notée f/n, où f est le nom du foncteur et n son arité. Nous supposerons que les ensembles Scvar et Sf sont disjoints. Les foncteurs d'arité 0 sont appelés des* constantes, *et leur ensemble est noté Sconst.*

Définition 2.1.2 *Un Ψ-terme est une expression de la forme $f(item_1 = val_1, \cdots, item_m = val_m)$ où*
- *f/n est un foncteur tel que $m \leq n$*
- *les $item_i$ sont des constantes distinctes les unes des autres*
- *$\forall i \in \{1, \cdots, m\}$, val_i est un élément de l'ensemble fini Sval, un Ψ-terme ou une variable de communication*
- *toute variable de communication figure tout au plus dans un Ψ-terme.*

L'ensemble des Ψ-termes est noté par Spterm.

Définition 2.1.3 *Un Ψ-terme est dit* clos *s'il ne contient aucune variable de communication. L'ensemble des Ψ-termes clos est noté par Scpterm.*

Dans l'introduction de cette section, nous avons expliqué une des particularités du langage $\mathcal{L}^r_\Psi$, celle qui permet de retrouver un tuple à partir de l'information contenue dans certains champs seulement. Dans les lignes qui suivent, nous définissons la notion qui rend possible ce mécanisme.

Définition 2.1.4 *Un Ψ-terme $f(t_1 = v_1, \cdots, t_n = v_n)$ est appelé une* restriction *du Ψ-terme $f(t'_1 = v'_1, \cdots, t'_m = v'_m)$ par rapport à l'ensemble d'items I si et seulement si*

$$\{t_j = v_j : 1 \leq j \leq n\} = \{t'_i = v'_i : 1 \leq i \leq m \wedge t'_i \in I\}.$$

Définition 2.1.5 *Soit $\Psi_1 = f(t_1 = v_1, \cdots, t_n = v_n)$ et $\Psi_2 = f'(t'_1 = v'_1, \cdots, t'_m = v'_m)$ deux Ψ-termes. Nous disons que Ψ_1* correspond *à Ψ_2 ssi*
- *f and f' sont deux foncteurs identiques ayant la même arité ;*
- *$\{t_i : 1 \leq i \leq n\} \subseteq \{t'_j : 1 \leq j \leq m\}$;*
- *pour tout i tel que v_i est un entier ou une chaîne de caractèress, si $t_i = t'_j$ alors $v_i = v'_j$;*

- *pour tout i tel que v_i est un Ψ-terme, si $t_i = t'_j$ alors v_i correspond to v'_j.*

Remarque 2.1.1
Lorsque Ψ_2 est clos, la correspondance *de Ψ_1 par rapport à Ψ_2 revient à l'existence d'un ensemble des valeurs pour les variables de communication de Ψ_1 tel que ce dernier devient identique à une certaine restriction de Ψ_2. Dans ce cas, cette propiété est notée par $\Psi_1 \lhd \Psi_2$ et l'*association *des valeurs aux variables de Ψ_1 dont l'application transforme en une restriction de Ψ_2 est notée par $\theta = bind(\Psi_1 \lhd \Psi_2)$.*

Définition 2.1.6 *Nous noterons par Sbind l'ensemble de toutes les associations partielles possibles des valeurs aux variables, i.e. l'ensemble de toutes les fonctions θ de la forme*

$$\theta : Scvar \rightarrow (Sval \cup Scpterm \cup \{\perp\})$$

où $\perp$ représente une valeur spéciale qui exprime une valeur indéfinie. Par abus de notation, nous noterons par $\perp$ l'association qui laisse toutes les variables indéfinies.

Dans les lignes qui suivent, nous définissons respectivement une relation d'ordre partiel et une composition dans l'ensemble *Sbind*.

Définition 2.1.7 *Pour tout couple (θ, φ) d'éléments de Sbind, nous dirons que $\theta \prec \varphi$ ssi quelle que soit la variable x, on a $\theta(x) = \perp$ chaque fois que $\varphi(x) = \perp$.*

Définition 2.1.8 *Pour tout couple (θ, φ) d'éléments de Sbind, nous définirons par $\theta\varphi$ la fonction suivante :*

$$Scvar \rightarrow (Sval \cup Scpterm \cup \{\perp\})$$

telle que pour toute variable x,

$$\theta\varphi(x) = \begin{cases} \varphi(x) & si\ \varphi(x) \neq \perp \\ \theta(x) & si\ \varphi(x) = \perp \end{cases}$$

A l'instar des autres langages de coordination, le langage $\mathcal{L}^r_\Psi$ que nous présentons dans cette section, s'occupe de l'aspect *interaction et communication* entre les processus. Il appartient aux langages d'accueil de spécifier l'aspect *calcul*. Les variables de communication jouent justement le rôle d'interface entre ces deux aspects : interactions/communication et calcul. En effet, les

variables de communication sont aussi bien utilisées par les primitives de coordination que par les instructions du langage d'accueil qui s'occupe de l'aspect calcul. Ainsi, nous noterons, dans la suite de ce chapitre, par *Sinstr* l'ensemble des instructions du langage d'accueil. Ce langage pouvant être de n'importe quel paradigme avec comme seule contrainte le fait qu'il ne peut pas intéragir directement avec l'espace partagé et ne peut pas, non plus, modifier les valeurs assignées aux variables.
Par ailleurs, la récursion est aussi modélisée dans $\mathcal{L}^r_\Psi$, et nous noterons par *Srvar* l'ensemble des variables de récursion que nous allons supposer disjoint de tous les ensembles définis précédemment.

Définition 2.1.9 *Soit Ψ un Ψ-terme, I une instruction de Sinstr et X une variable de récursion de Srvar :*

1. *Nous appellerons* actions de communication *les primitives c suivantes :*

$$c ::= \texttt{tell}(\psi) \mid \texttt{nask}(\psi) \mid \texttt{ask}(\psi) \mid \texttt{get}(\psi)$$

2. *Le langage $\mathcal{L}^r_\Psi$ est l'ensemble des agents A définis par la grammaire suivante où l'agent composé par l'opérateur $||$ utilise des ensembles de variables de communication disjoints et où l'ensemble $\mathcal{V}$ indexant la variable de récursion X définit l'ensemble des variables de coordination que cet agent peut partager avec d'autres, et $AcVar$ l'ensemble des variables de communication de l'agent A :*

$$A ::= c \mid A\,;\, A \mid A \,||\, A \mid A + A \mid I \mid X_\mathcal{V}$$

3. *L'ensemble* étendu *des agents, Seagent, est l'ensemble des agents Ae définis par la grammaire suivante :*

$$Ae ::= E \mid c \mid A\,;\, A \mid A \,||\, A \mid A + A \mid I \mid X_\mathcal{V}$$

4. *L'ensemble des agents* gardés, *noté $\mathcal{G}$, est l'ensemble des agents g définis comme suit :*

$$g ::= c \mid I \mid g; A \mid g_1 \,||\, g_2 \mid g_1 + g_2$$

Rappelons que les actions de communication définies plus haut expriment les opérations élémentaires du langage Linda notamment la primitive permettant d'insérer une information dans le système, `tell` ; celle qui vérifie la présence d'une information, `ask` ; celle qui vérifie l'absence d'une information, `nask` ; et la dernière qui permet de retirer une information, `get`. En

outre, les trois constructeurs ; , || et + définissent respectivement la composition séquentielle, la composition parallèle et la composition alternative. Par ailleurs, pour une utilisation ultérieure plus efficace, nous introduisons un agent spécial pour exprimer la terminaison d'une exécution quelconque. Nous allons noter par E le symbole qui exprime cette terminaison. En utilisant E, nous étendons l'ensemble des agents qui devient l'ensemble *Seagent* ci-dessus. Remarquons qu'avec cette extension, les agents suivants $(E\ ;\ A)$, $(E \mid\mid A)$, et $(A \mid\mid E)$ représentent tous un seul et même agent qui est A. Ceci est rendu possible par le fait que l'on définit la structure $(Seagent, \ ;\ , \mid\mid , E)$ comme un *bimonoïde*[1] dans lequel E est un *élément neutre* pour les deux lois «; »et « || ».

Dans le but de garder l'essence des variables de communication à travers l'espace des tuples et d'éviter une communication latérale entre les agents, il est exigé que deux agents qui sont composés parallèlement ne partagent aucune variable de communication en commun. Cela s'exprime dans la propriété suivante.

Propriété 2.1.1 *Lorsque deux agents sont composés parallèlement, les ensembles des variables de communication figurant dans les primitives ou dans les ensembles indexant les variables récursives apparaissant dans chacun de deux agents respectivement doivent être disjoints.*

Enfin, l'utilisation de l'ensemble $\mathcal{V}$ des variables de coordination avec une variable de récursion X est un moyen de modéliser la portée. Les variables de $\mathcal{V}$ peuvent être considérées comme des variables *globales* utilisées par la procédure X.

Définition 2.1.10 *Une* déclaration D *est une liste d'associations* $<X, g_X>$ *entre les variables de récursion et les agents gardés. Cette liste peut être infinie. Nous exigeons cependant que l'ensemble des foncteurs qui apparaissent dans les agents gardés soit un sous-ensemble strict de l'ensemble des tous les foncteurs* Sf .

Dans les lignes qui suivent, nous supposons qu'une déclaration D est donnée, ainsi allons-nous omettre de la mentionner dès lors qu'il n'y aurait pas de confusion.

[1]Un bimonoïde $M = (M, \star, \otimes, 1)$ est une structure où $(M, \star, 1)$ et $(M, \otimes, 1)$ sont respectivement un monoïde et un monoïde abélien

2.2 Sémantique opérationnelle de $\mathcal{L}^r_\Psi$

Dans cette section, la sémantique opérationnelle de $\mathcal{L}^r_\Psi$ est décrite sous forme des règles de transition présentées dans le style de Plotkin, i.e. en l'exprimant par un système de réécriture conditionnel. Mais avant cela, nous allons spécifier quelques notions qui vont nous aider à définir ce système de transitions.

2.2.1 Les configurations

Afin de définir de manière plus complète un agent, il est important de tenir compte du contenu de l'espace partagé et des différentes associations des variables de communication. Cette vue d'ensemble est définie comme une *configuration.* Intuitivement, elle peut être comprise comme une *vue instantanée* du système, montrant un agent ensemble avec tout son environnement. Formellement, elle est définie de la manière suivante.

Définition 2.2.1 *Nous définissons l'*espace de stockage *Sstore comme l'ensemble des* multi-ensembles *finis des Ψ-termes clos. Et, nous définissons l'ensemble des* situations, *Ssit , comme l'ensemble Sbind $\times$ Sstore.*

Définition 2.2.2 *L'ensemble des configurations Sconf est défini comme Seagent $\times$ Ssit. Une configuration est donnée par $\langle A \mid (\theta, \sigma)\rangle$, où A est un agent (étendu) et (θ, σ) est une situation.*

Avec ces définitions, nous pouvons maintenant donner les règles de transition de la sémantique opérationnelle du langage $\mathcal{L}^r_\Psi$.

2.2.2 Les règles de transition

Les règles de transition qui définissent la sémantique opérationnelle du langage $\mathcal{L}^r_\Psi$ sont données dans la figure 2.1.

La règle **(T)** déclare que si (θ, σ) est une situation telle que l'association courante des variables de communication, θ, transforme un Ψ-terme donné Ψ en un Ψ-terme clos, alors l'agent atomique $\texttt{tell}(\Psi)$ peut être exécuté. Et dans ce cas, nous remarquons que cette exécution permet à l'espace de stockage σ d'être enrichi du nouveau Ψ-terme clos $\Psi\theta$. C'est à dire, il passe de σ à $\sigma \cup \{\Psi\theta\}$.

La règle **(A)** déclare qu'un agent atomique $\texttt{ask}(\Psi)$ peut être exécuté dans une situation à condition l'espace de partagé σ contienne un Ψ-term Ψ_c tel

$$\textbf{(T)} \quad \frac{\Psi\theta \text{ est clos}}{\langle \texttt{tell}(\Psi) \mid (\theta, \sigma)\rangle \longrightarrow \langle E \mid (\theta, \sigma \cup \{\Psi\theta\})\rangle}$$

$$\textbf{(A)} \quad \frac{\mu = bind(\Psi \triangleleft \Psi_c)}{\langle \texttt{ask}(\Psi) \mid (\theta, \sigma \cup \{\Psi_c\})\rangle \longrightarrow \langle E \mid (\theta\mu, \sigma \cup \{\Psi_c\})\rangle}$$

$$\textbf{(N)} \quad \frac{\nexists \Psi_c : \Psi_c \in \sigma, \Psi \triangleleft \Psi_c}{\langle \texttt{nask}(\Psi) \mid (\theta, \sigma)\rangle \longrightarrow \langle E \mid (\theta, \sigma)\rangle}$$

$$\textbf{(G)} \quad \frac{\mu = bind(\Psi \triangleleft \Psi_c)}{\langle \texttt{get}(\Psi) \mid (\theta, \sigma \cup \{\Psi_c\})\rangle \longrightarrow \langle E \mid (\theta\mu, \sigma)\rangle}$$

$$\textbf{(I)} \quad \frac{executable(I, \theta)}{\langle I \mid (\theta, \sigma)\rangle \longrightarrow \langle E \mid (\theta, \sigma)\rangle}$$

$$\textbf{(S)} \quad \frac{\langle A \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle}{\langle A \ ; \ B \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \ ; \ B \mid (\theta', \sigma')\rangle}$$

$$\textbf{(P)} \quad \frac{\langle A \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle}{\begin{array}{l}\langle A \parallel B \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \parallel B \mid (\theta', \sigma')\rangle \\ \langle B \parallel A \mid (\theta, \sigma)\rangle \longrightarrow \langle B \parallel A' \mid (\theta', \sigma')\rangle\end{array}}$$

$$\textbf{(C)} \quad \frac{\langle A \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle}{\begin{array}{l}\langle A \ + \ B \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle \\ \langle B \ + \ A \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle\end{array}}$$

$$\textbf{(R)} \quad \frac{\langle ren(g_X, \mathcal{V}) \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle \wedge \ < X, g_X > \in D}{\langle X_\mathcal{V} \mid (\theta, \sigma)\rangle \longrightarrow \langle A' \mid (\theta', \sigma')\rangle}$$

FIG. 2.1 – Les règles de transition.

que Ψ *correspond* à Ψ_c. Dans ce cas, des valeurs peuvent être calculées pour les variables de communication entrainant ainsi une mise à jour qui change l'association θ en $\theta\mu$. Quant à l'espace de stockage, il ne change pas.

La règle **(G)** est similaire à la précédente sauf que le Ψ-terme est supprimé de l'espace de stockage qui redevient σ.

La règle **(N)** est le dual de **(A)** : elle réussit justement s'il n'existe pas dans l'espace de stokage un Ψ-terme Ψ_c dont Ψ soit une correspondance. Et, dans ce cas, l'exécution garde inchangés l'espace de stockage et les différentes associations.

Pour la règle **(I)**, il peut être intéressant de rappeler que les instructions de *Sinstr* peuvent consulter et utiliser les valeurs de variables de communication. Leur exécution peut donc dépendre de ces valeurs. Il se peut aussi qu'il n'y ait pas d'exécution d'instruction. Le prédicat *executable* est introduit pour modéliser cette réalité. Pour une instruction $I \in Sinstr$ et une association $\theta \in Sbind$, $executable(I, \theta)$ est vrai ssi I peut être exécutée sur θ. Et dans ce cas, l'espace de stockage et les associations restent intacts. Rappelons que du point de vue des langages de coordination, l'exécution de l'instruction I ne peut dépendre du contenu de l'espace partagé puisqu'elle n'y a pas accès. En plus, le résultat de l'exécution ne peut pas modifier le contenu de l'espace partagé et des associations exprimées dans θ.

Les règles **(S)**, **(P)**, et **(C)** décrivent de manière *classique* la signification opérationnelle des opérateurs séquentiel, parallèle et alternatif respectivement.

Et pour finir, la règle **(R)** définit le calcul d'une variable récursive $X_\mathcal{V}$ ainsi que celui de l'agent associé g_X. Et on utilise $ren(g_X, \mathcal{V})$ pour exprimer le *renommage* des variables locales à la procédure X, i.e. des variables de l'agent g_X qui n'apparaissent pas dans $\mathcal{V}$, étant donné que les éléments de celui-ci sont considérés comme des variables globales utilisées par X.

Chapitre 3

Les réseaux de Petri contextuels

Ce chapitre donne une rapide et succincte introduction aux réseaux de Petri. Les réseaux de Petri constituent probablement un des meilleurs formalismes pour modéliser le comportement dynamique des systèmes discrets faisant intervenir des événements concourants ou parallèles. Par conséquent, ils offrent quelques avantages dans la présentation de certaines caractéristiques propres aux systèmes concurrents et distribués. Voilà une des motivations pour lesquelles nous avons choisi d'exprimer la sémantique du langage $\mathcal{L}_{\Psi}^{r}$ au moyen de ce formalisme. En outre, il est un fait que les réseaux de Petri bénéficient d'une abondante documentation et des outils permettant diverses analyses, notamment le *model checking*.
Dans cette forêt que constitue la documentation sur les réseaux de Petri, nous avons choisi une présentation qui se rapproche le plus à l'utilisation que nous allons en faire dans les chapitres suivants, à savoir la présentation qui recourt aux *multi-ensembles*.

3.1 Brève présentation des réseaux de Petri

Avant de parler des réseaux de Petri, nous faisons un petit rappel sur les *multi-ensembles* (multisets, en anglais).

3.1.1 Rappel sur les multi-ensembles

Les multi-ensembles peuvent être considérés comme des ensembles dans lesquels les multiplicités des éléments sont prises en compte. Certains éléments peuvent aussi avoir une multiplicité infinie, ainsi nous étendons l'ensemble des naturels, $\mathbb{N}$, en lui adjoignant l'élément ∞. Cet nouvel ensemble, $\mathbb{N} \cup \{\infty\}$, sera noté $\mathbb{N}^{\infty}$. En étendant l'ensemble des naturels, nous y étendons

aussi l'addition et la soustraction usuelles :
- $\forall n \in \mathbb{N}^\infty, n + \infty = \infty + n = \infty$
- $\forall n \in \mathbb{N}, \infty - n = \infty$. Pour des raisons évidentes, nous allons éviter $\infty - \infty$.

De la même façon, nous étendons aussi la relation d'ordre $\leq$ sur $\mathbb{N}^\infty$ en posant $n \leq \infty, \forall n \in \mathbb{N}^\infty$.

Définition 3.1.1 *Un ∞-multi-ensemble sur un ensemble X est une fonction f définie comme suit :*

$$f : X \to \mathbb{N}^\infty$$

associant à chaque $x \in X$ un nombre non négatif ou ∞, qui est la multiplicité *de x. Souvent la multiplicité de x est notée f_x au lieu de $f(x)$.*

Définition 3.1.2 *Nous noterons par $\mathcal{M}_\infty(X)$ l'ensemble des tous les ∞-multi-ensembles sur X. Quelques fois nous considérerons un ∞-multi-ensemble comme un vecteur de l'espace $\mathcal{M}_\infty(X)$ dont les composants, appelés aussi* entrées, *sont indexés par les éléments de X.*

Définition 3.1.3 *Nous appelerons* multi-ensembles finis *les ∞-multi-ensembles dont aucune entrée ne vaut ∞. Et nous noterons par $\mathcal{M}_{fin}(X)$ l'ensemble des multi-ensembles finis sur X.*
En particulier, le multi-ensemble nul *sur X est la fonction $x \mapsto 0$ quel que soit $x \in X$. Et un multi-ensemble fini autre que le multi-ensemble nul sera simplement appelé un multi-ensemble, leur ensemble sera noté $\mathcal{M}(X)$.*

Remarque 3.1.1
D'une façon générale, nous confondrons assez souvent les multi-ensembles finis et les multi-ensembles. Désormais, nous écrirons $dom(f)$ pour désigner l'ensemble $\{x \in X \mid f(x) \neq 0\}$.

On peut définir quelques opérations et relations par les ∞-multi-ensembles. Elles sont induites par les mêmes opérations et relations sur les entiers. S'il apparaît clairement que les ∞-multi-ensembles sont fermés pour l'addition, ils ne le sont, par contre, pas pour la soustraction.

Définition 3.1.4 *Soient $f, g \in \mathcal{M}_\infty(X)$ et $h \in \mathcal{M}(X)$:*
- $f \subseteq g \Leftrightarrow \forall x \in X. f_x \leq g_x$
- $(f \oplus g)_x = f_x + g_x, \forall x \in X$
- *si $h \leq f$, alors $f \ominus h$ est un multi-ensemble avec*

$$(f \ominus h)_x = f_x - h_x, \forall x \in X.$$

Il y a des sous-ensembles de $\mathcal{M}(X)$ dont nous allons parler plus loin. Il s'agit notamment du sous-ensemble composé des $f \in \mathcal{M}(X)$ tels que $f_x \leq 1$ quel que soit x dans X.

3.1.2 Une vue générale sur les réseaux de Petri

Un réseau de Petri peut, intuitivement, être considéré comme un système de transitions où l'occurrence d'un événement n'affecte que les conditions de son voisinage immédiat. La réalisation de cette transition ne dépend pas du tout d'un état global du système. Plus formellement, nous définissons le réseau de Petri de la manière suivante.

Définition 3.1.5 *Un réseau de Petri est un triplet (S, T, m_0) où*
- *S est l'ensemble de* places, *appelées aussi* conditions
- *$T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S)$ est l'ensemble de* transitions, *appelées aussi* événements
- *m_0 est un multi-ensemble fini sur l'ensemble de places S.*

Tout multi-ensemble fini sur S est aussi appelé un marquage. *Et m_0 est appelé est* marquage initial.

Définition 3.1.6 *Soient un marquage m et une place s, nous disons que s contient $m(s)$* jetons *(*tokens, *en anglais). Autrement dit, la multiplicité d'une place s par rapport au multi-ensemble ou marquage m donne le nombre de jetons présents à cette place.*

Définition 3.1.7 *Un réseau de Petri est* fini *si les ensembles S et T sont tous deux finis.*

Définition 3.1.8 *Soit $t = (c, p)$ une transition, notée aussi $c \longrightarrow p$, le marquage c, noté aussi ${}^\bullet t$, est appelé le* pré-ensemble *de t ; le marquage p, noté aussi $t^\bullet$, est appelé le* post-ensemble *de t.*

Définition 3.1.9 *Une transition t est dite* franchissable *pour un marquage m si ${}^\bullet t \leq m$. On dit aussi t a une* concession *pour le marquage m. Cette condition est appelée* précondition de franchissement *ou* précondition de tir *de t. L'exécution d'une transition t franchissable pour un marquage m produit un marquage $m' = (m \ominus {}^\bullet t) \oplus t^\bullet$. Ceci peut aussi être noté $m \xrightarrow{t} m'$ ou encore $m \longrightarrow m'$ s'il n'est pas utile de préciser la transition.*

Remarque 3.1.2

1. *En fait, pour une transition t, ${}^\bullet t$ représente les jetons qui vont être* consommés *lorsque t est* tiré *; tandis que $t^\bullet$ représente les jetons produits lors du tir de t.*
2. *Il arrive que l'on impose une capacité pour chaque place d'un réseau, c'est à dire le nombre maximal de jetons que peut contenir une place. Mais dans le cas qui nous concerne dans ce travail, nous supposons que les places ont une capacité illimitée.*
3. *Nous avons parlé plus haut des multi-ensembles sur l'ensemble X dont la multiplicité est inférieure ou égale à 1 en tout point $x \in X$. On peut constater que le sous-ensemble de $\mathcal{M}(X)$ qui contient ce type d'éléments peut être confondu avec $\mathcal{P}(X)$, l'ensemble des parties de X. En effet, pour un multi-ensemble f tel que $f(x) \leq 1, \forall x \in X$, en considérant $dom(f)$ en lieu et place de f, on ne perd aucune information. Puisque $dom(f)$ ne contient que les éléments de X ayant une multiplicité non nulle. Ainsi, au lieu de considérer un sous-ensemble de $\mathcal{M}(X)$, donc un ensemble de multi-ensembles, nous considérons plutôt $\mathcal{P}(X)$. Généralement, nous écrirons $\mathcal{P}_{fin}(X)$ pour désigner l'ensemble des toutes les parties finies de X.*

La littérature sur les réseaux de Petri s'étale sur une multitude des propriétés de ces derniers. Ce travail n'ayant pas l'ambition de faire une étude en profondeur sur les réseaux de Petri, nous allons nous contenter d'en énumérer quelques unes qui, d'une façon ou d'une autre, vont nous être utiles dans les chapitres qui vont suivre. Nous pouvons cependant dire que, d'une manière générale, ces propriétés se déclinent selon trois axes principaux qui permettent d'analyser les réseaux :

- le premier axe est basé sur la notion d'*arborescence et de graphe de couverture*. Dans cet axe, il est mis en avant la construction des marquages accessibles à partir du marquage initial ;
- le deuxième axe est celui de l'*algèbre linéaire* qui permet l'étude des propriétés dites *structurelles* ;
- le troisième axe est celui de la *théorie des graphes* qui, lui, permet de décider sur certaines propriétés des réseaux de Petri et d'analyser certaines classes particulières de réseaux.

Nous avons pu trouver une synthèse intéressante sur les propriétés des réseaux de Petri notamment dans [21, 32].

Définition 3.1.10 *Si $\sigma = t_1, ..., t_n$ est une séquence de transitions, nous noterons par $m \xrightarrow{\sigma} m'$ pour signifier $m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$. Nous disons alors*

que le marquage m' est accessible *à partir du marquage m. Et si m et m' sont deux marquages, nous dirons que m' est accessible à partir de m s'il existe une séquence de transitions σ telle que $m \xrightarrow{\sigma} m'$.*

Définition 3.1.11 *Soit $N = (S, T, m_0)$ un réseau de Petri. Une place s est dite* k-bornée *si quel que soit le marquage m accessible à partir de m_0, la multiplicité de s, $m(s) \leq k$, avec $k < \infty$. Un réseau est dit* borné *si toutes ces places sont bornées. Il est* k-borné *si toutes ses places sont k-bornées. Une place* 1-bornée *est aussi appelée une place* binaire.

Définition 3.1.12 *Un marquage m est dit* mort *s'il n'existe aucun autre marquage qui soit accessible à partir de m. On écrit alors $m \not\rightarrow$.*

Définition 3.1.13 *Soit $N = (S, T, m_0)$ un réseau de Petri. N a un* blocage *ou une* impasse *(deadlock, en anglais) s'il existe un marquage mort accessible à partir du marquage initial m_0.*

Définition 3.1.14 *Soient $N = (S, T, m_0)$ et une transition $t \in T$. Nous dirons que t est* vivante *si pour chaque marquage m accessible à partir du marquage initial m_0, il existe un marquage m' accessible à partir de m tel que t est franchissable pour m'. N est* vivant *si chacune de ses transitions est vivante.*

Théorème 3.1.1 *Le problème d'existence d'un blocage (deadlock) dans un réseau de Petri peut être réduit à un problème de vivacité.*

Preuve : une démonstration élégante peut être trouvée dans [16].

Il serait intéressant de dire un mot sur le bienfondé de ce théorème. En effet, le problème d'existence d'un blocage pour les réseaux de Petri finis est décidable. La démonstration consiste justement à transformer ce problème de blocage à un problème de vivacité. Or il a été établi, notamment dans [37], que ce dernier problème est décidable, ainsi en réduisant le problème de blocage à un problème de vivacité, on démontre que le problème de blocage est aussi décidable. Le précédent théorème apporte une petite généralisation puisqu'il manipule des multi-ensembles et non des simples ensembles comme c'est le cas dans la référence susmentionnée.

3.2 Définition des réseaux de Petri contextuels

Les primitives de coordination utilisées dans certains langages de coordination, dont $\mathcal{L}^r_\Psi$, peuvent demander une certaine adaptation des réseaux

de Petri dans le but d'obtenir une plus grande lisibilité des primitives de ces langages en termes de réseaux de Petri. C'est principalement pour cette raison que l'on fait intervenir, dans la définition des réseaux de Petri dits *contextuels*, à part l'arc *normal*, deux autres types d'arcs : l'arc *contextuel* et l'arc *inhibiteur*.

Si l'arc *normal* est celui que nous trouvons habituellement dans les réseaux de Petri, et qui relie une place à une transition et vice-versa, en indiquant toujours le sens, l'arc *contextuel*, lui, relie également une place et une transition mais ne dispose pas de flèche pour indiquer un quelconque sens. Quant à l'arc *inhibiteur*, il relie une place à une transition mais avec la particularité d'avoir un petit cercle du côté où il touche la transition. La figure 3.1 montre les trois types d'arcs.

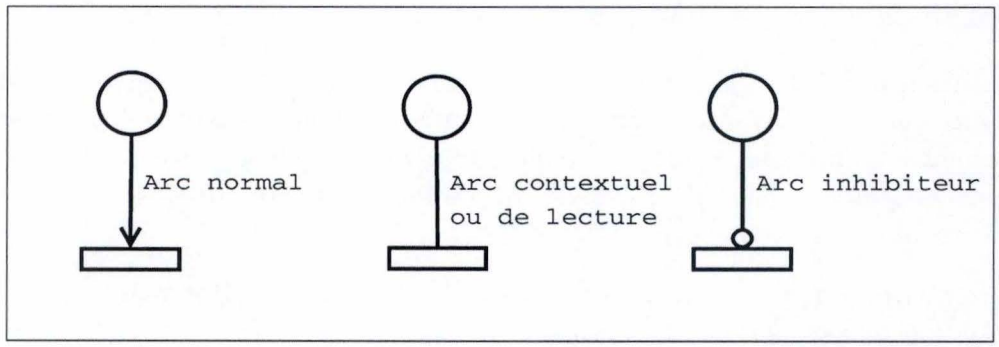

FIG. 3.1 – Les trois types d'arcs utilisés dans un réseau contextuel.

La signification d'un arc normal, par la présence de la flèche, est la plus intuitive : la flèche indique le sens du flux. Généralement quand l'arc ne porte pas une étiquette pour indiquer le nombre de tokens qu'il transmet, on suppose que ce nombre vaut 1. C'est à dire que la transition exige, pour son franchissement, la présence d'un token dans la place à laquelle elle est reliée en amont avec cet arc ; ou la transition produit un token pour la place après son franchissement si cette dernière se trouve en aval. Pour l'arc contextuel, l'absence de flèche signifie simplement que l'échange entre la transition et la place se fait dans les deux sens. Cela veut dire qu'un arc contextuel peut être remplacé par deux arcs normaux de sens contraires. En fin de compte, l'arc contextuel peut être considéré comme une facilité de représentation introduite pour avoir des réseaux moins encombrés. Pour l'arc inhibiteur, les choses se présentent sous un autre aspect. En fait, il est impossible de tester, dans un réseau de Petri traditionnel, si le contenu d'une place est vide.

S'il arrive qu'une transition est franchissable lorsqu'une place en entrée ne contient pas de jeton, il n'existera simplement pas d'arc entre la transition et la place en question. Or la réalité des langages de coordination fait que l'on peut tester l'absence d'un tuple dans l'espace partagé, et l'arc inhibiteur permet justement de transposer cette réalité en vérifiant l'absence d'un jeton dans une place donnée. Nous y reviendrons plus tard. Il est clair que l'introduction de l'arc inhibiteur ajoute plus d'expressivité pour les primitives de coordination en termes de réseaux de Petri. Les définitions ci-dessous sont essentiellement tirées de [16].

Définition 3.2.1 *Un réseau de Petri contextuel est un triplet* $N = (S, T, m_0)$ *où*

- S *est l'ensemble des places ;*
- $T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{P}_{fin}(S) \times \mathcal{P}_{fin}(S) \times \mathcal{M}_{fin}(S)$ *est l'ensemble des transitions, et*
- m_0 *le marquage initial.*

Définition 3.2.2 *Une transition* $t = (c, r, i, p)$ *sera aussi représentée par* $r, \not{t} : c \longrightarrow p$ *où* r *et* $\not{t}$ *seront omis s'ils sont vides. L'ensemble* r*, noté aussi* $\hat{t}$*, est appelé l'*ensemble contextuel *de* t *et représente les jetons dont on vérifie la présence pour tirer* t*. Et l'ensemble* i*, noté aussi* $^\circ t$*, est appelé l'*ensemble inhibiteur *de* t *et représente les jetons dont on vérifie l'absence en vue du franchissement de* t*. Les marquages* c *et* p *gardent la même signification que dans la définition 3.1.8.*

Définition 3.2.3 *Soient* $N = (S, T, m_0)$ *un réseau de Petri contextuel et une transition* $t \in T$*. Nous disons que* t *est franchissable pour un marquage* m *donné si* $^\bullet t \oplus \hat{t} \subseteq m$ *et* $dom(m) \cap {}^\circ t = \emptyset$*.*

3.3 Existence d'un réseau de Petri simple au comportement équivalent à un réseau de Petri contextuel

La plupart des outils qui existent sur les réseaux de Petri manipulent des réseaux de Petri simples. Et même pour ceux qui manipulent indistinctement les réseaux simples et les réseaux colorés, la stratégie revient souvent à transformer (dans un processus interne à l'outil ou pas) ces derniers en réseaux simples et, seulement, d'entreprendre ensuite les analyses voulues. Rappelons que nous voulons exprimer la sémantique du langage $\mathcal{L}^r_\Psi$ au moyen des réseaux de Petri contextuels, puisqu'ils peuvent exprimer le comportement

des agents des langages de coordination avec plus d'efficacité et plus de lisibilité. C'est l'objet du prochain chapitre. Comme bien d'autres outils sur les réseaux de Petri, INA, l'outil dont nous parlerons au chapitre 5, manipule les réseaux simples et les réseaux colorés. L'idée est donc de transformer nos réseaux contextuels en réseaux simples avant de les soumettre aux différentes analyses.

Cette transformation a un sens seulement si elle préserve le comportement du réseau contextuel dans le nouveau réseau. Une telle démarche a été proposée dans [16]. Nous devons nous rappeler que c'est la présence des arcs contextuels et des arcs inhibiteurs qui font qu'un réseau soit contextuel. L'idée générale qui sous-tend démarche est de transformer lesdits arcs en arcs normaux tout en préservant le comportement général du réseau.

Il est clair que la transformation d'un réseau en un autre susceptible d'avoir plus de places et/ou de transitions peut affecter certaines propriétés du réseau de départ. Ainsi dans la démarche proposée dans [16], les auteurs privilègient deux caractéristiques majeures : les séquences de franchissement et l'existence de blocages (deadlocks). En effet, dans cette démarche toutes les séquences de franchissement du réseau contextuel sont préservées dans le réseau simple obtenu après transformation. De même, tous les blocages éventuels dans l'ancien réseau sont préservés dans le nouveau, avec au moins les mêmes places et transitions.

Dans cette démarche, il est aisé de faire disparaître les arcs contextuels. Il suffit, en effet, de remplacer chacun d'eux, comme nous l'avons déjà dit plus haut, par deux arcs normaux ayant des sens opposés reliant même place et même transition. En fait, tester la présence d'un jeton dans une place revient à consommer et à émettre un jeton à la même place. Et il est clair qu'une telle transformation préserve les séquences de franchissement et le comportement de blocages éventuel dans un réseau donné. Pour faire disparaître les arcs inhibiteurs, le processus est plus compliqué. Ce que propose [16], est une sémantique qui ajoute pour chaque Ψ-terme clos trois autres places dont deux sont exclusives, c'est à dire ne peuvent porter de token au même moment. Ces deux places dont l'une est associée à la présence certaine d'un token dans la place du *Psi*-terme considéré, et l'autre place est le complémentaire de la première. La somme de tokens dans ces deux places est toujours égale à 1. Initialement, c'est la place liée à l'absence du Ψ-terme clos qui portera le token. Toute la démarche de cette transformation peut donc être vue en détail dans [16].

Nous aimerions redire que le réseau simple obtenu après cette transformation préservera non seulement les séquences de franchissement qui étaient présentes dans le réseau contextuel de départ, mais aussi les deadlocks.

Chapitre 4

Sémantique opérationnelle du langage $\mathcal{L}^r_\Psi$ exprimée en termes des réseaux de Petri

Dans ce chapitre, nous allons exprimer la sémantique opérationnelle du langage $\mathcal{L}^r_\Psi$ en utilisant les concepts et les termes relatifs aux réseaux de Petri. Dans le chapitre 2, basé sur [13], cette même sémantique a été exprimée sous forme des règles de transition dans le style de Plotkin. Nous savons, selon [14], qu'il existe dans la littérature deux groupes principaux de sémantiques relatives aux réseaux de Petri. Il y a, d'une part, le groupe des sémantiques dites *orientés location*. Celles-ci exploitent la *structure syntaxique* des termes de processus, notamment l'opérateur parallèle, pour définir les ensembles des places associés. D'autre part, il y a le groupe des sémantiques dites *orientées étiquette*. Ces dernières ignorent la structure syntaxique de l'opérateur parallèle et gardent seulement l'information contenue sur les étiquettes des transitions du réseau. N. Busi et R. Gorrieri affirment dans [14] que si le premier groupe produit des réseaux assez larges, il a néanmoins l'avantage d'être très général et peut, par conséquent, être appliqué dans plusieurs algèbres de processus. Tandis que le second groupe produit des réseaux plus petits, mais ne peut pas facilement être étendu pour faire face à certains types d'opérateurs dont la composition alternative et la restriction. De ce fait, la sémantique que nous proposons dans ce chapitre appartient plutôt au premier groupe précité d'autant plus que nous utilisons la composition alternative comme un des opérateurs dans la construction des agents (Cf. Définition 2.1.9).

Les primitives de coordination du langage $\mathcal{L}^r_\Psi$ portent des noms intuitivement indicatives, il peut être, par conséquent, assez aisé de comprendre que

nous utiliserons un arc normal dans l'expression de la primitive de retrait `get` ou de la primitive de placement `tell`, étant donné qu'il est question de *consommer* ou de *produire* un jeton. Par ailleurs, pour la primitive de lecture `ask`, nous utiliserons l'arc contextuel, puisque, dans ce cas, nous vérifierons la présence d'un jeton sans le consommer. Et, en dernier lieu, pour la primitive de *non-présence* `nask`, nous utiliserons l'arc inhibiteur, puisque nous aurons à tester l'absence d'un jeton.

Nous reviendrons plus en détail à chaque étape de la construction de cette sémantique dans la suite de ce chapitre.

Nous présentons une démarche progressive dans la présentation de la sémantique opérationnelle du langage $\mathcal{L}_{\Psi}^{r}$. Tout au long de ce chapitre, le lecteur se rendra compte que la construction de cette sémantique ne s'éloignera pas de l'option d'une éventuelle implémentation. Ainsi, la démarche suivie dans cette construction s'apparentera beaucoup à une analyse fonctionnelle en vue d'une implémentation.

4.1 Construction des éléments constitutifs

Nous nous proposons donc de partir d'un agent A exprimé en $\mathcal{L}_{\Psi}^{r}$ et d'aboutir à un réseau de Petri dont le comportement exprime celui de cet agent. Il importe de signaler à ce stade que nous nous imposons une hypothèse de travail qui est de limiter l'ensemble des valeurs que peuvent prendre les variables de communication. En effet, dans le chapitre 2, aucune contrainte n'est imposée sur les types de valeurs que peut prendre une variable de communication. Nous imposons cette contrainte ici pour éviter une explosion des places dans le réseau à construire et ainsi rendre plus abordable la conception d'un outil qui implémente la sémantique dont il est question dans ce chapitre.

Afin de dégager tous les éléments significatifs pour la compréhension de son comportement, nous proposons, nous avons choisi de faire un *parsing* de l'agent exprimé en $\mathcal{L}_{\Psi}^{r}$.
En fait, le *parsing complet* d'un agent A permet de répertorier :

1. toutes les valeurs constantes présentes dans l'agent, et lesquelles valeurs peuvent être classées selon leurs types respectifs. Cela peut être représenté par l'ensemble suivant :

$$\mathcal{M}_{\mathcal{A}} = \{T_1 = \{t_1^1, \cdots, t_{k_1}^1\} \cup \{\perp\}, \cdots, T_z = \{t_1^z, \cdots, t_{k_z}^z\} \cup \{\perp\}\} \quad (4.1)$$

où les T_i, $1 \leq i \leq z$, représentent les différents types présents dans l'agent A ; et pour un i et un j donnés, t_j^i représente la $j^{\text{ième}}$ valeur constante du type T_i présent dans l'agent A ; et $\perp$ représente une valeur non définie ;

2. toutes les variables de communication présentes dans l'agent A, en précisant le type de chacune d'elles, compte tenu de l'hypothèse de travail que nous nous sommes imposés. Et l'ensemble de ces variables est :

$$Acvar(A) = \{(X_1 : T_{i_1}), \cdots, (X_l : T_{i_l})\} \tag{4.2}$$

où les variables X_j sont toutes distinctes. Et chaque variable de communication X_j est de type T_{i_j}.
Dans un contexte où il n'y a pas de confusion, nous noterons souvent l'ensemble des variables de communication relatives à l'agent A en omettant le type, i.e. nous écrirons $Acvar(A) = \{X_1, \cdots, X_l\}$;

3. tous les Ψ-termes, clos et non-clos, présents dans l'agent A

$$\Psi_A = \{\psi_1, \cdots, \psi_m\} \tag{4.3}$$

où les ψ_i sont distincts les uns des autres ;

4. tous les agents (ou mieux *sous-agents*) composant l'agent A, hormis, bien sûr, l'agent A lui-même

$$\mathcal{A}_A = \{A_1, \cdots, A_n\} \tag{4.4}$$

où les A_i sont les *sous-agents* distincts figurant dans A.

Il peut être intéressant de rappeler que le fait d'utiliser les ensembles pour grouper les différents éléments constitutifs de l'agent permet d'éviter les dédoublements. Par la suite, nous verrons que ce choix peut aider à construire des réseaux où le nombre des places peut être réduit.

Nous pouvons remarquer, notamment dans (4.2), qu'une place importante est accordée à la notion de la *signature*. Dans la suite, lorsque nous parlerons d'un champ présent dans un foncteur, nous tiendrons compte de sa position, exprimée par l'indice du champ, et de son type.

4.2 Les places et les transitions d'un réseau associé à un agent

Après avoir répertorié tous les éléments qui composent l'agent A, nous pouvons construire un réseau de Petri qui *simule* le comportement de cet

agent. Mais avant, il importe de signifier que nous avons opté de présenter une vue *statique* du comportement de l'agent. C'est pour cette raison que nous allons construire un réseau qui tienne d'emblée compte de tous les cas de figure dans le comportement de l'agent. Ceci est rendu possible par le fait que tous les éléments qui constituent l'agent sont connus, à partir du parsing dont nous avons parlé plus haut. Cela nous permettra, dans un premier temps, de construire toutes les places nécessaires pour exprimer tous états de l'agent de départ ; et dans un second temps, de construire les transitions qui, d'une manière générale, représenteront les opérateurs et autres primitives comme nous le verrons plus loin.

Rappelons qu'un réseau de Petri N est défini comme un triplet (S, T, m_0), avec S l'ensemble des places, T l'ensemble des transitions et m_0 le marquage initial du réseau.

Comme nous venons de le dire ci-haut, nous allons définir l'un après l'autre les différents éléments qui composent un réseau de Petri que nous voulons construire.

4.2.1 Définition des places du réseau

Dans la suite de ce travail, nous écrirons $[P]$ pour dénoter la place P.

Définition 4.2.1 *On appelle l'ensemble des* places liées aux variables *d'un agent A, l'ensemble, noté S_V, défini comme suit :*

$$S_V = \bigcup_{(X:T_i)\in Acvar(A)} S_X, \text{ où } S_X = \{[X = t^i_j] : 1 \leq j \leq k_i\}, \forall (X : T_i) \in Acvar(A).$$

Autrement dit, S_X, appelé l'ensemble des places liées à la variable X, *contient autant de places qu'il y a des valeurs constantes de type T_i (le type de la variable de communication X) dans l'agent A. Par ailleurs, nous choisissons de marquer dès le départ les places $[X = \perp]$. En fait, à chaque instant et pour chaque variable de communication X, une seule place liée à cette variable portera un jeton. Et cette place est $[X = \perp]$ au début du processus.*

Définition 4.2.2 *On appelle l'ensemble des* places liées aux Ψ-termes, *noté S_P, l'ensemble décrit comme suit :*

$$S_P = \bigcup_{\psi\in\Psi_A} S_\psi,$$

avec S_ψ, appelé l'ensemble des places liées au Ψ-terme ψ, *défini, selon le cas, de la manière suivante : pour $\psi \in \Psi_A$,*

1. *si le Ψ-terme $\psi = f_{/r}(x_1 = value_1, \ldots, x_r = value_r)$ est clos, alors l'ensemble S_ψ est défini comme suit :*

$$S_\psi = \{[f_{/r}(x_1 = value_1, \cdots, x_r = value_r)]\}$$

2. *et si $\psi = f_{/r}(x_1 = value_1, \ldots, x_r = value_r)$ est un Ψ-terme non clos et contient, par exemple, une variable de communication, soit $value_k = X$, avec $k \in \{1, \ldots, r\}$. Dans ce cas, l'ensemble des places est défini comme suit :*

$$S_\psi = \{[f_{/r}(x_1 = value_1, \cdots, x_{k-1} = value_{k-1}, x_k = t^i_j, x_{k+1} = v_{k+1}, \cdots, x_r = v_r)] : t^i_j \in T_i, T_i \text{ contenant des éléments de type de } X\}.$$

Remarque 4.2.1

1. *A propos de la définition 4.2.2(1), nous pouvons remarquer que pour un Ψ-terme clos donné $f_{/r}(x_1 = value_1, \ldots, x_r = value_r)$, nous créons une nouvelle place à mettre dans le réseau de Petri à construire. Il nous importe de signaler que dans un premier temps, nous avons pensé créer autant de places qu'il y a des croisements entre tous les ensembles d'éléments dont le type est présent dans le Ψ-terme. Nous aurions, dans ce cas, $(\sharp T_{h_1}) \times \cdots \times (\sharp T_{h_r})$ places à ajouter dans le réseau, avec T_{h_i} l'ensemble de tous éléments de même type que $value_i$. Bien que cette dernière conception ait l'avantage d'avoir à portée de la main toutes les valeurs possibles que puisse prendre un foncteur ayant une signature donnée, elle présente un gros inconvénient, celui de risquer de se retrouver ensuite avec plusieurs places isolées dans le réseau, à moins de définir des transitions qui les* désenclaveraient, *mais des transitions qui ne seront pas tirées et, par conséquent, n'influenceront pas le comportement du réseau. Cette façon de procéder n'est pas de nature à rendre aisée l'analyse ultérieure dudit réseau.*
2. *Notons pour la définition 4.2.2(2) que le nombre des places que nous aurons à ajouter dans le réseau de Petri en construction est le cardinal de T_i, l'ensemble de tous les éléments de même type que la variable de communication X. En fait, pour cette dernière, les places qui lui sont associées figurent déjà dans le réseau et sont dans l'ensemble S_X, lui même inclus dans S_V.*

   Par cette même définition, nous pouvons comprendre que si le ψ contient, de manière plus générale, n variables de communication, $X_{i_1}, \cdots, X_{i_n}$,

$n \in \mathbb{N}$, nous croiserons les différents ensembles pour avoir les places à ajouter. Ainsi, nous aurons donc $\sharp(T_{i_1}) \times \cdots \times \sharp(T_{i_n})$ places à ajouter dans le réseau, avec T_{i_j} l'ensemble de toutes les valeurs constantes de même type que X_{i_j}.

Il serait, par ailleurs, utile de signaler aussi que si pour $\psi = f_{/r}(x_1 = val_1, \ldots, x_r = val_r) \in \Psi_A$, un des val_i est un Ψ-terme, le raisonnement restera le même que pour ce cas. En fait, si val_i est un Ψ-terme, noté ψ_1, et que $\forall j, 1 \leq j \leq r, j \neq i$, val_j est une valeur constante, la démarche va consister à traiter *d'abord le Ψ-terme* intérieur *ψ_1. Si ψ_1 est clos, alors ψ est aussi clos, car ne contenant aucune variable de communication. Par conséquent, l'ensemble S_{ψ_1} sera d'abord construit (selon la définition 4.2.2(1)), avant l'ensemble S_ψ.*

De même, si ψ_1 n'est pas clos, nous nous situons dans une démarche récursive dans laquelle S_{ψ_1} est d'abord construit selon la définition 4.2.2(2), et éventuellement d'autres ensembles des places associées aux Ψ-termes de même niveau de ψ_1 dans ψ. C'est seulement après toutes ces étapes que S_ψ sera construit.

Brièvement, nous disons que la construction des ensembles des places associées aux Ψ-termes se fait de l'intérieur vers l'extérieur en utilisant éventuellement la démarche récursive. Rappelons ici que le fait d'avoir choisi des définitions sur base des ensembles nous permettra d'éviter des dédoublements des éléments.

Définition 4.2.3 *Soit un agent A. Nous appellerons l'ensemble des* places relatives à l'agent *A, notée S_{Ag}, un ensemble composé d'une ou deux places directement liées à l'*agent *A.*

Définition 4.2.4 *Nous appellerons un agent* élémentaire *tout agent qui ne contient pas un autre agent.*

Note 4.2.1
Rappelons que l'état global d'un réseau de Petri, en un instant, est donné son marquage en cet instant. Dans le formalisme qui est le nôtre dans ce travail, l'évolution du marquage d'un réseau associé à un agent est lié aux différentes opérations et primitives présentes dans cet agent. Or les opérations et les primitives sont, d'une certaine façon, représentées par les transitions, les ensembles des places associées aux agents sont intimément liés aux transitions. Nous allons donc définir, dans le détail, ces ensembles ces places particulières en même temps que les transitions dans la sous-section suivante.

4.2.2 Les transitions et les places liées aux sous-agents

Rappelons que dans le langage $\mathcal{L}^r_\Psi$ un agent (étendu) A est dans une des formes suivantes :

$$A ::= E \mid c \mid A \,;\, A \mid A \,||\, A \mid A + A \mid I \mid X_\nu$$

Nous devons signaler que, dans le cadre de ce travail, nous ne définissons pas la sémantique pour les deux derniers agents. Pour le premier de ces agents, l'agent I est sensé être une instruction dans un langage d'accueil donné. Et dans ce travail, nous avons fait le choix d'étudier le langage $\mathcal{L}^r_\Psi$ en dehors de tout langage d'accueil. En ce qui concerne le deuxième agent définissant le mécanisme de récursivité, il pourrait être ajouté par la suite si l'on précise une manière adéquate de définir le rafraichissement de renommage des variables locales. Nous pensons que cela peut ouvrir une piste pour une étude ultérieure.

Pour revenir au réseau de Petri en construction, nous savons, à ce stade, que les places relatives aux Ψ-termes et aux variables de communication qui composent l'agent sont déjà présentes dans le réseau de Petri par les étapes décrites dans la sous-section précédente. L'ensemble des places relatives aux agents va être défini simultanément avec les transitions. Le réseau de Petri que nous allons construire est, bien sûr, contextuel. Et, à ce niveau, nous considérons acquise la transformation d'un réseau de Petri contextuel en un réseau de Petri traditionnel (Cf. section 3.3). Ainsi, nous utilisons la définition 4.2.2 pour les transitions.

Pour rappel, une transition t est définie comme un quadruplet

$$(c, r, i, p) \in T \subseteq \mathcal{M}_{fin}(S) \times \mathcal{P}_{fin}(S) \times \mathcal{P}_{fin}(S) \times \mathcal{M}_{fin}(S).$$

Il est un fait que si nous nous en tenons aux agents qui nous intéressent dans le présent travail, dans un agent élémentaire la principale opération est une primitive de coordination.

Nous allons définir les ensembles des transitions des agents élémentaires, en tenant compte des deux cas possibles : le cas où ψ est clos et le cas où il ne l'est pas. Mais avant, nous définissons deux concepts dont l'un n'est pas sans rappeler la notion de l'agent E qui exprime la fin d'une exécution dans les règles de transitions du langage $\mathcal{L}^r_\Psi$.

Définition 4.2.5 *Soit N_A le réseau relatif à un agent A.*

a. *Nous appellerons place de* lancement *ou place de* départ *de A, notée $s_{\mathcal{L}_A}$, la place dans laquelle le jeton doit être placé pour lancer l'exécution de l'agent A.*
Et de façon analogue :

b. *Nous appellerons place* finale *ou* terminale, *notée $s_{\mathcal{T}_A}$, une place qui n'appartient ni à S_V, ni à S_P et dans laquelle le jeton* échoue *en dernier lieu si l'exécution relative à l'agent A se déroule* normalement.

Remarque 4.2.2

- *Hormis pour les réseaux relatifs aux agents élémentaires, nous devons signaler que les deux places, $s_{\mathcal{L}_A}$ et $s_{\mathcal{T}_A}$ ne portent, d'une manière générale, pas de sémantique particulière, mais renforceront l'expressivité de la sémantique et seront fort utiles lorsqu'il sera, par exemple, question de* composer *- séquentiellement, parallèlement ou alternativement - deux ou plusieurs réseaux de Petri. Elles serviront à rendre lesdites compositions plus faciles à réaliser et plus lisibles.*
- *De manière plus précise, la place* finale *reçoit un jeton de la même façon que les places constituant le* post-ensemble *de la dernière transition tirée pendant une exécution normale.*
- *Contrairement pour la place finale, nous indiquerons souvent le nom de la place de lancement en utilisant le nom de l'agent auquel elle est associée s'il s'agit d'un agent élémentaire ; et pour un agent obtenu par composition, le nom sera formé par les noms des agents composants et de l'opération utilisée.*

Définition 4.2.6 *Pour tout agent élémentaire A, l'ensemble des places liées à l'agent A, S_{Ag}, est défini par :*

$$S_{Ag} = \{[A], s_{\mathcal{T}_A}\},$$

où $[A]$ représente la place de lancement, $s_{\mathcal{L}_A}$.

Définition 4.2.7 *Soit A un agent élémentaire et N_A son réseau associé. L'ensemble des places de N_A, noté S, est défini comme ceci*

$$S = S_V \cup S_P \cup S_{Ag},$$

où

- *S_V est l'ensemble des places liées aux variables de A, conformément à la définition 4.2.1 ;*

- S_P *l'ensemble des places liées aux* Ψ*-termes figurant dans* A*, selon la définition 4.2.2;*
- S_{Ag} *l'ensemble des places liées à l'agent* A*, conformément aux définitions 4.2.3 et 4.2.6.*

Remarque 4.2.3

- *Etant donné la nature différente des entités auxquelles ils se rapportent, les trois ensembles* S_V*,* S_P *et* S_{Ag} *forment une partition de l'ensemble des places* S*. C'est-à-dire, ces trois ensembles nous donnent toutes les places du réseau, et ils sont disjoints l'un de l'autre.*
- *Puisque nous nommons les places liées à l'agent à partir du nom de ce dernier, nous prendrons toujours un nom* frais *pour gérer la situation où l'aurait deux réseaux qui ont les mêmes noms pour les places de lancement ou places finales.*

Nous allons maintenant définir les ensembles des transitions pour les agents élémentaires. Avec l'ensemble des places et l'ensemble des transitions, il ne manquera plus que le marquage initial pour que le réseau soit complet. Nous avons fait le choix de définir en même temps que l'ensemble des transitions du réseau relatif à un agent et le marquage initial associé.

Définition 4.2.8 *Soit* $A = \texttt{tell}(\psi)$*. L'ensemble des transitions du réseau associé à* A*, noté* T*, est construit comme suit :*

- *si* ψ *est clos, alors*

$$T = \{(\{[tell(\psi)]\}, \emptyset, \emptyset, \{[\psi], s_{T_A}\})\};$$

- *et si* ψ *est non clos et* $Acvar(\psi) = \{(X_1 : T_1), \ldots, (X_l : T_l)\}$ *l'ensemble des variables de communication figurant dans* ψ*, alors*

$$\begin{aligned} T \;=\; & \{(\{[\texttt{tell}(\psi)]\}, \{[X_1 = t_1], \ldots, [X_l = t_l]\}, \emptyset, \{[\psi_{|\{[X_1=t_1],\ldots,[X_l=t_l]\}})]\} \\ & \cup\{s_{T_A}\}) : (t_1, \ldots, t_l) \in \prod_{i=1}^{l} T_i \wedge (X_i : T_i) \in Acvar(\psi), 1 \leq i \leq l\} \end{aligned}$$

  où $\psi_{|\{[X_1=t_1],\ldots,[X_l=t_l]\}}$ *est le* Ψ*-terme clos obtenu à partir de* ψ *où chaque variable de communication* X_i *prend une valeur* t_i *dans l'ensemble des valeurs de type* T_i*.*
- *Et pour ces deux cas, le marquage initial sera donné par* $m_0 = \{([\texttt{tell}(\psi)], 1)\}$*.*

Par rapport à cette définition, il y a certaines choses que nous pouvons éclaircir. D'abord, nous pouvons comprendre assez facilement que la place de lancement, que nous avons notée $s_{\mathcal{L}_A}$ plus haut, est noté $\texttt{tell}(\psi)$. En effet, c'est dans cette place que nous mettons le jeton qui va servir à exécuter le réseau relatif à l'agent A.

Par ailleurs, nous savons, par la définition 3.1.5, qu'un marquage est un multi-ensemble sur l'ensemble des places, et dans la troisième partie de la précédente définition, nous représentons le marquage initial par un ensemble de couples dont les abscisses sont justement des places, et les ordonnées sont des entiers (exprimant le nombre de jetons contenus dans la place). Et, c'est le même raisonnement qui prévaut pour les autres agents élémentaires.

Définition 4.2.9 *Soit un agent $A = \texttt{ask}(\psi)$. L'ensemble des transitions du réseau associé à A, noté T, est défini comme suit :*

- *si ψ est clos, alors l'ensemble des transitions relatives à cet agent est donné par*

$$T = \{(\{[\texttt{ask}(\psi)]\}, \{[\psi]\}, \emptyset, \{s_{\mathcal{T}_A}\})\};$$

- *et si ψ est non clos, et si l'on suppose, comme précédemment, $Acvar(\psi) = \{(X_1 : T_1), \ldots, (X_l : T_l)\}$ l'ensemble des variables de communication figurant dans ψ ; et si l'on note par $\psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}}$ un Ψ-terme clos, ψ', auquel ψ correspond (au sens de la définition 2.1.5, $\psi \lhd \psi'$), c'est à dire chaque champ ch_i correspond en position et en type à la variable de communication X_i, apparaissant dans ψ, et prend une valeur t_i (il est clair que la notion de la signature d'un foncteur intervient de manière notable dans cette démarche). L'ensemble des transitions relatives à cet agent est donné par*

$$\begin{aligned} T \;=\; & \{(\{[\texttt{ask}(\psi)]\} \cup \{[X_1 = t'_1], \ldots, [X_l = t'_l]\}, \{[\psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}}]\}, \emptyset, \\ & \{[X_1 = t_1], \ldots, [X_l = t_l]\} \cup \{s_{\mathcal{T}_A}\}) : ([X_1 = t'_1], \ldots, [X_l = t'_l]) \in \\ & \prod_{i=1}^{l} S_{X_i} \wedge \psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}} \in S \wedge (X_i : T_i) \in Acvar(\psi), 1 \leq i \leq l\}; \end{aligned}$$

- *et le marquage initial est représenté, comme pour la définition précédente, par un jeton à la place de lancement, $m_0 = \{([\texttt{ask}(\psi)], 1)\}$.*

Définition 4.2.10 *Soit un agent $A = \texttt{nask}(\psi)$. L'ensemble des transitions est défini comme suit :*

- *si ψ est clos, alors l'ensemble des transitions relatives à cet agent est donné par*

$$T = \{(\{[\texttt{nask}(\psi)]\}, \emptyset, \{[\psi]\}, \{s_{T_A}\})\};$$

- *et si ψ est non clos, alors, en considérant ψ' comme dans la définition précédente, l'ensemble des transitions est :*

$$\begin{aligned} T \;=\; & \{(\{[\texttt{nask}(\psi)]\}, \emptyset, \{[\psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}}] : \psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}} \in S_P \\ & \wedge (X_i : T_i) \in Acvar(\psi), 1 \leq i \leq l\}, \{s_{T_A}\})\}; \end{aligned}$$

- *le marquage initial se présente de la même façon que précédemment, $m_0 = \{([\texttt{nask}(\psi)], 1)\}$.*

Définition 4.2.11 *Si l'agent $A = \texttt{get}(\psi)$, son ensemble des transitions est défini comme suit :*

- *si ψ est clos, l'ensemble des transitions associées est*

$$T = \{(\{[\texttt{get}(\psi)], [\psi]\}, \emptyset, \emptyset, \{s_{T_A}\})\};$$

- *et si ψ est non-clos et si ψ' est comme dans les deux dernières définitions, l'ensemble des transitions associées est*

$$\begin{aligned} T \;=\; & \{(\{[\texttt{get}(\psi)]\} \cup \{[\psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}}]\} \cup \{[X_1 = t'_1], \ldots, [X_l = t'_l]\}, \\ & \emptyset, \emptyset, \{[X_1 = t_1], \ldots, [X_l = t_l]\} \cup \{s_{T_A}\}) : ([X_1 = t'_1], \ldots, [X_l = t'_l]) \\ & \in \prod_{i=1}^{l} S_{X_i} \wedge \psi'_{\ni\{ch_1=t_1,\ldots,ch_l=t_l\}} \in S \wedge (X_i : T_i) \in Acvar(\psi), 1 \leq i \leq l\}; \end{aligned}$$

- *le marquage initial se présente, comme pour tous les agents élémentaires définis précédemment, $m_0 = \{([\texttt{get}(\psi)], 1)\}$.*

A ce niveau, les réseaux associés aux agents élémentaires sont définis. Nous allons revenir plus tard sur le fonctionnement de ces réseaux et leur équivalence avec, respectivement, les règles **(T)**, **(A)**, **(N)** et **(G)** de la section 2.2.

Pour l'instant, définissons les différentes opérations de composition. L'idée est de pouvoir *fusionner* deux ou, par récurrence, n ($n \in \mathbb{N}$) réseaux de Petri, chacun associé à un (sous-)agent, en utilisant les opérations de composition séquentielle, parallèle et alternative respectivement afin d'obtenir un réseau de Petri (plus grand) représentant le comportement de tout l'agent. Pour nous donner les moyens d'exprimer plus facilement les changements qui interviennent au niveau du réseau globlal, nous allons distinguer certains ensembles particuliers de transitions, comme nous l'avons fait précédemment pour les places.

Définition 4.2.12 *Soit N_A le réseau associé à l'agent A.*

a. *Nous appellerons une transition de* lancement, *notée $t_{\mathcal{L}}$, toute transition de N_A dont le pré-ensemble contient la place de lancement de N_A. L'ensemble des transitions de* lancement *de A sera noté $T_{\mathcal{L}_A}$.*
De la même façon,

b. *nous appellerons une transition de* terminaison *ou transition* finale, *notée $t_{\mathcal{T}}$, toute transition de N_A dont le post-ensemble contient la place finale de N_A. Et l'ensemble des transitions de terminaison de N_A sera noté $T_{\mathcal{T}_A}$.*

Rappelons que notre démarche consiste à composer deux réseaux de Petri au moyen de chacune des opérations de composition.

Définition 4.2.13 *Soient les agents A, B et, N_A et N_B les réseaux associés respectifs. Pour l'agent $A;B$ obtenu par composition séquentielle de A et B,*

- *l'ensemble des places est défini par $S_{A;B} = S_A \cup S_B$, où S_A est l'ensemble des places de N_A et S_B celui de N_B ;*
- *quant à l'ensemble des transitions, il est :*

$$T_{A;B} = T_A \cup T_B \cup \{(\{s_{\mathcal{T}_A}\}, \emptyset, \emptyset, \{s_{\mathcal{L}_B}\})\};$$

- *pour le marquage initial, il est $m_0 = \{(s_{\mathcal{L}_A}, 1)\}$.*

Exemple 4.2.1
Si l'on considère les deux réseaux relatifs sur la figure 4.1, relatifs à deux agents A et B respectivement, alors leur composition séquentielle est le réseau donné à la figure 4.2.

Faisons tout de suite remarquer que cette opération consiste, en fait, à mettre le réseau N_B à la suite du réseau N_A, comme le montre la figure 4.2. Pour ce faire, nous utilisons la place *finale* du premier réseau et la place de *lancement* du second en les reliant par une nouvelle transition.
Dans ce cas précis, nous pouvons constater que la place de *lancement* du réseau associé à l'agent $A;B$ n'est rien d'autre que celle de N_A, et la place *finale* du réseau global est la place finale du réseau N_B. Autrement dit, $s_{\mathcal{L}_{A;B}} = s_{\mathcal{L}_A}$ et $s_{\mathcal{T}_{A;B}} = s_{\mathcal{T}_B}$. Nous y reviendrons avec force détail dans la suite.

Définition 4.2.14 *Soient A et B deux agents, et N_A et N_B leurs réseaux de Petri associés. Pour l'agent $A \| B$,*

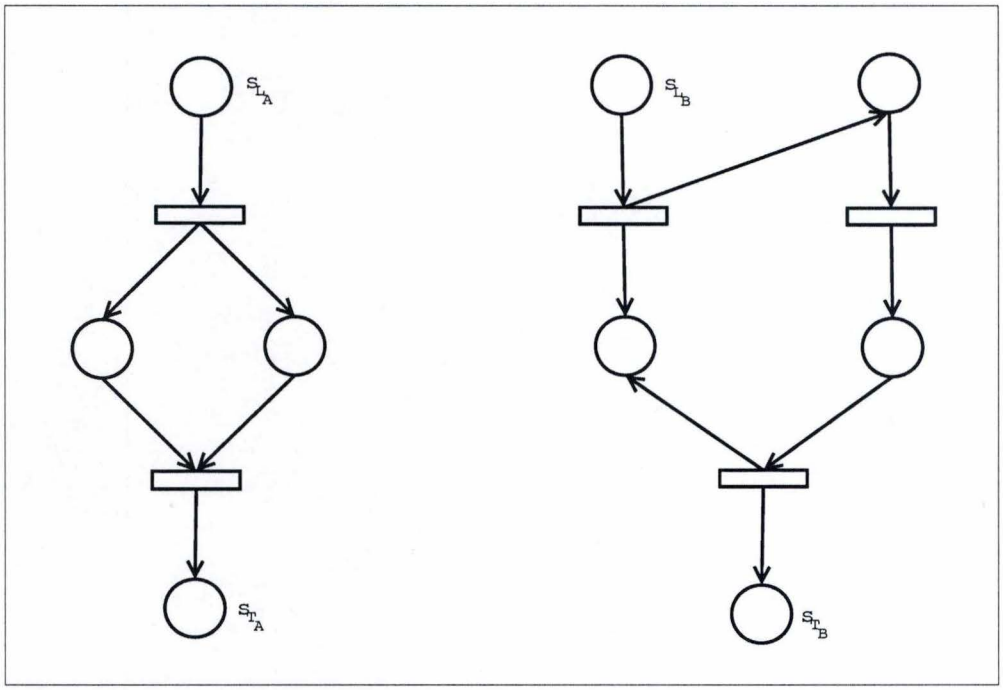

FIG. 4.1 – Deux réseaux relatifs à deux agents A et B respectivement.

- *l'ensemble des places est donné par :*

$$S_{A\|B} = S_A \cup S_B \cup S_{par}$$

avec S_A et S_B les ensembles des places associés respectivement aux réseaux N_A et N_B et, $S_{par} = \{[A\|B], s_{\mathcal{T}_{A\|B}}\}$;
- *et l'ensemble des transitions est, quant à lui, donné par*

$$\begin{aligned} T_{A\|B} \;=\; & T_A \cup T_B \cup \{(\{[A\|B]\}, \hat{t}, {}^\circ t, t^\bullet \cup \{s_{\mathcal{L}_B}\}) : t \in T_{\mathcal{L}_A}\} \\ & \cup \{(\{[A\|B]\}, \hat{t'}, {}^\circ t', t'^\bullet \cup \{s_{\mathcal{L}_A}\}) : t' \in T_{\mathcal{L}_B}\} \\ & \cup \{(\{s_{\mathcal{T}_A}, s_{\mathcal{T}_B}\}, \emptyset, \emptyset, \{s_{\mathcal{T}_{A\|B}}\})\}; \end{aligned}$$

- *le marquage initial est donné par $m_0 = \{([A\|B], 1)\}$.*

Comme pour les différents réseaux construits jusqu'à présent, nous reviendrons aussi pour celui-ci de manière détaillée plus tard. Nous pouvons tout de même épingler le fait d'utiliser encore S_{Ag} pour désigner des places qui sont directement liées à l'agent courant, $A\|B$ pour le cas présent. Et S_{Ag} est, cette fois, constitué des places de lancement et de terminaison de l'agent courant. Bien qu'il soit possible de définir une *concurrence* avec les réseaux

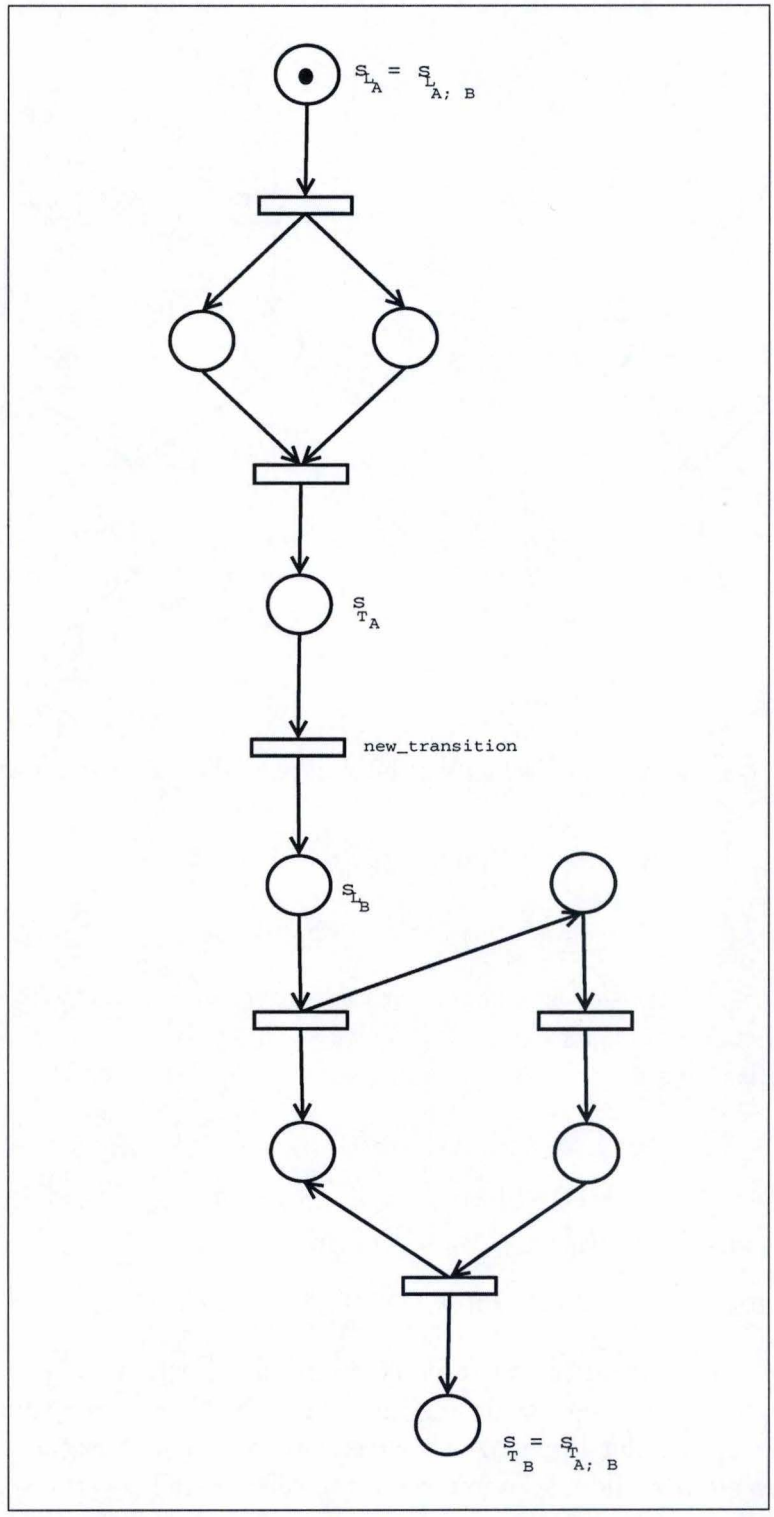

FIG. 4.2 – Composition séquentielle de deux réseaux relatifs aux agents de la figure 4.1.

de Petri comme proposé dans [6], le parallélisme tel que défini ci-dessus est perçu sous l'angle de l'entrelacement de l'exécution de deux agents.

Exemple 4.2.2
Si l'on considère les réseaux de la figure 4.1, leur composition parallèle donne le réseau de la figure 4.3. On peut constater qu'aussi bien le premier que le deuxième réseau n'a qu'une transition de lancement. Il s'ensuit, selon la définition 4.2.14, que le réseau résultant de la composition parallèle aura deux places et trois transitions de plus que les deux réseaux réunis.

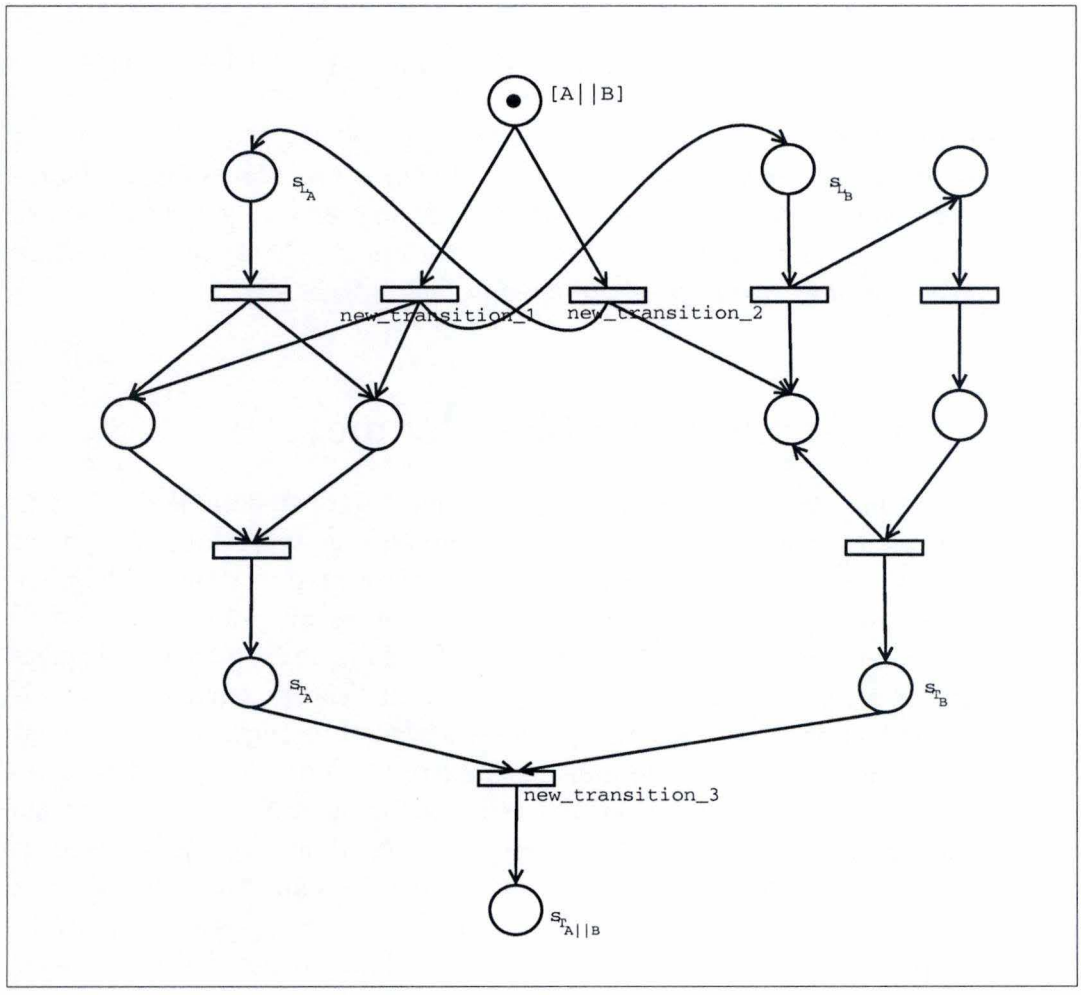

FIG. 4.3 – Composition parallèle de deux réseaux relatifs aux agents de la figure 4.1.

Définition 4.2.15 *Soient A et B deux agents, et N_A et N_B leurs réseaux de Petri associés. Pour l'agent $A+B$, si l'on définit $S_{alt} = \{[A+B], s_{\mathcal{T}_{A+B}}\}$ l'ensemble des places associées à cet agent, alors*

- *l'ensemble de toutes les places est donné par*

$$S_{A+B} = (S_A \setminus \{s_{\mathcal{L}_A}\}) \cup (S_B \setminus \{s_{\mathcal{L}_B}\}) \cup \{[A+B], s_{\mathcal{T}_{A+B}}\};$$

- *l'ensemble des transitions est donné par*

$$\begin{aligned} T_{A+B} \;=\; & (T_A \setminus T_{\mathcal{L}_A}) \cup (T_B \setminus T_{\mathcal{L}_B}) \cup \{(\{[A+B]\}, \hat{t}, {}^\circ t, t^\bullet) : t \in (T_{\mathcal{L}_A} \cup T_{\mathcal{L}_B})\} \\ & \cup \{(\{s_{\mathcal{T}_A}, s_{\mathcal{T}_B},\}, \emptyset, \emptyset, \{s_{\mathcal{T}_{A+B}}\})\}; \end{aligned}$$

- *et pour le marquage initial, il est donné par $m_0 = \{([A+B], 1)\}$.*

Exemple 4.2.3
En considérant les réseaux de la figure 4.1. On a une transition de lancement dans chacun de deux réseaux. Après leur suppression et la création des nouvelles places et transitions, selon la définition 4.2.15 de la composition alternative, on a le réseau présenté dans la figure 4.4.

4.3 Une sémantique bien définie

Nous allons, dans cette section, prouver que les constructions de réseaux que nous avons proposées définissent une sémantique opérationnelle qui est équivalente avec celle décrite par les règles de transition dans le chapitre 2. Nous voulons préalablement faire quelques mises au point afin d'éviter toute confusion. Premièrement, dans notre formalisme les places marquées (i.e. portant au moins un jeton) représentent tout l'espace partagé, qui comprend aussi bien les Ψ-termes clos que les différents bindings, respectivement représentés par *Sstore* et *Sbind* dans le chapitre 2. Pour un agent donné, les Ψ-termes clos *se transforment* en places appartenant à S_P, et les différents bindings sont *transformés* par des places qui sont dans S_V. Contrairement aux places liées aux Ψ-termes clos, celles qui sont liées aux variables peuvent avoir une capacité maximale de 1. Autrement dit, ces places ne peuvent contenir plus d'un jeton. C'est une contrainte dont on peut se passer sans fausser les résultats qui suivront.

Une deuxième mise au point tient plutôt d'un rappel : pour un agent donné, les places liées à ses Ψ-termes et à ses variables de communication, s'il en a, sont placées dans le réseau avant les places liées directement à l'agent

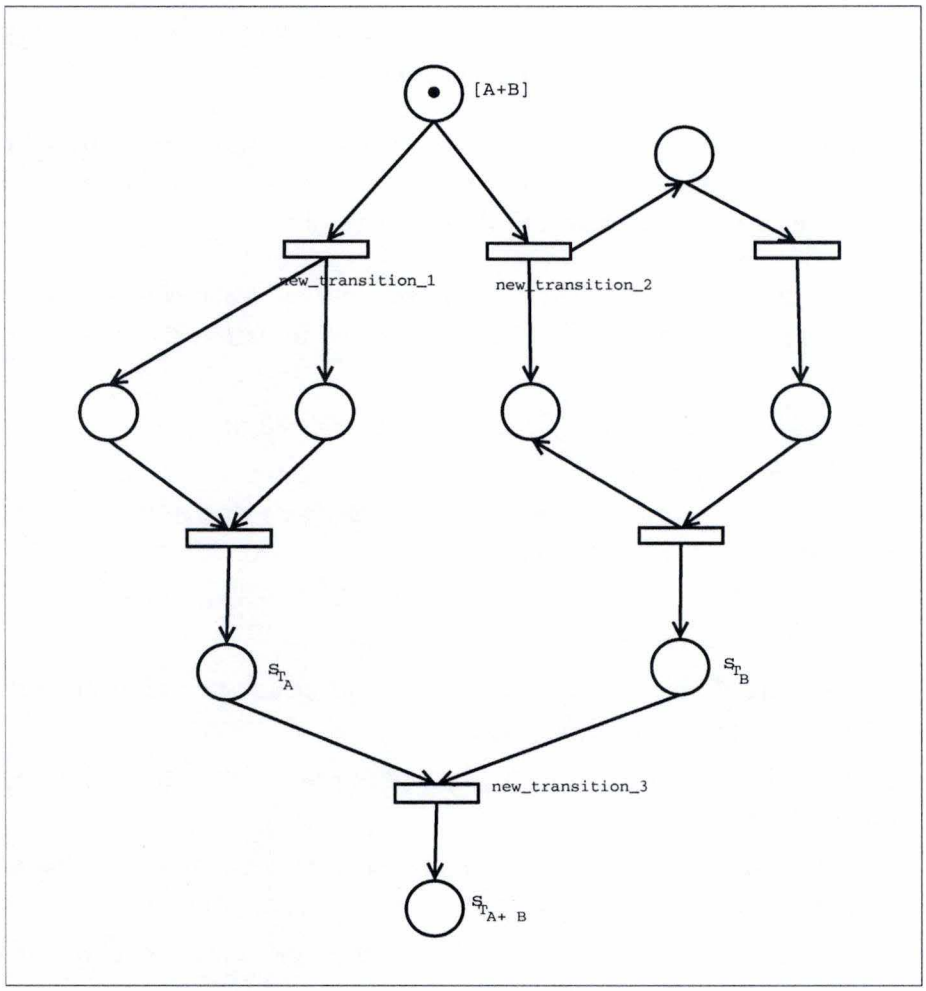

FIG. 4.4 – Composition alternative de deux réseaux relatifs aux deux agents de la figure 4.1.

lui-même.

Une troisième mise au point est aussi un rappel : l'exécution d'un agent consiste en une séquence de franchissement qui commence par le tirage d'une transition de lancement et se termine quand il n'y a plus de concession, i.e. de transition franchissable.

Quatrièmement, lorsque nous construisons un réseau relatif à un agent complexe (i.e. non élémentaire), la place de lancement courante est celle du réseau global. Par conséquent, c'est cette place seulement qui reçoit un jeton

tel que décrit dans les définitions de la section précédente. Nous ne mettrons donc pas de jeton dans les places de lancement des réseaux intermédiaires.

Ces mises au point étant faites, nous pouvons maintenant établir l'équivalence entre la sémantique opérationnelle proposée dans ce chapitre et celle décrite dans les règles de transition du chapitre 2.

Théorème 4.3.1 *Soit un agent A. En considérant les règles de transition* **(T)**, **(A)**, **(N)**, **(G)**, **(S)**, **(P)** *et* **(C)** *exprimées dans la section 2.2 dans le chapitre 2 :*

(i) *si $A = \texttt{tell}(\psi)$, alors la définition 4.2.8 propose une même sémantique opérationnelle que la règle* **(T)** ;

(ii) *si $A = \texttt{ask}(\psi)$, alors la définition 4.2.9 propose une même sémantique opérationnelle que la règle* **(A)** ;

(iii) *si $A = \texttt{nask}(\psi)$, alors la définition 4.2.10 propose une même sémantique opérationnelle que la règle* **(N)** ;

(iv) *si $A = \texttt{get}(\psi)$, alors la définition 4.2.11 propose une même sémantique opérationnelle que la règle* **(G)** ;

(v) *si $A; B$, alors la définition 4.2.13 propose une même sémantique opérationnelle que la règle* **(S)** ;

(vi) *si $A \| B$, alors la définition 4.2.14 propose une même sémantique opérationnelle que la règle* **(P)** ;

(vii) *si $A + B$, alors la définition 4.2.15 propose une même sémantique opérationnelle que la règle* **(C)**.

Preuve : Nous allons démontrer quelques unes de ces assertions, et le raisonnement restera, en somme, le même pour les autres assertions.

(i) - Si ψ est clos, l'unique transition de l'ensemble $T = \{(\{[\texttt{tell}(\psi)]\}, \emptyset, \emptyset, \{[\psi], s_{T_A}\})\}$ indique que le pré-ensemble est le singleton composé de la place de lancement et le post-ensemble est la paire composée de la place finale et de la place du Ψ-terme clos $[\psi]$. Cette dernière existait préalablement dans le réseau et appartient à S_P. Et par le fait que la définition 4.2.8(3) dit que le marquage initial place un jeton à la place de lancement, $[\texttt{tell}(\psi)]$, l'unique transition de T est donc franchissable. Après son tirage, les éléments de son post-ensemble, la place finale et la place $[\psi]$, reçoivent chacun un jeton. Le fait, pour la place $[\psi]$, de recevoir un jeton nous indique qu'elle appartient maintenant à l'espace partagé. Et de là, il n'y a plus de concession. Donc l'exécution s'arrête. C'est ce qui est exprimé

par $\langle E \mid (\theta, \sigma \cup \{\Psi\theta\})\rangle$ dans la section 2.2.

- Et pour le cas où ψ n'est pas clos, avec $Acvar(\psi) = \{(X_1 : T_1), \ldots, (X_l : T_l)\}$ l'ensemble de ses variables de communication, l'ensemble de transitions

$$T = \{(\{[\texttt{tell}(\psi)]\}, \{[X_1 = t_1], \ldots, [X_l = t_l]\}, \emptyset, \{[\psi_{|\{[X_1=t_1],\ldots,[X_l=t_l]\}})]\wedge\} \cup\{s_{T_A}\}) : (t_1, \ldots, t_l) \in \prod_{i=1}^{l} T_i \wedge (X_i : T_i) \in Acvar(\psi), 1\wedge \leq i \leq l\}$$

nous dit que nous créons autant de transitions qu'il y a des *l-uplets* $(t_1, \cdots, t_l)$ -dont les places associées $[X_1 = t_1], \ldots, [X_l = t_l]$ sont déjà dans le réseau- rendant ψ clos. Nous devons comprendre ici que nous tenons compte de toutes les possibilités, c'est à dire que tout l-uplet de $(t_1, \cdots, t_l)$, dont les types de composants correspondent avec ceux des variables de communications $(X_1, \cdots, X_l)$ figurant dans ψ, a sa "chance" de rendre ce dernier clos. Ces places sont déjà présentes dans le réseau et appartiennent à S_V. Mais il est clair qu'un seul l-uplet participera au franchissement effectif d'une transition de lancement lors de l'exécution de l'agent puisque le marquage initial ne place qu'un seul jeton à la place de lancement $[\texttt{tell}(\psi)]$. Nous avons choisi de mettre les places liées aux variables dans l'ensemble contextuel puisque nous savons, par la sémantique de la section 2.2, que les bindings sont les mêmes avant et après l'exécution de l'agent. Et dans notre réseau, cela se traduit par le fait que nous « lisons »sans consommer les jetons qui sont dans ces places et qui fixent les valeurs de toutes les variables de communication *dans* ψ. Comme corrolaire de ce franchissement, il y a production de jeton dans la paire composée de la place finale et de la place contenant le Ψ-terme rendu clos par le bon binding, $\psi_{|\{[X_1=t_1],\cdots,[X_l=t_l]\}})$. Cela traduit, dans les termes propres aux réseaux de Petri, ce qui est exprimé dans la section 2.2 par $\langle \Psi \mid (\theta, \sigma)\rangle \longrightarrow \langle E \mid (\theta, \sigma \cup \{\Psi\theta\})\rangle$.

(ii) - Pour la primitive `ask`, si ψ est clos, la seule transition qui est proposée dans la définition 4.2.9(1) a comme pré-ensemble le singleton formé par la place de lancement $[\texttt{ask}(\psi)]$, et comme post-ensemble un autre singleton composé de la place finale $[s_{T_A}]$. Et contrairement à la précédente primitive, nous avons dans cette transition un ensemble contextuel. Cela se justifie par le fait que le franchissement de la transition de lancement est conditionné par la présence d'un jeton à la place $[\psi]$ mais ce dernier n'est pas consommé par ladite transition. En fait, si ψ est clos, il n'a plus besoin d'un autre Ψ-terme auquel il correspondrait. La présence d'un jeton à la place $[\psi]$, suffit, avec la présence du jeton à la place de lancement (garanti par le marquage initial), pour qu'il y ait franchissement. Ce qui est bien conforme à la règle **(A)**.

- Pour le cas où ψ est non clos, si l'on suppose que $Acvar(\psi) = \{(X_1 : T_1), \ldots, (X_l : T_l)\}$, par la définition 4.2.9(2), on construit, pour chaque Ψ-terme $\psi'_{\ni\{ch_1=t_1,\cdots,ch_l=t_l\}}$ auquel ψ correspond, autant des transitions qu'il y a des l-uplets $([X_1 = t'_1], \ldots, [X_l = t'_l])$ dans S_V. En fin de compte, si l'on a m Ψ-termes clos dans S auquels ψ peut correspondre, alors nous aurons $m \times \sharp(S_{X_1}) \times \cdots \times \sharp(S_{X_l})$ transitions. Mais nous savons qu'il n'y aura qu'une transition franchissable, simplement puisqu'étant donné le caractère exclusif dans le placement de tokens pour les places liées à une variable, il y a un seul l-uplet $([X_1 = t'_1], \ldots, [X_l = t'_l])$ dont tous les composants seront marqués. Et si A est un réseau totalement autonome, i.e. ses places n'appartiennent à aucun autre réseau, alors ce l-uplet sera $([X_1 = \bot], \ldots, [X_l = \bot])$ comme l'indique la définition 4.2.1. Par la définition 4.2.9, nous pouvons voir que l'existence d'un l-uplet tel que décrit ci-dessus ne suffit pour le franchissement d'une transition de l'ensemble T, il faut, en plus, un jeton dans la place de lancement $[\mathtt{ask}(\psi)]$ et un jeton dans une place $[\psi']$ dont ψ soit une correspondance. Le premier est donné par le marquage initial. Ainsi donc avec l'existence d'un Ψ-terme clos dont ψ est une correspondance, le franchissement sera possible. En y regardant de plus près, on comprend que le fait d'avoir les places $[X_1 = t'_1], \ldots, [X_l = t'_l]$ dans le pré-ensemble des transitions, et les places $[X_1 = t_1], \ldots, [X_l = t_l]$ dans le post-ensemble des mêmes transitions permet de faire la mise à jour des bindings, en « renouvellant »les valeurs des variables de communications figurant dans ψ. C'est bien ce qui est exprimé la règle **(A)** de la section 2.2 par $\langle \mathtt{ask}(\Psi) \mid (\theta, \sigma \cup \{\Psi_c\})\rangle \longrightarrow \langle E \mid (\theta\mu, \sigma \cup \{\Psi_c\})\rangle$.

Pour les points **(iii)** et **(iv)**, on fait un raisonnement analogue. La composition séquentielle est aussi relativement facile à comprendre.

(vi) La composition parallèle proposée dans la définition 4.2.14 nous met en face de deux réseaux que l'on veut exécuter parallèlement. Selon cette définition, les places de ces deux réseaux sont mises ensemble, en plus, on ajoute deux places qui seront la place de lancement et la place finale du réseau global. Les transitions respectives de deux réseaux sont aussi gardés intactes. Ceci veut simplement que nous ne touchons pas aux séquences d'exécution de deux agents en présence. Nous ajoutons des transitions qui rendent possible l'exécution parallèle. En fait, la règle **(P)** de la sémantique opérationnelle originelle de $\mathcal{L}^r_\Psi$ nous dit :

si l'on sait que l'exécution d'un agent A sous la situation (θ, σ) conduit à un agent A' sous la situation (θ', σ')

alors l'exécution parallèle de l'agent A avec un autre agent B sous la

situation (θ, σ) conduit à l'exécution parallèle de A' et B sous la situation (θ', σ').

En des termes plus simples, si l'on exécute la première étape de la séquence d'exécution de A, on est encore en mesure d'exécuter la séquence de B. Dans les réseaux de Petri, nous avons traduit ce mécanisme en nous appuyant sur les transitions de lancement (i.e. celles qui ont une place de lancement en entrée) des réseaux de A et B respectivement. Nous définissons donc autant des nouvelles transitions qu'il y a des transitions de lancement dans l'ensemble de deux réseaux. Pour une transition de lancement t de A, par exemple, nous définissons une nouvelle transition qui se distingue de t par deux choses. D'abord, par le pré-ensemble qui change de $\{[s_{\mathcal{L}_A}]\}$ à $\{[A\|B]\}$. Ensuite, par le post-ensemble dans lequel on ajoute, en plus de $t^\bullet$, $s_{\mathcal{L}_B}$, la place de lancement du réseau B. Cette façon de faire garantit qu'après le franchissement de cette nouvelle transition, nous pouvons exécuter le réseau associé à l'agent B. Et nous construisons de la même façon une nouvelle transition pour transition de lancement t' de B. Ce qui traduit bien la règle **(P)** de la section 2.2. Et pour faire complet, nous ajoutons, en dernier lieu, une transition qui relie les places terminales de deux réseaux à la nouvelle place terminale. Finalement, nous avons construit $\sharp(T_{\mathcal{L}_A}) + \sharp(T_{\mathcal{L}_B}) + 1$ nouvelles transitions. Tout ce que nous venons de décrire peut être observé dans la figure 4.3 associée à l'exemple 4.2.2.

(vii) La composition alternative ou l'opérateur de *choix* entre deux agents permet, contrairement à la composition parallèle, de choisir, une fois pour toutes, la branche à exécuter. Afin de permettre une certaine équité dans le choix de cette branche à exécuter, la définition 4.2.15 propose mécanisme de composition des réseaux que l'on peut appeler *fusion par le sommet.* En fait, contrairement aux deux compositions précédentes où les deux réseaux de départ sont gardés intacts, nous procédons ici tout autrement. En effet, si nous voulons construire un réseau par l'opérateur de choix à partir de deux réseaux associés respectivement à l'agent A et à l'agent B, nous considérons toutes leurs places hormis leurs places de lancement respectives, mais en ajoutant une nouvelle place de lancement et une nouvelle place terminale. Et pour chaque transition de lancement t de deux réseaux, nous construisons une nouvelle transition qui ne diffère de t que par le fait que son pré-ensemble est le singleton contenant la nouvelle place de lancement. Nous supprimons, ensuite, toutes ces transitions de lancement de deux réseaux de depart. Les transitions de lancement du nouveau réseau sont donc ces nouvelles transitions que nous avons construites à partir des anciennes que nous venons de supprimer. Et la dernière transition à construire, selon la définition 4.2.15,

est bien celle qui relie les deux anciennes places finales à la nouvelle place finale. Finalement, la démarche a consisté à supprimer les anciens "sommets", aussi bien du côté des places que des transitions, et de les remplacer par des nouvelles. La figure 4.4 peut nous aider à visionner comment se construit le nouveau réseau. Puisque le marquage initial positionne un jeton à la place de lancement $[A + B]$, les actuelles transitions de lancement, construites à partir des anciennes transitions de lancement de deux réseaux, se situent indistinctement au même niveau et ont la même chance d'être tirées. Ce qui traduit bien la règle **(C)** de la section 2.2. •

Nous avons établi par ce théorème que la sémantique opérationnelle en termes des réseaux de Petri qui est défini dans ce chapitre est équivalente à celle présenté dans les règles de transition au chapitre 2.

Chapitre 5

INA : un outil d'analyse des réseaux de Petri

Le langage $\mathcal{L}^r_\Psi$, comme tous les langages de coordination, peut être utilisé pour la conception des systèmes distribués et concurrents. La vérification des propriétés de ces systèmes n'est pas toujours une tâche aisée. Ainsi, on fait souvent appel à certaines méthodes qui existent, dont les *méthodes formelles*, construites sur des solides et rigoureuses bases mathématiques. Parmi les méthodes formelles, on trouve le *model checking* qui peut, en quelques mots, être compris comme "une collection des techniques pour une vérification formelle automatique des systèmes concurrents à états finis" [20]. Dès lors que le choix a été fait, dans le cadre de ce travail, d'utiliser les réseaux de Petri pour exprimer la sémantique de $\mathcal{L}^r_\Psi$, il devenait impérieux de trouver un outil d'analyse des réseaux qui propose un éventail assez large d'analyses pour la vérification des propriétés.

Dans ce chapitre, nous allons présenter l'outil que nous avons choisi pour analyser les réseaux que nous construisons en utilisant la sémantique opérationnelle définie dans le chapitre précédent. Nous allons aussi rapidement expliquer les raisons qui ont prévalu dans le choix de cet outil.

5.1 Motivation et choix de INA

Le tableau synoptique présenté dans [36] nous a permis de nous faire une idée sur les outils existant pour l'analyse des réseaux de Petri. Notre intérêt s'est très vite porté sur trois outils : Design/CPN [1], PEP [2] et INA [40]. Tous ces outils sont, moyennant quelques conditions à remplir pour Desing/CPN notamment, gratuits.

Design/CPN, le premier outil ci-dessus cité, présentait, par rapport à l'usage futur que nous voulions en faire, quelques désavantages. Il est très orienté réseaux de Petri colorés (CPN). Les réseaux que nous concevons dans ce travail ne le sont pas. Ensuite, le model checking n'était pas parmi les analyses qu'il offrait. Enfin, sa portabilité était limitée à certaines plates-formes.

L'outil le plus complet, à notre avis, est assurément le PEP (comme **P**rogramming **E**nvironment based on **P**etri nets). Il offre, en plus d'un éditeur graphique, un plus grand éventail d'analyses sur les réseaux, dont le model checking. Il offre aussi une interface vers SMV, INA ou SPIN. PEP analyse aussi bien des réseaux de Petri simples que les réseaux de Petri temporisés. Il offre, en outre, un générateur des réseaux de Petri et, depuis peu, il offre également une interface avec le PNML (**P**etri **N**et **M**arkup **L**anguage) qui est un format de fichiers des réseaux de Petri basé sur le XML. Mais, étant donné, d'une part, la nature simple des analyses que nous pensions effectuer sur les réseaux ; et, d'autre part, le manque de portabilité de certains fichiers de PEP, nous avons opté pour un outil plus léger, facile d'installation et dont les fichiers des réseaux sont portables sur la plupart des plates-formes.

Integrated **N**et **A**nalyzer, en sigle INA, est l'outil que nous avons finalement choisi pour analyser nos réseaux de Petri. C'est un outil conçu sous la direction du professeur Peter H. Starke, à l'Institut für Informatik, Humbolt-Universität zu Berlin. La première ébauche a vu le jour en 1993, et c'est la version 2.2, sortie en 2003, que nous avons utilisée dans le cadre de ce travail.

5.2 Présentation de INA

INA est présenté comme un des outils les plus efficaces dans l'analyse des réseaux de Petri. Bien qu'il ne soit pas muni d'un éditeur graphique, il offre, à l'instar de bien d'outils, différents types d'analyses allant de l'analyse des espaces d'états à l'analyse structurelle, en passant par le model checking selon la logique CTL. Il analyse aussi bien les réseaux de Petri simples que les réseaux de Petri temporisés et les réseaux de Petri colorés. Dans le cadre de ce travail, nous n'avons nullement l'intention de nous étendre indéfiniment sur les qualités et les défauts de INA, ou encore sur toutes les analyses qu'il propose, mais nous voulons montrer au lecteur quelques éléments qui nous ont servis dans l'application que nous avons écrite et que nous présentons au chapitre suivant. Le lecteur désireux de mieux faire connaissance avec INA peut se référer à [40].

5.2.1 Fonctionnement de INA

L'installation de INA est des plus aisées. Quant à l'utilisation, elle est sensiblement facilitée par un menu interactif qu'offre l'outil. Bien qu'assez sommaire (pas graphique), ce menu permet d'accéder aux analyses et autres services que propose INA.

Au lancement de l'outil, ce qu'il convient d'appeler une *session* est ouverte. Dès cet instant jusqu'au moment où l'utilisateur quitte l'outil ou mieux ferme sa session, toutes ses commandes et/ou toutes ses demandes sont enregistrées dans un fichier, comme nous verrons plus loin. En ce moment de lancement, il est demandé à l'utilisateur s'il désire refaire la dernière procédure qui a été faite avec l'outil. Répondre affirmativement à cette question fera que l'outil reproduira toutes les commandes ou autres demandes qui ont été exécutées à la dernière session de INA et sur le même réseau que précédemment. Et s'il répond négativement, il accède alors au menu où il a la possibilité de faire son choix comme nous pouvons le voir dans la figure 5.1 ci-dessous.

C'est, par exemple, en appuyant sur "O" que l'utilisateur a la possibilité de changer les options et pouvoir dire s'il veut construire un réseau de Petri coloré ou non, s'il veut définir une certaine priorité entre les transitions pour le franchissement, s'il veut une règle de franchissement particulière, ...

Une des choses dont il faudrait bien faire attention, à propos du menu de INA, est qu'il y apparaît souvent les propositions où l'utilisateur a le choix entre 'Y'(pour 'yes') et 'N' (pour 'no'), puisque INA est ignore la casse du caractère, toute réponse en déhors de ces quatres caractères, 'Y', 'y', 'N' et 'n', est assimilée à 'no'.

INA fonctionne en deux types de modes, un mode *par défaut* et un mode *terminal*. La principale caractéristique du premier mode est que le réseau à analyser est présenté sous forme d'un fichier, éventuellement construit par l'utilisateur ou simplement lors d'une précédente session. Quant au mode terminal, il permet à l'utilisateur, comme son nom l'indique, de 'saisir' tout son réseau *en ligne*, et ensuite le soumettre, comme pour le mode par défaut, à l'analyse. Lorsque l'outil est en mode par défaut et attend un fichier en entrée, le fait de taper la touche <esc> le fait basculer en mode terminal. Et pour construire un réseau en mode terminal, l'utilisateur a le choix entre une syntaxe *orientée-place* ou une syntaxe *orientée-transition*. Avec la syntaxe orientée-place, le réseau est saisi place après place dans le format BNF tel

```
------------------------------------------------------------------

>>>>>>>>>>>  Welcome to the Integrated Net Analyzer! <<<<<<<<<<<
Version 2.2              Jul 31 2003          Peter Starke, Berlin
.....................Same procedure as last time? Y/N N

Current net options are:
    token  type: black        (for Place/Transition nets)
    time option: no times
    firing rule: normal
    priorities : not to be used
    strategy   : single transitions
    line length: 80

Do You want to
    edit ? ....................................E
    fire ? ....................................F
    analyse ? .................................A
    reduce ? ..................................R
    read the session report ? .................S
    delete the session report ? ...............D
    change options ? ..........................O
    quit ? ....................................Q
 choice > O
token  type:
    black (indistinguishable) tokens...........B
    coloured tokens............................C
token type := B
------------------------------------------------------------------
```

FIG. 5.1 – Une image du lancement de INA sur un terminal DOS.

que présenté dans la figure 5.2.

```
-------------------------------------------------------------
Input syntax:
<nr>" " <tokens>" "[ <prelist> ] [ "," <postlist> ] "<cr>"
 <prelist> ::= { <nr> [ ":"<mult> ]" " }
<postlist> ::= { <nr> [ ":"<mult> ]" " }
P>
-------------------------------------------------------------
```

FIG. 5.2 – La syntaxe des places.

Dans cette figure :
- `<nr>` désigne le numéro que l'utilisateur donne à cette place. Ce numéro étant suivi par un espace blanc ;
- `<tokens>` désigne le nombre des tokens que porte cette place ;
- `<prelist>` désigne, comme on peut le comprendre intuitivement, la liste des pré-transitions de cette place. Chaque transition étant représentée par un numéro et une multiplicité (`<mult>`), les deux séparés par un double-point (:) ;
- `<postlist>` désigne de manière logique la liste des post-transitions de cette place. Les transitions étant représentées comme ci-dessus.

Nous pouvons aisément comprendre que pour la syntaxe orientée-transition, c'est la transition qui est l'élément central dans la définition du réseau. Elle est alors définie avec son pré-ensemble ou ses *pré-places* et son post-ensemble ou ses *post-ensembles*. Avant de définir ainsi les places de son réseau, l'utilisateur doit préalablement donner un nom et un numéro à son réseau. Les noms des réseaux, comme ceux des places et des transitions, que l'utilisateur devra saisir après l'étape précédente, ne doivent pas dépasser 16 caractères.

5.2.2 Les principaux fichiers de INA

Comme nous avons eu à le dire plus haut, INA offre, entre autres choses, la possibilité d'échanger et de sauvegarder l'information concernant un réseau à travers plusieurs types de fichiers. Dans cette section, nous allons voir un peu plus en détail certains de ces fichiers, et particulièrement le fichier qui contient la définition même du réseau.

Pour les raisons d'analyse, INA peut avoir à créer ou utiliser plusieurs types de fichiers (une bonne vingtaine). Nous allons nous attarder sur certains types de ces fichiers. Il peut être, à ce niveau, utile de savoir qu'un réseau que l'on veut garder, pour une éventuelle utilisation ultérieure, est sauvegardé dans un fichier qui porte une extension `.cnt` si le réseau est coloré et `.pnt` s'il ne l'est pas. Il est, par exemple, possible de garder dans un fichier la trace de toutes les séquences de franchissement pour un réseau donné. Un tel type de fichier porte l'extension `.mar` comme marquage. Mais les fichiers les plus importants lors d'une session sont `OPTIONS.ina`, `COMMAND.ina` et `SESSION.ina`.

(a) `OPTIONS.ina` est le fichier dans lequel INA sauvegarde toutes les options que vous avez choisies durant votre session. Elles vont du type du réseau (coloré ou non) à la stratégie de franchissement des transitions, en passant par la profondeur des graphes d'accessibilité. A l'ouverture d'une nouvelle session, les options de la précédente sont encore proposées et il appartient à l'utilisateur de les changer à travers le menu proposé et selon ses besoins.

(b) `COMMAND.ina` est un fichier créé à la fin d'une session et sur demande de l'utilisateur. En effet, lorsque l'utilisateur ferme sa session, INA lui demande s'il désire sauvegarder le fichier `COMMAND.ina`. C'est dans ce fichier que sont enregistrées toutes les commandes entrées par l'utilisateur durant la session. Et à la prochaine ouverture d'une nouvelle session, INA demande si l'utilisateur voudrait faire la même séquence de commandes que lors de la dernière session :

```
Same procedure as last time? Y/N
```

à la réponse `'Y'` de l'utilisateur, les commandes enregistrées dans `COMMAND.ina` sont exécutées. Autrement dit, la session précédente est répétée. Et pendant ce temps, l'utilisateur appuie sur la touche `<h>` (comme halt), INA bascule en mode terminal.

(c) `SESSION.ina` est le fichier dans lequel INA écrit tous les résultats et déductions obtenus durant une session. Ce fichier est réécrit à l'ouverture de la prochaine session. Mais, si l'utilisiteur estime utile de garder les résultats relatifs à l'analyse d'un réseau donné, INA lui donne la possibilité de renommer le fichier `SESSION.ina` à la sortie de la session.

INA prend la précaution de sauvegarder dans un fichier auxiliaire, nommé `ININET.pnt`, le réseau initial avant toutes les réductions initiées par l'utilisateur lors d'une session. En effet, INA travaillant directement sur le réseau qu'on lui soumet, il peut s'avérer quelques fois utile de retrouver le réseau de

départ puisqu'il n'est pas forcément le même à la sortie de la session.

Hormis les fichiers susmentionnés, INA peut créer ou utiliser beaucoup d'autres fcihiers encore. Leur nombre et leur variété dépendent selon la finesse des analyses que l'utilisateur veut entreprendre sur le réseau. Il est, par exemple, possible de vérifier la validité d'un prédicat pour une place ou pour une transition. Ce prédicat peut être saisi en ligne (mode terminal), ou être contenu dans un fichier. Il est important de se référer au manuel [40] pour voir la syntaxe dans laquelle les prédicats sont écrits. On comprendra, par exemple, qu'un prédicat sera toujours représenté comme une *forme normale disjonctive*. Un fichier contenant de prédicats porte l'extension `.pdc`. De même, pour faire le model checking, l'utilisateur peut placer toutes les formules-CTL dont il veut tester la validité dans un fichier `.ctl` ou saisir chaque formule en ligne. Dans ce cas aussi, un coup d'oeil au manuel peut aider à respecter la syntaxe acceptée par INA.

5.2.3 Un réseau dans un fichier

Un des avantages pour lesquels nous avons choisi INA est bien le fait qu'il nous permette d'échanger les informations concernant un réseau à travers de fichiers, en l'occurrence des fichiers `.pnt`, puisque nous avons choisi de travailler avec des réseaux simples, et non colorés [1]. L'essentiel est alors de créer ou de préserver un tel fichier selon la syntaxe des fichiers de réseaux acceptée par INA. En fait, l'utilisateur peut modifier les informations se trouvant dans un tel fichier, il reste valable tant qu'il respecte la syntaxe. La figure 5.3 donne justement la syntaxe INA d'un fichier contenant la description d'un réseau de Petri.

Dans cette description d'un fichier `.pnt`, nous pouvons voir qu'il est divisé en trois parties dont le caractère `"@"` constitue le délimiteur. La première partie contient, d'une part, l'identité du réseau (son numéro et son nom qui le distinguent des autres réseaux du même répertoire) ; et d'autre part, la structure du réseau. Dans cette sous-partie, chaque place, identifiée par son numéro, est présentée avec ses pré-transitions et ses post-transitions, le nombre de tokens qu'elle porte ; et chaque transition est décrite avec son numéro et la multiplicité de l'arc qui la relie à la place pour laquelle elle est pré ou post-transition. Cette multiplicité peut être omise si elle vaut 1. L'ensemble de pré-transitions ou des post-transitions d'une place peut aussi être vide.

[1] On parlerait des fichiers `.cnt` s'il s'agissait des réseaux de Petri colorés

```
-----------------------------------------------------------------------
<pntfile>::= <netheader> "<cr>"
<netstruct> "<cr>"
"@<cr>"
<placedata> "<cr>"
"@<cr>"
<transdata> "<cr>"
"@<cr>"

<netheader>::= "P   M    PRE,POST    NETZ" <netid>
    <netid>::= <netnr> [ ":" <netname> ]
    <netnr>::= <number>
  <netname>::= <name>
<netstruct>::= <placedef> { "<cr>" <placedef> }
 <placedef>::= { " " <placenr>" "<tokens>"  "[ <prelist> ]
                [ ", " <postlist> ]}
  <placenr>::= <number>
   <tokens>::= <number>
  <prelist>::= { <transnr> [ ": " <arcmult> ] " "}
 <postlist>::= { <transnr> [ ": " <arcmult> ] " "}
  <transnr>::= <number>
  <arcmult>::= <number>
<placedata>::= "place nr.   name   capacity   time"
               { "<cr>  " <placenr> ": " <placename> "   "
               <capacity> <time> }
<placename>::= <name>
 <capacity>::= " " <number> | "oo"
     <time>::= <number>
<transdata>::= "trans nr.   name   priority   time"
               { "<cr>  " <transnr ": " <transname> "   "
               <priority> <time> }
<transname>::= <name>
 <priority>::= <number>
-----------------------------------------------------------------------
```

FIG. 5.3 – La syntaxe d'un fichier .pnt.

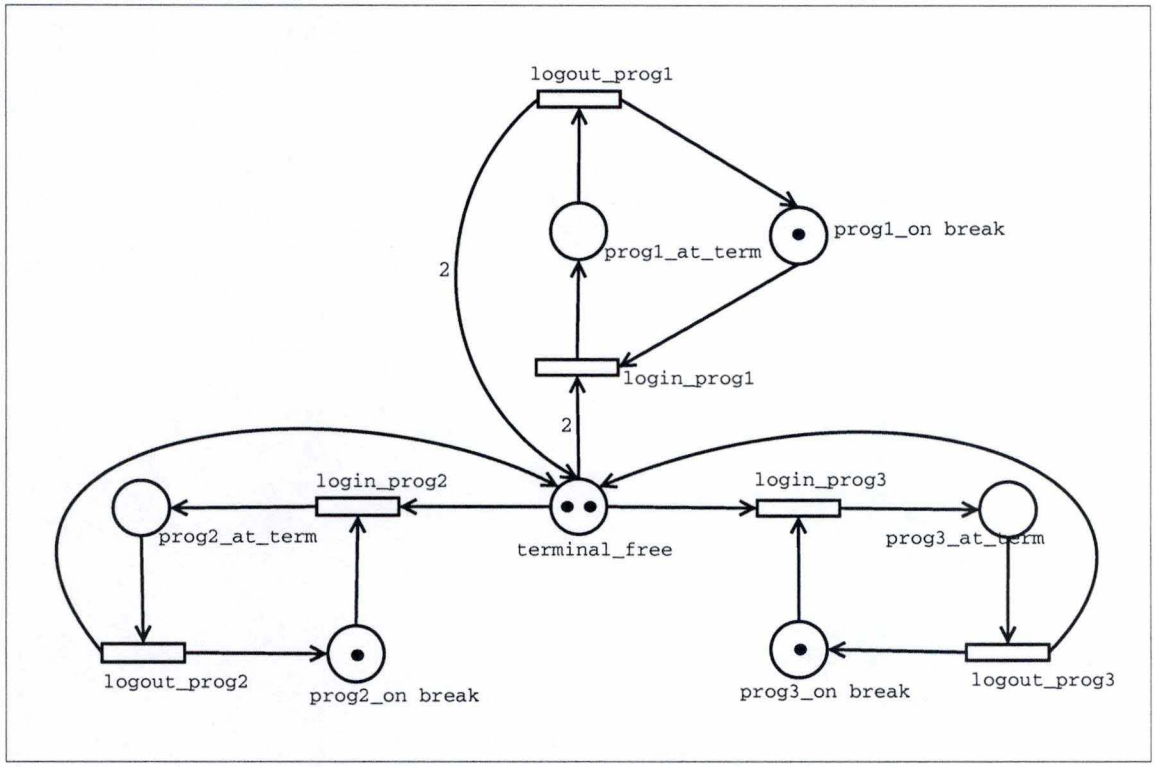

FIG. 5.4 – Réseau représentant l'activité des trois programmeurs qui partagent deux terminaux.

L'exemple suivant tiré de [40] nous aidera à mieux comprendre cette représentation. Cet exemple rappelera vaguement au lecteur le problème du dîner des philosophes.

Exemple 5.2.1

Soient trois programmeurs qui doivent se partager deux terminaux pour leur travail. Le programmeur 1 a besoin de deux terminaux pour son travail. Ainsi, cette contrainte fait que si le programmeur 1 travaille, alors il est le seul à le faire, les deux autres sont au repos. Et si les deux autres programmeurs travaillent, alors le programmeur 1 devra attendre qu'ils soient tous les deux au repos pour reprendre du service. En termes des réseaux de Petri, la situation peut être modélisée comme dans le réseau dans la figure 5.4. Nous pouvons remarquer qu'au départ nos trois programmeurs sont en repos et les deux terminaux sont disponibles.

Le fichier `trois_progs.pnt` sur la figure 5.5 est la représentation que

l'on obtient de INA après que l'on ait introduit les éléments du réseau sur la figure 5.4 .

```
----------------------------------------
P   M   PRE,POST  NETZ 1:3_Prog_2Term
  0 2     4: 2 5 6, 1: 2 2 3
  1 0     1, 4
  2 0     2, 5
  3 0     3, 6
  4 1     4, 1
  5 1     5, 2
  6 1     6, 3
@
place nr.              name capacity time
       0: terminal_free          oo    0
       1: prog1_at_term          oo    0
       2: prog2_at_term          oo    0
       3: prog3_at_term          oo    0
       4: prog1_on_break         oo    0
       5: prog2_on_break         oo    0
       6: prog3_on_break         oo    0
@
trans nr.              name priority time
       1: login_prog1             0    0
       2: login_prog2             0    0
       3: login_prog3             0    0
       4: logout_prog1            0    0
       5: logout_prog2            0    0
       6: logout_prog3            0    0
@
----------------------------------------
```

FIG. 5.5 – Le fichier `trois_progs.pnt` représentant le réseau du problème de trois programmeurs.

C'est ce type de fichiers que produit l'application que nous présentons dans chapitre suivant.

Chapitre 6

Application : Net File Generator, un générateur des fichiers de réseaux de Petri

Ce chapitre présente l'application que nous avons conçue afin de générer automatiquement les fichiers relatifs aux réseaux de Petri. Ces fichiers sont conformes à la syntaxe acceptée par INA. Le processus de génération part des agents de coordination décrits selon le formalisme présenté au chapitre 2. A l'aide d'une application de type compilateur, nous obtenons en sortie un fichier de réseau (`.pnt`) pouvant être traité par INA.

Etant donné l'option que nous avons levée d'écrire cette application en Java, nous avons utilisé JLex [11] et Cup [31] comme respectivement générateur d'analyseur lexical et générateur de parseur pour Java. Ce choix d'avoir une application complètement écrite en Java a eté, pour nous, la seule raison de préférer ces outils aux célèbres Lex et Yacc. JLex et Cup sont, comme leurs fameux précédesseurs, des *open source* et gratuitement téléchargeables sur Internet. Ils sont entièrement écrits en Java et produisent respectivement de lexeurs et des parseurs en Java. Dans les sections qui suivent, nous présentons brièvement ces deux outils.

6.1 Une brève présentation de JLex

Exécuter une instruction ou reconnaître un commentaire dans un langage donné, nécessite que l'on distingue tous les éléments constitutifs de ce langage, à commencer par ses entités minimales, appelées lexèmes. C'est le travail que fait un analyseur lexical. Il « tronçonne » un flot de caractères

en lexèmes. Ecrire un analyseur lexical « à la main »peut s'avérer être une tâche ardue. Vu le caractère systématique du processus de reconnaissance des lexèmes pour un langage, des outils ont été créés pour rendre ce procédé plus aisé. Assurément, le plus connu des outils le plus aptes à cette tâche est le Lex [34].

JLex, générateur d'analyseur lexical, est écrit totalement en Java et se base sur le modèle de Lex. Il accepte un fichier de spécifications similaire à celui de Lex et crée un fichier source Java correspondant à l'analyseur lexical. Nous avons utilisé la version 1.2.6 de JLex dans le cadre de ce travail.

6.1.1 Les spécifications de JLex

Un fichier de spécifications de JLex est organisé en trois parties, délimitées par les *directives* double-pourcent ("%%") et se présente comme dans la figure 6.1 .

```
---------------------------------
user code
%%
JLex directives
%%
regular expression rules
---------------------------------
```

FIG. 6.1 – Le format d'un fichier de spécifications de JLex.

Dans la première partie, « user code », l'utilisateur écrit tout ce qu'il veut voir apparaître tel quel dans le code source de l'analyseur lexical. Et cela apparaîtra au sommet du fichier. Ainsi, par exemple, si l'utilisateur veut déclarer un package ou faire un « import »d'une classe externe dans son analyseur lexical, cette partie devrait commencer par cette déclaration ou cet « import ».

La deuxième partie, i.e. celle qui vient juste après la première directive "%%", est un peu la partie fourre-tout du fichier de spécifications. Elle peut aussi bien contenir :

- un code interne à l'analyseur lexical. En fait, avec la directive *%{ ...%}*, l'utilisateur peut écrire un code Java qui sera copié dans la classe de

l'analyseur lexical. Ainsi, par exemple, si l'utilisateur écrit

```
%{
 <code>
%}
```

alors, en supposant que la classe de l'analyseur porte le nom par défaut *Ylex.java*, nous aurons quelque part dans cette classe

```
class Ylex{
    ...
    <code>
    ...
}
```

Cette façon de faire donne à l'utilisateur un moyen de définir des variables et des méthodes internes à la classe de l'analyseur lexical. L'utilisateur devra néanmoins éviter les noms de variables commençant par *yy* puisqu'ils sont réservés pour un usage spécifique par la classe de l'analyseur généré. En plus, JLex exige que *%{* et *%}* soient placés au début de la ligne où ils figurent pour qu'ils soient reconnus dans leurs rôles de délimiteurs d'une directive ;

- la directive *% init { ...% init }* qui permet à l'utilisateur de d'écrire du code qui sera copié dans le constructeur de la classe de l'analyseur lexical. Si l'utilisateur écrit :

```
%init{
      <code>
%init}
```

alors dans le constructeur de l'analyseur lexical *Ylex.java*, cela sera rendu comme suit :

```
class Ylex{
    Ylex(){
        ...
        <code>
        ...
    }
    ...
}
```

Cette directive permet à l'utilisateur de faire, une fois pour toutes, les initialisations dans le constructeur de l'analyseur lexical. Et comme précédemment, l'utilisateur devra placer *%init{* et *% init }* au début

des lignes où ils apparaissent. Il se peut que le code placé dans *% init{ ...% init }* lance une exception ou en propage une venant d'une autre fonction. Dans ce cas, l'utilisateur doit utiliser la directive *% initthrow { ...% initthrow}* pour déclarer cette exception. Les choses se présentent alors comme suit :

```
%initthtrow{
     <exception[1]>[, <exception[2]>, ....]
%initthrow}
```

et cela apparaît dans le constructeur de la classe de l'analyseur lexical de la manière suivante :

```
Ylex()
     throws <exception[1]>[, <exception[2]>, ...]{
     ...
     <code>
     ...
}
```

Et si le code Java dans la directive *% init{ ...% init }* lance une exception qui n'est pas déclarée alors l'analyseur lexical généré par ces spécifications peut ne pas compiler correctement ;

- la directive *%eof{ ...%eof}* qui permet à l'utilisateur d'écrire du code qui sera copié dans l'analyseur lexical et qui sera exécuté lorsque ce dernier atteindra la fin d'un fichier. Les exceptions sont gérées de la même façon que pour la précédente directive ;
- les états lexicaux *%state state[0][, state[], state[], ...]* qui sont définis pour le contrôle lorsqu'il y a matching avec certaines expressions régulières. Un seul état lexical est défini implicitement par JLex, il s'agit de *YYINITIAL*. L'analyseur lexical généré commence justement son analyse en cet état ;
- une directive pour définir la compatibilité de JLex et de Cup, *%cup*. Cette compatibilité doit être établie de manière explicite, puisqu'elle n'existe pas par défaut. En fait, JLex est en mesure de travailler avec d'autres générateurs de parseurs, et s'il doit travailler avec Cup, c'est cette directive qui l'indique ;
- les directives *%class <name>*, *%function <name>* et *%type <name>* respectivement pour changer le nom de la classe de l'analyseur lexical (à la place du nom par défaut *Ylex*), pour changer le nom de la fonction qui reconnaît les lexèmes (à la place de *yylex* le nom par défaut), et pour changer le nom du type de l'objet retourné par la fonction qui reconnaît les lexèmes (*Ytoken* étant le nom par défaut de ce type).

Bien d'autres éléments encore peuvent être insérés dans cette deuxième partie du fichier de spécifications. Le manuel [11] peut servir pour celui qui veut en savoir plus.

La troisième partie du fichier de spécifications est, quant à elle, constituée des règles d'expressions régulières pour les différents lexèmes du flux d'entrée. En fait, ces règles spécifient les expressions régulières et les associent à des actions exprimées en code Java. Une règle est donc constituée de trois parties : une liste optionnelle d'états lexicaux, une expression régulière et une action associée. Elle se présente selon le format suivant :

$$[< states >] < expression > < action > \tag{6.1}$$

S'il arrive que plusieurs règles matchent des chaînes de caractères en entrée, alors le lexeur généré gère les conflits entre les règles en choisissant simplement la règle qui matche la plus longue chaîne. Et si plusieurs règles matchent des chaînes de caractères de même longueur, alors le lexeur va choisir parmi les règles en conflit celle qui est vient en premier dans les spécifications. Ainsi, plus haut apparaît une règle dans la liste des règles, plus grande est sa priorité. Il est important de savoir que toute entrée doit pouvoir matcher au moins une règle. Cela est garanti par le fait que l'utilisateur doir placer comme dernière règle de sa liste, une règle qui ressemble à la suivante :

```
. {java.lang.System.out.println("Unmatched input :"+yytext()) ;}.
```

Ceci voudrait simplement dire : quel que soit l'état dans lequel le lexeur se trouve, s'il rencontre un lexème qu'il ne reconnaît pas, qu'il le signale par un message. Le caractère point (".") exprime justement tout caractère à l'exception du caratère qui exprime une nouvelle ligne (\n). Et le fait de placer cette règle à la fin de toute la liste des règles signifie qu'elle porte la plus petite priorité qui soit.

Dans (6.1), la liste facultative *[<states>]* peut s'exprimer, comme dans la deuxième partie, *<state[0][, state[1], state[2], ...]*. Chaque règle commence donc par une liste facultative d'états. Si elle existe, cette liste précède l'expression régulière et spécifie les états dans lesquels la règle peut être matchée. Si aucune liste n'est spécifiée pour une règle, alors celle-ci peut être matchée dans tous les états lexicaux du lexeur.

Une expression régulière ne devra pas contenir un espace blanc, car celui-

ci est interprété comme la fin de l'expression régulière courante. A moins que ce caractère apparaisse entre guillemets. L'alphabet de JLex est l'ensemble des caractères Ascii, c'est à dire les caractères dont les codes sont entre 0 et 127 inclus. Si e et f sont deux expressions régulières, alors ef représente leur concaténation et $e|f$ représente le choix entre e et f. Les caractères

```
? * + | ( ) ^ $ . [ ] { } '' \
```

sont des méta-caractères et ont une signification spéciale dans une expression régulière de JLex.

Une action associée à une règle lexicale est composée d'un bloc de code Java délimité par des accollades. C'est, en principe, l'action que doit accomplir le lexeur s'il reconnaît l'entité exprimée dans l'expression régulière dans les états présents dans la liste. L'exemple suivant montre une partie du fichier de spécifications que nous avons écrit dans le cadre de ce travail et dont l'intégralité se trouve dans les annexes.

Exemple 6.1.1

Voici l'exemple d'un fichier de spécifications où nous pouvons distinguer les trois parties ainsi que d'autres éléments dont nous venons de parler :

```
package NFGTool;

import java.lang.*;
import java_cup.runtime.*;

class TokenValue {
     public String text;
     public Object first_object;
     public Object second_object;

     TokenValue() {
          this("", null, null);
     }

     TokenValue(String text){
          this(text, null, null);
     }

     TokenValue(Object first, Object second) {
```

```
            this("", first, second);
        }
}

%%
%cup
%class Lexer
%unicode

%state COMMENTS

NUMBER = [0-9]+
CHARACTER = [a-zA-Z] | [0-9] | [\ \\-_\'()<>]
WHITE_SPACE = ([\ \n\r\t\f])+
SEQ = ";"
PAR = "||"
ALT = "+"
SLASH = "/"

%%
<YYINITIAL> {WHITE_SPACE} {
}

<YYINITIAL> {NUMBER} {
    return new Symbol(sym.NUMBER, new MyNumber(yytext()));
}

<YYINITIAL> {CHARACTER} {
    return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}

<YYINITIAL> . {
    return new Symbol(sym.error, null);
}
```

Il est clair que l'on ne sait pas toujours totalement comprendre un fichier de spécifications d'analyse lexicale si l'on ne poursuit pas la démarche jusqu'à voir comment ces spécifications sont utilisées dans la construction du parseur. La section suivante nous parle du générateur de parseur que nous avons utilisé, Cup.

6.2 Une brève présentation de Cup

A l'instar de JLex construit sur le modèle de Lex, Cup, lui, est construit sur le modèle de Yacc [34]. Depuis la version 0.10 (celle que nous avons utilisée dans ce travail), les concepteurs de Cup prétendent même qu'il fait plus que son illustre prédécesseur en rendant encore plus facile l'écriture des spécifications. Et comme Yacc, Cup (Java Based **C**onstructor of **U**seful **P**arsers) est donc un système pour générer des parseurs LALR[1] à partir des simples spécifications. Il est écrit en Java, utilise des spécifications incluant Java et produit des parseurs implémentés en Java.

L'utilisation de Cup revient à écrire des spécifications sur base d'une grammaire pour laquelle un parseur est nécessaire. Prenons un petit exemple, tiré de [31], où l'on considère un système qui doit lire des expressions arithmétiques sur les entiers, les évaluer et imprimer les résultats. On suppose que chaque expression se termine par un point-virgule (" ;"). Une grammaire pour une entrée d'un tel système ressemblera à ce qui est présenté dans la figure 6.3.

```
-------------------------------------------------------------

expr_list ::= expr_list expr_part | expr_part
expr_part ::= expr ';'
expr      ::= expr '+' expr | expr '-' expr | expr '*' expr
      | expr '/' expr | expr '%' expr | '(' expr ')'
      | '-' expr | number

-------------------------------------------------------------
```

FIG. 6.2 – Un exemple de grammaire qui évalue des expressions arithmétiques simples sur des entiers.

Afin de spécifier le parseur qui se base sur cette grammaire, la première tâche sera d'identifier et de nommer tous les symboles *terminaux* qui peuvent apparaître dans le flot d'entrée ainsi que tous les symboles *non-terminaux*. A partir de l'exemple dans la figure 6.3, nous pouvons voir que les non-terminaux sont :

`expr_list, expr_part et expr.`

[1] **L**ook**A**head LR, et LR comme abréviation de **L**eft to right scanning of the input constructing a **R**ightmost derivation in reverse

Et pour les symboles terminaux, nous pouvons choisir :

`SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, NUMBER, LPAREN et LPAREN`

nous pouvons remarquer que la plupart des terminaux représentent les opérateurs utilisés dans l'évaluation des expressions, mais aussi le type « primitif »(`NUMBER`) d'éléments à manipuler.

Avant de revenir sur les symboles terminaux et non terminaux, disons un petit mot sur le fichier de spécifications de Cup.

6.2.1 Un fichier de spécifications Cup

Un fichier de spécifications Cup comprend quatre parties principales.

- La première partie comprend les déclarations préliminaires et diverses qui spécifient dans quelles conditions le parseur va être généré. C'est à ce niveau que l'on fait l'« import »du package `java_cup.runtime` qui contient l'essentiel des méthodes de génération des parseurs, et c'est aussi à ce niveau l'on peut fixer certaines choses pour l'initialisation ; ou encore spécifier la méthode du lexeur qui doit récupérer le prochain lexème en entrée. Le prochain exemple nous éclairera davantage à ce sujet.
- La deuxième partie déclare les terminaux et les non-terminaux, et associe des classes d'objets à chacun de ces deux ensembles. Un terminal ou non terminal peut être avec ou sans type. Si aucun type n'est spécifié pour un terminal ou un non terminal, cela veut simplement dire qu'il ne porte pas de valeur. La forme général pour les éléments de cette partie est :

  `terminal` *nom_de_classe nom1, nom2,...*;
  `non terminal` *nom_de_classe nom1, nom2, ...*;
  `terminal` *nom1, nom2, ...*;
  `non terminal` *nom1, nom2, ...*;

  où, bien sûr, *nom_de_classe* est le type des terminaux ou non-terminaux qui le suivent, et peut aussi prendre le nom *long* d'une classe Java.
- La troisième partie est optionnelle. Elle spécifie la précédence et l'associativité des terminaux. Ceci peut être fort utile pour les grammaires qui sont ambigues. La déclaration générale de précédence/associativité se fait sous une des trois formes suivantes :

`precedence left` *terminal[, terminal ...]*;
`precedence right` *terminal[, terminal ...]*;
`precedence nonassoc` *terminal[, terminal ...]*;

Cup considère tout terminal qui n'apparaît pas dans la liste des précédences comme ayant la précédence la plus faible. Et pour Cup, plus bas apparaît une précédence dans la liste, plus grande est la précédence de terminaux associés. Ainsi, les terminaux associés à la dernière précédence de la liste ont la plus grande précédence de toutes. L'associativité, elle, ne prend qu'une des trois valeurs (`left`, `right` et `nonassoc`). Par exemple, dans cette simple addition $3 + 5 + 7 + 9$, si l'associativité de $'+'$ est `left`, alors on additionne ces chiffres en allant de la gauche vers la droite.
Il est clair que la précédence peut aider à résoudre certains conflits *shift/reduce*. Par exemple, si l'on des précédences définies comme suit :

```
precedence left ADD, SUBSTRACT
precedence left TIMES, DIVIDE
```

l'expression $2 + 4 * 6$ qui serait ambigue sans la définition des précédences ci-dessus trouve facilement de réponse, puisque nous voyons que la multiplication (`TIMES`) a une précédence plus grande que l'addition (`ADD`).

- Et la quatrième et dernière partie des spécifications contient la grammaire. Elle peut optionnellement commencer par :

  `start with` *non-terminal*

  pour indiquer le non-terminal par lequel commencera le parsing. Il s'ensuit alors une liste de productions qui constituent la grammaire. Si la déclaration `start` n'est pas explicitement écrite, alors le non-terminal dans le membre gauche de la première production de la grammaire est considéré comme celui par lequel le parsing doit commencer. Chaque production a, dans son membre gauche, un non-terminal qui est suivi d'un symbole `''::=''`, et d'une série de zéro ou plusieurs actions. Une action se présente comme un bloc de code Java délimité par `{ : ... :}`.

L'exemple sur la figure 6.3 montre une liste de précédences suivie d'une production dans laquelle nous pouvons voir l'action qui doit être faite si le parseur reconnaît le `'moins'` unaire.

```
-------------------------------------------------------------
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;

expr ::= MINUS expr : e
         {: RESULT = new Integer(0 - e.intValue()); :}

-------------------------------------------------------------
```

FIG. 6.3 – Un fragment de spécification avec quelques précédences et une production de grammaire.

Dans cette section, nous avons voulu montrer un peu comment fonctionne Cup, comme nous l'avons fait dans la section précédente avec JLex. Mais il doit être clair que ce n'est qu'un survol rapide que nous avons donné. Celui qui voudrait mieux connaître Cup peut se référer au manuel [31].

Nous présentons à la figure 6.4 le fichier des spécifications Cup de l'exemple sur les expressions arithmétiques. Les expressions arithmétiques sont lues par l'entrée standard, sont évaluées (au moyen de quatre opérations élémentaires) et les résultats sont imprimées sur la sortie standard. On peut soumettre plusieurs expressions, à condition de les séparer les unes des autres par des points-virgules (" ;"). Le fait, par exemple, dans ce fichier de n'avoir pas explicitement utilisé la déclaration `start` fait que le non-terminal `expr_list` sera le premier à être parsé. Dans ce fichier, il est précisé aussi que le terminal `NUMBER` et le non-terminal `expr` sont du type `Integer`. Ce qui d'ailleurs facilite les évaluations, puisqu'on utilise les opérations prédéfinies sur les entiers dans Java.

Dans les annexes se trouvent le fichier des spécifications de notre application.

6.3 Présentation de NFG

En écrivant cette application, le but était simplement de *valider* les résultats théoriques que nous avançons dans les chapitres précédents. Pour rappel, NFG prend en entrée un agent exprimé dans le langage $\mathcal{L}^r_\Psi$ et le transforme en un fichier de réseau à même d'être analysé par INA.

```
package Example;

import java_cup.runtime.*;

parser code {:
     public static void main(String args[]) throws Exception {
           new parser(new Yylex(System.in)).parse();
     }
:}

terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, LPAREN, RPAREN;
terminal Integer NUMBER;

non terminal expr_list, expr_part;
non terminal Integer expr;

precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;

expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr:e {: System.out.println("="+e+";"); :}SEMI;
expr      ::= NUMBER:n
              {: RESULT=n; :}
              | expr:l PLUS expr:r
                {: RESULT=new Integer(l.intValue() + r.intValue()); :}
              | expr:l MINUS expr:r
                {: RESULT=new Integer(l.intValue() - r.intValue()); :}
              | expr:l TIMES expr:r
                {: RESULT=new Integer(l.intValue() * r.intValue()); :}
              | expr:l DIVIDE expr:r
                {: RESULT=new Integer(l.intValue() / r.intValue()); :}
              | LPAREN expr:e RPAREN
                {: RESULT=e; :}
              ;
```

FIG. 6.4 – Un fichier de spécifications Cup pour l'évaluation des expressions arithmétiques.

Dans un premier temps, nous avons défini les spécifications lexicales selon la syntaxe de JLex.

6.3.1 Esquisse d'architecture de NFG

Comme nous avons eu à le dire précédemment, NFG est une application qui génère, à partir des agents exprimés en $\mathcal{L}_{\Psi}^{r}$, des fichiers de réseau conformes à un format défini par INA. Pour rendre cette transformation possible, nous avons défini des spécifications lexicales pour permettre la reconnaissance des lexèmes utilisés dans $\mathcal{L}_{\Psi}^{r}$, nous avons ensuite défini les spécifications pour reconnaître la structure des agents tels qu'ils sont définis, et enfin, utilisant la sémantique opérationnelle que nous avons définie dans le chapitre 4, nous avons transformé les éléments de ces agents en éléments des réseaux de Petri. Nous donnons dans la figure 6.5 une esquisse d'architecture qui décrit succinctement le flux de processus de transformation.

Utilisant la vision orientée objet, nous avons défini une classe pour chacun tous terminaux et non-terminaux *significatifs*, mais également pour les différents types de tokens. Et chacune de ces classes a hérité d'une classe unique `TokenValue` qui regroupe les caractéristiques minimales d'un token tel que nous le définissons.

6.4 Du langage $\mathcal{L}_{\Psi}^{r}$ à INA

Cette section a pour but de montrer le cheminement que suit un *agent* exprimé dans le langage $\mathcal{L}_{\Psi}^{r}$ jusqu'à l'analyse du fichier de réseau de Petri associé en passant par les différentes étapes décrites dans ce travail. Pour ce faire, nous partons d'un agent exprimé en $\mathcal{L}_{\Psi}^{r}$ et nous voyons quelques analyses faites au moyen de INA sur le réseau obtenu àl'aide de NFG. Pour des raisons purement pratiques, nous avons ajouté le symbole dièse (« ♯ ») pour signifier la fin d'un agent. Considérons l'agent A :

```
(
  tell(f/4(x1 = 7, x2 = "essai", x3 = "travail", x4 = 1))
  ;
  get(f/4(x1 = 7, x2 = "essai", x3 = "travail", x4 = 1))
)
||
tell( f/3(x1="a",x2=9,x3=1))#
```

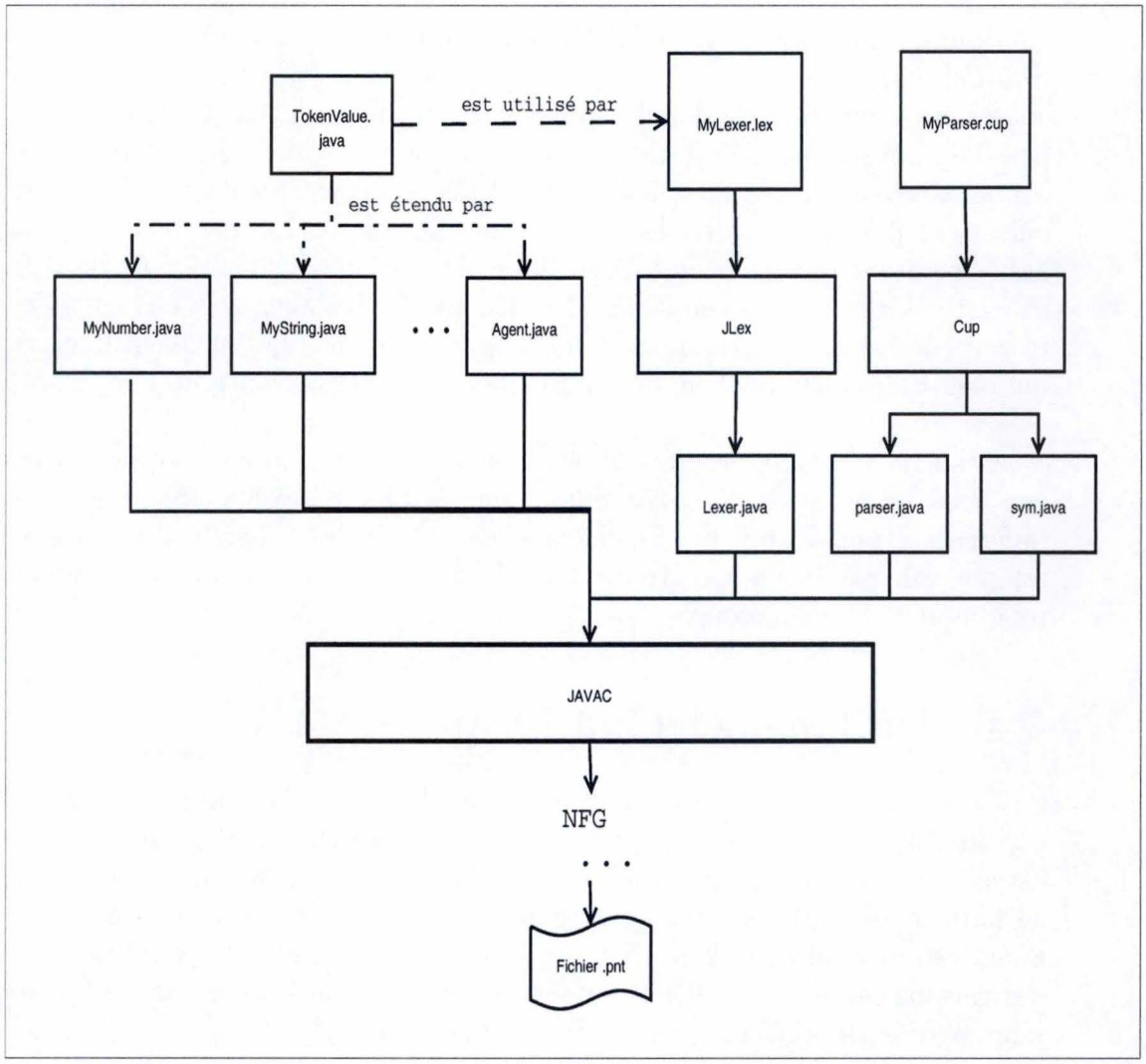

FIG. 6.5 – Une esquisse d'architecture de NFG basée sur le flux de traitement.

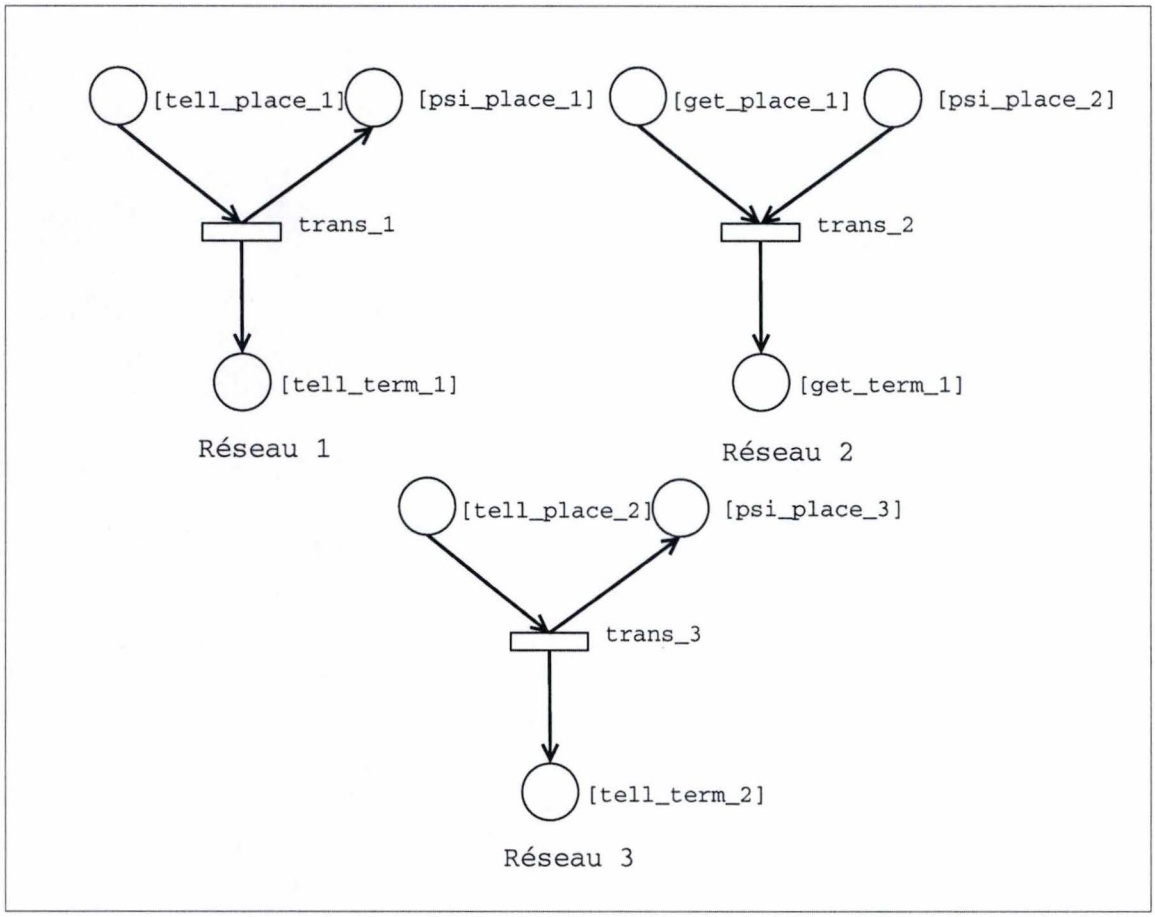

FIG. 6.6 – Les réseaux relatifs aux agents élémentaires composant l'agent A.

dans lequel nous avons d'abord une composition séquentielle et ensuite une composition parallèle. Si nous devons considérer de manière individuelle les réseaux relatifs aux trois agents élémentaires composant l'agent A, nous aurons la représentation graphique de la figure 6.6 dans laquelle les trois agents sont respectivement représentés par Réseau 1, Réseau 2 et Réseau 3. Par la composition séquentielle des deux premiers agents, et puisqu'ils partagent tous les deux le Ψ-terme `f/4(x1 = 7, , x2 = "essai", x3 = "travail", x4 = 1)`, la place de `psi_place_2` disparaît, et le réseau composé est représenté à la figure 6.7. Suivant la définition 4.2.13, nous pouvons remarquer que `tell_place_1` est la place de lancement de ce réseau composé, et `get_term_1` en est la place terminale. En faisant la dernière opération, qui est la composition parallèle, nous avons le réseau de la figure 6.8 qui représente le réseau de l'agent A construit selon la définition 4.2.14. Nous avons renommé la

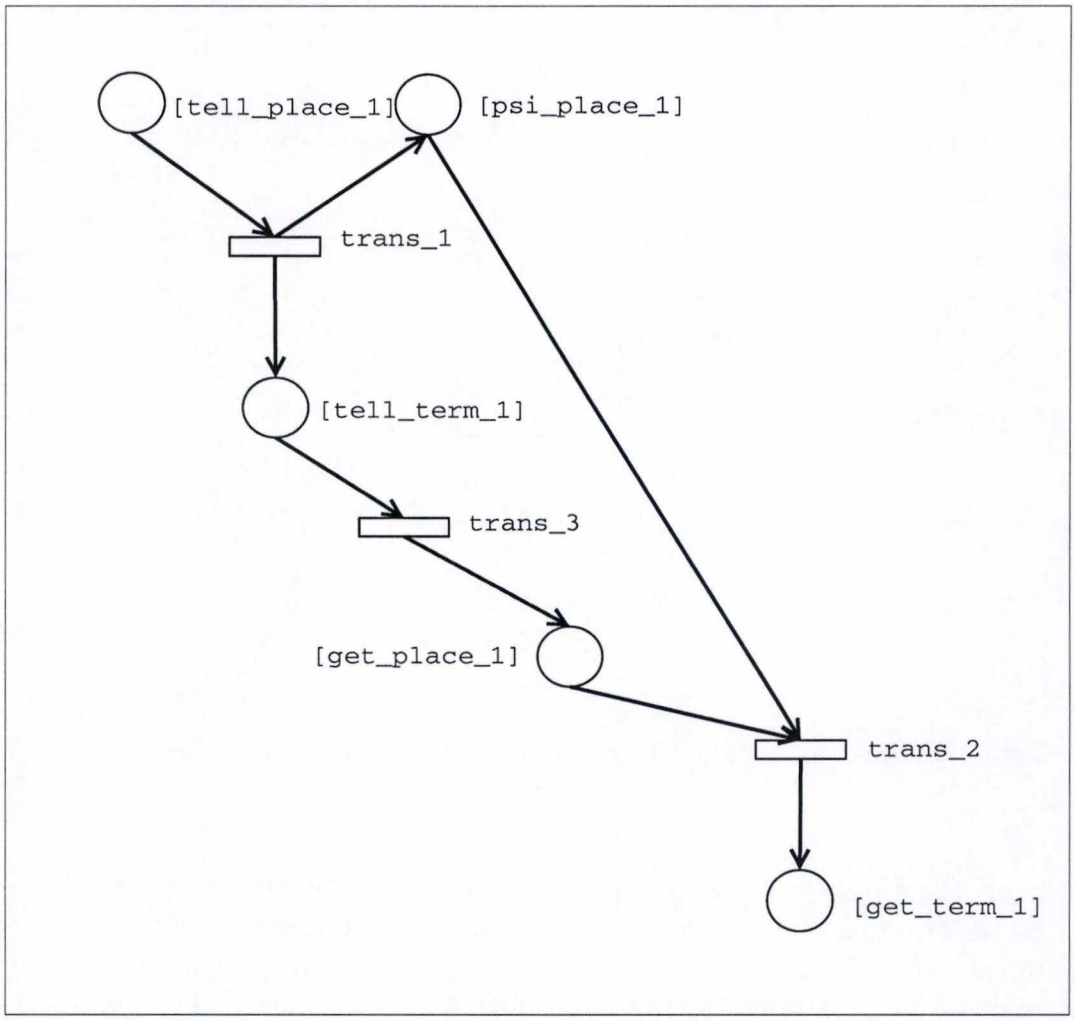

FIG. 6.7 – Le réseau relatif à l'agent obtenu par composition séquentielle de deux premiers agents élémentaires de l'agent A.

transition `trans_3` de Réseau 3 (relatif au troisième agent élémentaire) en `trans_4` afin d'éviter toute confusion.

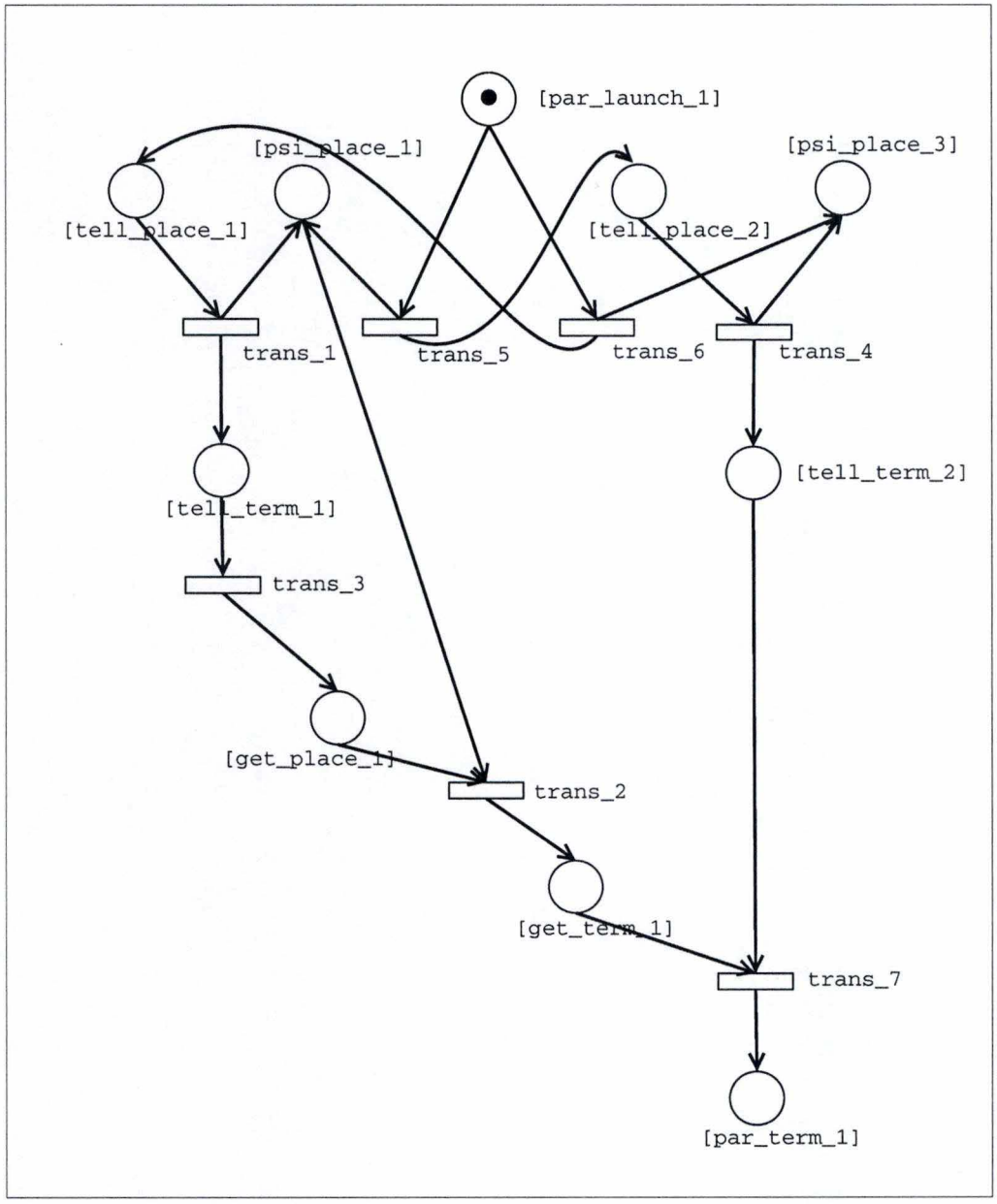

FIG. 6.8 – Le réseau associé l'agent A.

Nous avons fait exprès de nommer les transitions et les places comme notre application le fait. NFG génère, pour l'agent A, le fichier de réseau

donné à la figure 6.9.

```
P   M   PRE,POST  NETZ 0:
  1 0     1 5, 2
  2 0     6, 1
  3 0     1 5, 3
  4 0     3, 2
  5 0     2, 7
  6 0     4 6
  7 0     5, 4
  8 0     4 6, 7
  9 1    , 5 6
 10 0     7
@
place nr.              name capacity time
       1: psi_place_1             oo    0
       2: tell_place_1            oo    0
       3: tell_term_1             oo    0
       4: get_place_1             oo    0
       5: get_term_1              oo    0
       6: psi_place_3             oo    0
       7: tell_place_2            oo    0
       8: tell_term_2             oo    0
       9: par_launch_1            oo    0
      10: par_term_1              oo    0
@
trans nr.              name priority time
       1: trans_1                  0    0
       2: trans_2                  0    0
       3: trans_3                  0    0
       4: trans_4                  0    0
       5: trans_5                  0    0
       6: trans_6                  0    0
       7: trans_7                  0    0
@
```

FIG. 6.9 – Fichier de réseau `net050829_1820.pnt` associé à l'agent A généré par NFG.

Nous pouvons, par exemple, voir dans ce fichier que la place 1, [psi_place_1], a comme pré-transitions les transitions 1 et 5 (respectivement trans_1 et trans_5) et comme post-transition la transition 2 (trans_2. Et la place 9, [par_launch_1], qui est la place de lancement de ce réseau, n'a pas de pré-transitions et a comme post-transitions les transitions 5 et 6 (trans_5 et trans_6 respectivement). C'est ce que montre le réseau sur la figure 6.8. La figure 6.10 montre le fichier net050829_1820.ina qui décrit quelques analyses que nous avons effectuées avec INA.

```
------------------------------------------------------------------------
Integrated Net Analyzer [v2.2final-Jul 31 2003-win32] session report:
Current net options are:
    token  type: black        (for Place/Transition nets)
    time option: no times
    firing rule: normal
    priorities : not to be used
    strategy   : single transitions
    line length: 80

Net read from net050829_1820.pnt
Information on elementary structural properties:
Current name options are:
     transition names not to be written
     place names not to be written
Static conflicts:
transition 5 is in conflict with:
       6,
transition 6 is in conflict with:
       5,

The net is not statically conflict-free.
The net is pure.
The net is ordinary.
The net is homogenous.
The net is not conservative.
The net is not subconservative.
The net is not a state machine.
The net is free choice.
The net is extended free choice.
```

```
The net is extended simple.
The net has places without post-transition.
The net is not state machine decomposable (SMD).
The net is not state machine allocatable (SMA).
The net is not strongly connected.
Place 6 has no post-transition.
The net has places without pre-transition.
The net is not covered by semipositive T-invariants.
The net is not live.
The net is not live and safe.
Place 9 has no pre-transition.
The deadlock-trap-property is not valid.
Place 10 has no post-transition.
The net is marked.
The net is marked with exactly one token.
The net is not a marked graph.
The net has not a non-blocking multiplicity.
The net is structurally bounded.
The net is bounded.
There are no proper semipositive T-surinvariants.
The net has a nonempty clean trap.
The maximal clean trap:
      1,  2,  3,  4,  5,  6,  7,  8, 10,
The net has no transitions without pre-place.
The net has no transitions without post-place.
Maximal in/out-degree:  3
The net is connected.
ORD HOM NBM PUR CSV SCF CON SC  Ft0 tF0 Fp0 pF0 MG  SM  FC  EFC ES
 Y   Y   N   Y   N   N   Y   N   N   N   Y   Y   N   N   Y   Y   Y
DTP SMC SMD SMA CPI CTI  B  SB  REV DSt BSt DTr DCF  L   LV L&S
 N   ?   N   N   ?   N   Y   Y   ?   ?   ?   ?   ?   N   ?   N
Current analysis options are:
     no symmetrical reduction
     no stubborn reduction
     no depth restriction
     do not use a 'bad' predicate
Computation of the reachability graph
States generated: 9
Arcs generated: 11
```

```
Capacities needed:
Place  1  2  3  4  5  6  7  8  9 10
 Cap:  1  1  1  1  1  1  1  1  1  1
Dead states:
        6,
Number of dead states found: 1
The net has dead reachable states.
The net is not reversible (resetable).
The net has no dead transitions at the initial marking.
The net is not live, if dead transitions are ignored.
The net is safe.

State nr.    1
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  0  0  0  0  0  0  0  0  1  0
==t5=> s2
==t6=> s9
State nr.    2
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  1  0  1  0  0  0  1  0  0  0
==t3=> s3
==t4=> s8
State nr.    3
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  1  0  0  1  0  0  1  0  0  0
==t2=> s4
==t4=> s7
State nr.    4
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  0  0  0  0  1  0  1  0  0  0
==t4=> s5
State nr.    5
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  0  0  0  0  1  1  0  1  0  0
==t7=> s6
State nr.    6
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  0  0  0  0  0  1  0  0  0  1
dead state
State nr.    7
```

```
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  1  0  0  1  0  1  0  1  0  0
==t2=> s5
State nr.    8
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  1  0  1  0  0  1  0  1  0  0
==t3=> s7
State nr.    9
P.nr:  1  2  3  4  5  6  7  8  9 10
toks:  0  1  0  0  0  1  0  1  0  0
==t1=> s8

Searching for dynamic conflicts:
State  Conflict
   1:   5.trans_5 #   6.trans_6

The net is not dynamically conflict-free.
Computing the strongly connected components

The computed graph is not strongly connected.
List of strongly connected components:
Component nr. 1: 1,
Reachable scc's: 2 .. 9,
Component nr. 2: 9,
Reachable scc's: 4, 6, 8, 9,
Component nr. 3: 2,
Reachable scc's: 4 .. 9,
Component nr. 4: 8,
Reachable scc's: 6, 8, 9,
Component nr. 5: 3,
Reachable scc's: 6 .. 9,
Component nr. 6: 7,
Reachable scc's: 8, 9,
Component nr. 7: 4,
Reachable scc's: 8, 9,
Component nr. 8: 5,
Reachable scc's: 9,
Component nr. 9: 6, term.
ORD HOM NBM PUR CSV SCF CON SC  FtO tFO FpO pFO MG  SM  FC  EFC ES
 Y   Y   N   Y   N   N   Y   N   N   N   Y   Y   N   N   Y   Y   Y
```

```
DTP SMC SMD SMA CPI CTI  B  SB  REV DSt BSt DTr DCF  L   LV L&S
 N   ?   N   N   ?   N   Y   Y   N   Y   ?   N   N   N   N   N
Fire
Current name options are:
     transition names not to be written

 PL:  1  2  3  4  5  6  7  8  9 10
 MA:  0  0  0  0  0  0  0  0  1  0
fired transition: 5

 PL:  1  2  3  4  5  6  7  8  9 10
 MA:  1  0  1  0  0  0  1  0  0  0
fired transition: 3

 PL:  1  2  3  4  5  6  7  8  9 10
 MA:  1  0  0  1  0  0  1  0  0  0
fired transition: 4

 PL:  1  2  3  4  5  6  7  8  9 10
 MA:  1  0  0  1  0  1  0  1  0  0
fired transition: 2

 PL:  1  2  3  4  5  6  7  8  9 10
 MA:  0  0  0  0  1  1  0  1  0  0
fired transition: 7

 PL:  1  2  3  4  5  6  7  8  9 10
 MA:  0  0  0  0  0  1  0  0  0  1
 no transition has concession!
Net written to net050829_1821.pnt
Current options written to options.ina

End of Analyzer session.
------------------------------------------------------------------------
```

FIG. 6.10 – Le fichier `net050829_1820.ina` montrant les analyses effectuées sur le fichier `net050829_1820.pnt`.

A travers ce fichier, nous pouvons observer certaines propriétés du réseau de l'agent *A*. Nous voyons, par exemple, que ce réseau est *ordinaire*, *homogène* ou encore *borné* ; nous pouvons également voir qu'il n'est pas *vivant* ou que son graphe n'est pas *fortement connexe*. Nous pouvons aussi observer une séquence de différents marquages et les franchissements qui sont à la base de ces marquages.

Nous n'avons pris qu'une poignée d'analyses pour montrer, en exemple, ce que INA peut faire à partir d'un fichier de réseau que lui est soumis. Et dans le cas d'espèce, ce fichier a été généré par NFG.

Conclusion générale et pistes éventuelles

Dans ce travail, il a été question d'écrire la sémantique opérationnelle d'un langage de coordination dans le formalisme des réseaux de Petri. Etant donné la nature même des langages de coordination, nous avons choisi de profiter de l'expressivité qu'offrent les réseaux de Petri contextuels utilisant les arcs inhibiteurs pour exprimer de manière plus claire et plus précise certaines primitives des langages de coordination, notamment celles qui vérifient l'absence ou la présence d'un tuple dans l'espace partagé. Notre apport a été d'exprimer la sémantique opérationnelle du langage de coordination $\mathcal{L}_{\Psi}^{r}$ au moyen de réseaux de Petri contextuels. Pour ce faire, nous avons introduit des concepts tels que place de lancement, place terminale ou encore transition de lancement pour pouvoir définir de manière efficace les différentes compositions (séquentielle, parallèle et alternative) de $\mathcal{L}_{\Psi}^{r}$. Nous avons adopté une vue *statique* du comportement de l'agent, ce qui nous a permis d'éviter une explosion du nombre des places du réseau tout en les localisant dans des ensembles disjoints selon leur *nature*. Nous avons établi que la sémantique exprimée en termes des réseaux de Petri contextuels était équivalente à celle présentée dans le style de Plotkin. L'existence dans la littérature des résultats qui permettent de transformer un réseau de Petri contextuel en un réseau de Petri simple tout en préservant certaines propriétés majeures du réseau original, nous a permis d'implémenter une partie de cette sémantique en utilisant le générateur d'analyseurs lexicaux JLex et le générateur de parseurs Cup. L'application, de type compilateur, que nous avons écrite permet de transformer un agent exprimé dans le langage $\mathcal{L}_{\Psi}^{r}$ en un fichier de réseau accepté par INA, un outil d'analyse et de vérification de propriétés des réseaux de Petri.

La démarche que nous présentons dans ce travail est un pas de plus dans l'application des techniques de vérification des programmes aux langages de coordination. D'autres approches mériteraient des investigations.

Par exemple, la sémantique du langage étudié peut être exprimée dans une algèbre de processus. Des automates exprimés en PROMELA pourraient alors être générés et soumis à SPIN pour la vérification des propriétés.

Nous pensons que ce travail peut être poursuivi notamment dans l'écriture des règles de transformation du réseau de contextuel au réseau simple en utilisant la sémantique opérationnelle de $\mathcal{L}^r_\Psi$ que nous avons présentée. En effet, nous avons fait recours à un résultat proposant une transformation qui préserve essentiellement les séquences de franchissement de transitions et les deadlocks. Il serait intéressant de penser à de transformations qui pourront préserver d'autres propriétés de réseaux en plus des deux précitées. La particularité de $\mathcal{L}^r_\Psi$ consistant à manipuler de l'information partielle dans les Ψ-termes complexifiera, peut-être, cette tâche et le rendra davantage digne d'intérêt. Une autre piste concerne l'application NFG. En effet, on pourrait élargir le champs de traitement de NFG en permettant à l'utilisateur d'avoir le choix entre plusieurs outils d'analyse des réseaux : INA, PEP ou un autre outil. Puisque nous parlons de l'amélioration de NFG, on pourrait également entrevoir la possibilité d'ajouter un autre formalisme, en plus de celui des réseaux de Petri, en y associant les outils d'analyse adéquats.

Bibliographie

[1] Design/CPN Tool Site. http ://www.daimi.au.dk/designCPN/.

[2] PEP Tool Site. http ://peptool.sourceforge.net/.

[3] Projet ACCORD. Etat de l'art sur les langages de coordination. Technical report, Réseau National des Techniques Logicielles, RNTL, Juin 2002.

[4] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs : Principes, Techniques et Outils.* Dunod, 2000.

[5] Jean-Marc Andreoli and François Pacull. Distributed Print on Demand Systems in the Xpect Framework, 1997. http ://www.xrce.xerox.com/Publications/Attachments/1999-212/HTML/.

[6] N. A. Anisimov, E. A. Golenkov, and D. I. Kharitonov. Compositional Petri net approach to the development of concurrent and distributed systems. *Programming and Computer Software*, 27(6) :309–319, 2001.

[7] Farhad Arbab, Ivan Herman, and P. Spilling. An overview of Manifold and its implementation. *concurrency : Practice and Experience*, 5(1) :23–70, 1993.

[8] Farhad Arbab and Georges A. Papadopoulos. Control-driven Coordination Programming in Shared Dataspace. In V. Malyshkin, editor, *Parallel Computing Technologies : Proceedings of 4th International Conference*, volume 1277, pages 247–261. Lect. Notes in Comp. Sci., Springer, September 1997.

[9] Farhad Arbab and Georges A. Papadopoulos. Coordination models and languages. Technical report, Department of Software Engineering. CWI, December 1998.

[10] Laurent Berger, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. Interactions between objects : an aspect of object-oriented languages, 1998.

[11] Elliot Berk and C. Scott Ananian, 2000. http ://www.cs.princeton.edu/ appel/modern/java/JLex/current/manual.html.

[12] Stephen L. Bloom and Zoltán Ésik. Varieties generated by languages with poset operations. *Mathematical Structures in Computer Science*, 7 :701–713, December 1997.

[13] Antonio Brogi, Jean-Marie Jacquet, and Isabelle Linden. On modeling coordination via asynchronous communication and enhanced matching. *Electr. Notes Theor. Comput. Sci.*, 68(3), 2003.

[14] Nadia Busi and Roberto Gorrieri. A Petri Net Semantics for π-Calculus. In *Proceedings of the 6th International Conference on Concurrency Theory table of contents. LNCS*, volume 962, pages 145–159. Springer-Verlag, 1995.

[15] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A Truly Concurrent View of Linda Interprocess Communication. Technical report, Department of Computer Science, University of Bologna, February 1997.

[16] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the Expressiveness on Linda Coordination Primitives. *Information and Computation*, 156 :90–121, 2000.

[17] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4) :444–458, 1989.

[18] Juan-Carlos Cruz and Stéphane Ducasse. A Group Based Approach for Coordinating Active Objects. In *Proceedings of Coordination'99, LNCS*, volume 1594, pages 355–371, Amsterdam, The Netherlands, 1999.

[19] Gjalt G. de Jong and Bill Lin. A communicating petri net model for the design of concurrent asynchronous modules. In *Annual ACM IEEE Design Automation Conference. Proceedings of the 31st annual conference on Design automation*, pages 49–55. ACM Press, 1994.

[20] Boris Feigin. Model checking, March 2005. http ://www.cs.ucl.ac.uk/staff/W.Emmerich/lectures/3C05-04-05/ModelChecking.pdf.

[21] Jean Fichefet. Théorie des graphes et compléments de mathématiques, 1999-2000. Institut d'Informatique, FUNDP de Namur.

[22] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1) :80–112, 1985.

[23] David Gelernter. Multiple Tuple Spaces in Linda. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89 : Parallel Architectures and Languages Europe. Volume II : Parallel Languages*, volume 366 of *LNCS*, pages 20–27. Springer-Verlag, 1989.

[24] David Gelernter and Nicholas Carriero. How to write parallel programs : A guide to the perplexed. *ACM Computing surveys*, 21(3) :323–357, 1989.

[25] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2) :97–107, 1992.

[26] David Gelernter and Lenore D. Zuck. On What Linda Is : Formal Description of Linda as a Reactive System. In *COORDINATION*, pages 187–204, 1997.

[27] Luuk Groenewegen and Erik de Vink. Operational Semantics for Coordination in Paradigm. In F. Arbab and C. Talcott, editors, *Proc. Coordination 2002*, pages 191–206. LNCS 2315, April 2002.

[28] Béat Hirsbrunner. Introduction à Linda, Mai 1991. Notes de cours, Université de Fribourg.

[29] Béat Hirsbrunner. Langages de coordination, 2000. Exposé à l'Université de Montréal.

[30] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions of Software Engineering*, 23(5), May 1997. http ://spin-root.com/spin/Doc/ieee97.pdf.

[31] Scott E. Hudson, Frank Flannery, C. Scott Ananian, and Dan Wang, 1999. http ://www.cs.princeton.edu/ appel/modern/java/CUP/manual.html.

[32] Jarle Hulaas and David Billard. Cours de Systèmes Informatiques 2, 2000-2001. Département d'Informatique, Université de Génève.

[33] Daniel Kühni. APROCO : A Programmable Coordination Medium, October 1998. Diploma thesis, University of Bern.

[34] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly, February 1995.

[35] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[36] Kjeld H. Mortensen. Petri nets world site. http ://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html.

[37] Christophe Reutenauer. *Mathematics of Petri Nets.* Prentice Hall, Inc., June 1990.

[38] Michael Schumacher, Olivier Krone, Fabrice Chantemargue, and Béat Hirsbrunner. STL : Un modèle et langage de coordination pour les systèmes distribués, 1998.

[39] Marc L. Smith, Rebecca J. Parsons, and Charles E. Hughes. View-Centric Reasoning for Linda and Tuple Space Computation. In James Pascoe, Roger Loader, and Vaidy Sunderman, editors, *Communicating Process Architectures.* IOS Press, 2002.

[40] Peter H. Starke and Stephan Roch. Integrated Nets Analyzer manual, 2003. http ://www.informatik.hu-berlin.de/ starke/ina.html.

[41] Rune Teigen. COOrdination Language, 1997. http ://eil.utoronto.ca/profiles/rune/node7.html.

[42] Pierre Wolper. The Algorithmic Verification of Reactive Systems, 1998. http ://www.montefiore.ulg.ac.be/

Annexes

Annexe A

Les spécifications relatives à l'analyseur lexical de l'application NFG

Voici le fichier `MyLexer.lex` des spécifications lexicales. Ce fichier a été soumis à JLex afin de générer l'analyseur lexical `Lexer.java` de l'annexe C qui permet la reconnaissance des lexèmes du langage $\mathcal{L}_{\Psi}^{r}$ dans l'application NFG.

```
package NFGTool;

import java.lang.*;
import java_cup.runtime.*;

class ErrorManaging {
    public static void assertFunction(boolean bool){
        if(false == bool){
            throw(new Error("Error : Assertion failed."));
        }
    }

    private static final String errorMsg[] = {
        "Error: Unmatched end-of-comment punctuation.",
        "Error: Unmatched start-of-comment punctuation.",
        "Error: Unclosed string.",
        "Error: Illegal character.",
        "Error: Missing terminal character."
    };
```

```
    public static final int E_ENDCOMMENT = 0;
    public static final int E_STARTCOMMENT = 1;
    public static final int E_UNCLOSEDSTRING = 2;
    public static final int E_UNMATCHED = 3;
    public static final int E_ENDMISSING  = 4;

    public static void error(int code) {
        System.out.println(errorMsg[code]);
    }
}

class TokenValue {
/* Cette classe définit un token standard qui
sera reconnu par le lexeur */

    public String text;
    public Object first_object;
    public Object second_object;

    /*Les différents constructeurs */
    TokenValue() {
        this("", null, null);
    }

    TokenValue(String text) {
        this(text, null, null);
    }

    TokenValue(Object first, Object second) {
        this("", first, second);
    }

    TokenValue(String text, Object first, Object second) {
        this.text = text;
        this.first_object = first;
        this.second_object = second;
    }

    public String toString() {
        return text;
```

```
    }

    public Object getFirstObject() {
        return first_object;
    }

    public Object getSecondObject() {
        return second_object;
    }

    public void setFirstObject(Object obj) {
        first_object = obj;
    }

    public void setSecondObject(Object obj) {
        second_object = obj;
    }
}

%%
%cup
%class Lexer
%unicode
%line
%char
%public
%eofval{
    return new Symbol(sym.EOF, null);
%eofval}

%state COMMENTS

NUMBER = [0-9]+
CHARACTER = [a-zA-Z] | [0-9] | [\ \\-_\'()<>]
WHITE_SPACE = ([\ \n\r\t\f])+
SEQ = ";"
PAR = "||"
ALT = "+"
SHARP = "#"
LPAREN = "("
RPAREN = ")"
```

```
SLASH = "/"
QUOTATION = "\""
EQUAL_SIGN = "="
COMMA = ","
TELL = "tell"
GET = "get"
ASK = "ask"
NASK = "nask"
COMM_VAR = [X-Z]({NUMBER})*
ITEM = [x-z]({NUMBER})*
MY_STRING = {QUOTATION}({CHARACTER})*{QUOTATION}

FUNCTOR = [f-h]{SLASH}{NUMBER}

%%
<YYINITIAL> {WHITE_SPACE} {
}

<YYINITIAL> {NUMBER} {
    return new Symbol(sym.NUMBER, new MyNumber(yytext()));
}

<YYINITIAL> {CHARACTER} {
    return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}

<YYINITIAL> {MY_STRING} {
    String str = yytext().substring(1, yytext().length()-1);
    return new Symbol(sym.MY_STRING, new MyString(str));
}

<YYINITIAL> {QUOTATION}({CHARACTER})* {
    String str = yytext().substring(1, yytext().length());
    ErrorManaging.error(ErrorManaging.E_UNCLOSEDSTRING);
    return new Symbol(sym.MY_STRING, new MyString(str));
}

<YYINITIAL> {SEQ} {
    return new Symbol(sym.SEQ, new TokenValue(yytext()));
}
```

```
<YYINITIAL> {PAR} {
    return new Symbol(sym.PAR, new TokenValue(yytext()));
}

<YYINITIAL> {ALT} {
    return new Symbol(sym.ALT, new TokenValue(yytext()));
}

<YYINITIAL> {LPAREN} {
    return new Symbol(sym.LPAREN, new TokenValue(yytext()));
}

<YYINITIAL> {RPAREN} {
    return new Symbol(sym.RPAREN, new TokenValue(yytext()));
}

<YYINITIAL> {SLASH} {
    return new Symbol(sym.SLASH, new TokenValue(yytext()));
}

<YYINITIAL> {QUOTATION} {
    return new Symbol(sym.QUOTATION, new TokenValue(yytext()));
}

<YYINITIAL> {EQUAL_SIGN} {
    return new Symbol(sym.EQUAL_SIGN, new TokenValue(yytext()));
}
<YYINITIAL> {COMMA} {
    return new Symbol(sym.COMMA, new TokenValue(yytext()));
}

<YYINITIAL> {FUNCTOR} {
    return new Symbol(sym.FUNCTOR, new Functor(yytext()));
}

<YYINITIAL> {ITEM} {
    return new Symbol(sym.ITEM, new Item(yytext()));
}

<YYINITIAL> {COMM_VAR} {
    return new Symbol(sym.COMM_VAR, new CommunicationVariable(yytext()));
```

```
}

<YYINITIAL> {TELL} {
    return new Symbol(sym.TELL, new TokenValue(yytext()));
}

<YYINITIAL> {GET} {
    return new Symbol(sym.GET, new TokenValue(yytext()));
}


<YYINITIAL> {ASK} {
    return new Symbol(sym.ASK, new TokenValue(yytext()));
}

<YYINITIAL> {NASK} {
    return new Symbol(sym.NASK, new TokenValue(yytext()));
}

<YYINITIAL> {SHARP} {
    return new Symbol(sym.SHARP, new TokenValue(yytext()));
}

<YYINITIAL> "//" {
    yybegin(COMMENTS);
}

<COMMENTS> [^\n] {}

<COMMENTS> [\n] {
    yybegin(YYINITIAL);
}

<YYINITIAL> . {
    ErrorManaging.error(ErrorManaging.E_UNMATCHED);
    return new Symbol(sym.error, null);
}
```

Le lecteur familier du Lex peut se retrouver assez facilement dans la structure de ce fichier. Certaines différences ont été expliquées dans la première section du chapitre 6. Pour plus de détails, on peut se référer au manuel

[11].

Annexe B

Les spécifications relatives au parseur de l'application NFG

Ceci est le fichier `MyParser.cup` contenant les spécifications relatives au parseur de NFG. Ce fichier contient la grammaire qui décrit les agents dans le langage $\mathcal{L}_{\Psi}^{r}$, et utilise la structure d'arbre qui nous permet de reconnaître tous les types d'agents, aussi bien élémentaires que complexes. Un rapide coup d'oeil dans la section 6.2 du chapitre 6 peut aider à mieux comprendre la structure de ce fichier. Le lecteur désireux d'avoir plus de détail sur les spécificatins de Cup peut se référer au manuel [31].

```
package NFGTool;

import java_cup.runtime.*;

parser code {:
    Lexer lexer;

    public parser(Lexer lexer) {
        this.lexer = lexer;
    }

    /* Une méthode par gérer les messages d'erreur émis par
    le parseur */
    public void report_error(String message, Object info) {
        StringBuffer m = new StringBuffer("Error");

        if (info instanceof java_cup.runtime.Symbol) {
            java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol)
```

```
                                        info);
            if (s.left >= 0) {
                m.append(" in line "+(s.left+1));
                if (s.right >= 0)
                    m.append(", column "+(s.right+1));
            }
        }
        m.append(" : "+message);
        System.err.println(m);
    }

    public void report_fatal_error(String message, Object info) {
        report_error(message, info);
        System.exit(1);
    }
:}

scan with {: return lexer.next_token(); :};

/* Les terminaux */

terminal CHARACTER, WHITE_SPACE, SEQ, PAR, ALT, SHARP,
         LPAREN, RPAREN, SLASH;
terminal MyNumber  NUMBER;
terminal MyString  MY_STRING;
terminal Item ITEM;
terminal CommunicationVariable COMM_VAR;
terminal Functor FUNCTOR;
terminal PRIMITIVE, QUOTATION, EQUAL_SIGN, TELL, GET,
         ASK, NASK, COMMA;

/* Les non-terminaux */

non terminal agent_list, agent_part;
non terminal Agent agent, comm_action;
non terminal PsiTerm psi_term;
non terminal ValueList value_funct_arg_list;
non terminal Value value_funct_arg;

/*Les précédences et l'associativité des opérateurs de composition*/
```

```
precedence left SEQ, PAR, ALT;

/* La grammaire du formalisme */

start with agent_list;

agent_list ::= agent_list agent_part
               | agent_part
               ;

agent_part ::= agent:a
               {:
                   Agent.generateFileForAgent(a);
               :} SHARP
               ;

agent ::= comm_action:c
               {:
                   Agent.processPrimitive(c);
                   RESULT = c;
               :}

               | LPAREN agent:a RPAREN
               {:
                   RESULT = a;
               :}

               | agent:a SEQ agent:b
               {:
                   Agent res = new Agent(a, b);
                   Agent.sequentialComposition(res, a, b);
                   RESULT = res;
               :}

               | agent:a PAR agent:b
               {:
                   Agent res = new Agent(a, b);
                   Agent.parallelComposition(res, a, b);
                   RESULT = res;
               :}
```

```
                | agent:a ALT agent:b
                {:
                    Agent res = new Agent(a, b);
                    Agent.alternativeComposition(res, a, b);
                    RESULT = res;
                :}
                ;

comm_action ::= TELL LPAREN psi_term:p RPAREN
                {:
                    RESULT = new Agent("tell", p);
                :}

                | GET LPAREN psi_term:p RPAREN
                {:
                    RESULT = new Agent("get", p);
                :}

                | ASK LPAREN psi_term:p RPAREN
                {:
                    RESULT = new Agent("ask", p);
                :}

                | NASK LPAREN psi_term:p RPAREN
                {:
                    RESULT = new Agent("nask", p);
                :}
                ;

psi_term ::= FUNCTOR:f LPAREN value_funct_arg_list:vl RPAREN
                {:
                    RESULT = new PsiTerm(f,
        ValueList.buildValueArray(vl));
                :}
                ;

value_funct_arg_list ::= value_funct_arg:v COMMA
                                value_funct_arg_list:l
                {:
                    RESULT = new ValueList(v, l);
                :}
```

```
                | value_funct_arg:v
                {:
                    RESULT = new ValueList(v);
                :}
                ;

value_funct_arg ::= ITEM:i EQUAL_SIGN NUMBER:n
                {:
                    RESULT = new Value(i, n.getValue());
                :}

                | ITEM:i EQUAL_SIGN MY_STRING:s
                {:
                    RESULT = new Value(i, s.getValue());
                :}

                | ITEM:i EQUAL_SIGN COMM_VAR:c
                {:
                    RESULT = new Value(i, c.getValue());
                :}

                | ITEM:i EQUAL_SIGN psi_term:p
                {:
                    RESULT = new Value(i,p);
                :}
                ;
```

Ce fichier des spécifications a été soumis au générateur de parseurs Cup qui produit en sortie les fichiers `parser.java` (Annexe D, section D.1) et `sym.java` (Annexe D, section D.2) utilisés dans NFG.

Annexe C

L'analyseur lexical de NFG généré par JLex

Voici le fichier `Lexer.java` généré par JLex à partir des spécifications présentées dans `MyLexer.lex` de l'annexe A.

```
package NFGTool;

import java.lang.*;
import java_cup.runtime.*;

class ErrorManaging {
public static void assertFunction(boolean bool){
if(false == bool){
throw(new Error("Error : Assertion failed."));
}
}

private static final String errorMsg[] = {
"Error: Unmatched end-of-comment punctuation.",
"Error: Unmatched start-of-comment punctuation.",
"Error: Unclosed string.",
"Error: Illegal character.",
"Error: Missing terminal character."};

public static final int E_ENDCOMMENT = 0;
public static final int E_STARTCOMMENT = 1;
public static final int E_UNCLOSEDSTRING = 2;
public static final int E_UNMATCHED = 3;
```

```
public static final int E_ENDMISSING  = 4;

public static void error(int code) {
System.out.println(errorMsg[code]);
}
}

class TokenValue {
/* Cette classe définit un token standard qui sera reconnu
par le lexeur */

public String text;
public Object first_object;
public Object second_object;

/*Les différents constructeurs */
TokenValue() {
this("", null, null);
}

TokenValue(String text) {
this(text, null, null);
}

TokenValue(Object first, Object second) {
this("", first, second);
}

TokenValue(String text, Object first, Object second) {
this.text = text;
this.first_object = first;
this.second_object = second;
}

public String toString() {
return text;
}

public Object getFirstObject() {
return first_object;
}
```

```
public Object getSecondObject() {
return second_object;
}

public void setFirstObject(Object obj) {
first_object = obj;
}

public void setSecondObject(Object obj) {
second_object = obj;
}
}

public class Lexer implements java_cup.runtime.Scanner {
private final int YY_BUFFER_SIZE = 512;
private final int YY_F = -1;
private final int YY_NO_STATE = -1;
private final int YY_NOT_ACCEPT = 0;
private final int YY_START = 1;
private final int YY_END = 2;
private final int YY_NO_ANCHOR = 4;
private final int YY_BOL = 65536;
private final int YY_EOF = 65537;
private java.io.BufferedReader yy_reader;
private int yy_buffer_index;
private int yy_buffer_read;
private int yy_buffer_start;
private int yy_buffer_end;
private char yy_buffer[];
private int yychar;
private int yyline;
private boolean yy_at_bol;
private int yy_lexical_state;

public Lexer (java.io.Reader reader) {
this ();
if (null == reader) {
throw (new Error("Error: Bad input stream initializer."));
}
yy_reader = new java.io.BufferedReader(reader);
```

```
}

public Lexer (java.io.InputStream instream) {
this ();
if (null == instream) {
throw (new Error("Error: Bad input stream initializer."));
}
yy_reader = new java.io.BufferedReader(new java.io.
InputStreamReader(instream));
}

private Lexer () {
yy_buffer = new char[YY_BUFFER_SIZE];
yy_buffer_read = 0;
yy_buffer_index = 0;
yy_buffer_start = 0;
yy_buffer_end = 0;
yychar = 0;
yyline = 0;
yy_at_bol = true;
yy_lexical_state = YYINITIAL;
}

private boolean yy_eof_done = false;
private final int YYINITIAL = 0;
private final int COMMENTS = 1;
private final int yy_state_dtrans[] = {
0,
42
};
private void yybegin (int state) {
yy_lexical_state = state;
}
private int yy_advance ()
throws java.io.IOException {
int next_read;
int i;
int j;

if (yy_buffer_index < yy_buffer_read) {
return yy_buffer[yy_buffer_index++];
```

```
}

if (0 != yy_buffer_start) {
i = yy_buffer_start;
j = 0;
while (i < yy_buffer_read) {
yy_buffer[j] = yy_buffer[i];
++i;
++j;
}
yy_buffer_end = yy_buffer_end - yy_buffer_start;
yy_buffer_start = 0;
yy_buffer_read = j;
yy_buffer_index = j;
next_read = yy_reader.read(yy_buffer,
yy_buffer_read,
yy_buffer.length - yy_buffer_read);
if (-1 == next_read) {
return YY_EOF;
}
yy_buffer_read = yy_buffer_read + next_read;
}

while (yy_buffer_index >= yy_buffer_read) {
if (yy_buffer_index >= yy_buffer.length) {
yy_buffer = yy_double(yy_buffer);
}
next_read = yy_reader.read(yy_buffer,
yy_buffer_read,
yy_buffer.length - yy_buffer_read);
if (-1 == next_read) {
return YY_EOF;
}
yy_buffer_read = yy_buffer_read + next_read;
}
return yy_buffer[yy_buffer_index++];
}
private void yy_move_end () {
if (yy_buffer_end > yy_buffer_start &&
    '\n' == yy_buffer[yy_buffer_end-1])
yy_buffer_end--;
```

```
if (yy_buffer_end > yy_buffer_start &&
    '\r' == yy_buffer[yy_buffer_end-1])
yy_buffer_end--;
}
private boolean yy_last_was_cr=false;
private void yy_mark_start () {
int i;
for (i = yy_buffer_start; i < yy_buffer_index; ++i) {
if ('\n' == yy_buffer[i] && !yy_last_was_cr) {
++yyline;
}
if ('\r' == yy_buffer[i]) {
++yyline;
yy_last_was_cr=true;
} else yy_last_was_cr=false;
}
yychar = yychar
+ yy_buffer_index - yy_buffer_start;
yy_buffer_start = yy_buffer_index;
}
private void yy_mark_end () {
yy_buffer_end = yy_buffer_index;
}
private void yy_to_mark () {
yy_buffer_index = yy_buffer_end;
yy_at_bol = (yy_buffer_end > yy_buffer_start) &&
            ('\r' == yy_buffer[yy_buffer_end-1] ||
             '\n' == yy_buffer[yy_buffer_end-1] ||
             2028/*LS*/ == yy_buffer[yy_buffer_end-1] ||
             2029/*PS*/ == yy_buffer[yy_buffer_end-1]);
}
private java.lang.String yytext () {
return (new java.lang.String(yy_buffer,
yy_buffer_start,
yy_buffer_end - yy_buffer_start));
}
private int yylength () {
return yy_buffer_end - yy_buffer_start;
}
private char[] yy_double (char buf[]) {
int i;
```

```
char newbuf[];
newbuf = new char[2*buf.length];
for (i = 0; i < buf.length; ++i) {
newbuf[i] = buf[i];
}
return newbuf;
}
private final int YY_E_INTERNAL = 0;
private final int YY_E_MATCH = 1;
private java.lang.String yy_error_string[] = {
"Error: Internal error.\n",
"Error: Unmatched input.\n"
};
private void yy_error (int code,boolean fatal) {
java.lang.System.out.print(yy_error_string[code]);
java.lang.System.out.flush();
if (fatal) {
throw new Error("Fatal Error.\n");
}
}
private int[][] unpackFromString(int size1, int size2, String st) {
int colonIndex = -1;
String lengthString;
int sequenceLength = 0;
int sequenceInteger = 0;

int commaIndex;
String workString;

int res[][] = new int[size1][size2];
for (int i= 0; i < size1; i++) {
for (int j= 0; j < size2; j++) {
if (sequenceLength != 0) {
res[i][j] = sequenceInteger;
sequenceLength--;
continue;
}
commaIndex = st.indexOf(',');
workString = (commaIndex==-1) ? st :
st.substring(0, commaIndex);
st = st.substring(commaIndex+1);
```

```
colonIndex = workString.indexOf(':');
if (colonIndex == -1) {
res[i][j]=Integer.parseInt(workString);
continue;
}
lengthString =
workString.substring(colonIndex+1);
sequenceLength=Integer.parseInt(lengthString);
workString=workString.substring(0,colonIndex);
sequenceInteger=Integer.parseInt(workString);
res[i][j] = sequenceInteger;
sequenceLength--;
}
}
return res;
}
private int yy_acpt[] = {
/* 0 */ YY_NOT_ACCEPT,
/* 1 */ YY_NO_ANCHOR,
/* 2 */ YY_NO_ANCHOR,
/* 3 */ YY_NO_ANCHOR,
/* 4 */ YY_NO_ANCHOR,
/* 5 */ YY_NO_ANCHOR,
/* 6 */ YY_NO_ANCHOR,
/* 7 */ YY_NO_ANCHOR,
/* 8 */ YY_NO_ANCHOR,
/* 9 */ YY_NO_ANCHOR,
/* 10 */ YY_NO_ANCHOR,
/* 11 */ YY_NO_ANCHOR,
/* 12 */ YY_NO_ANCHOR,
/* 13 */ YY_NO_ANCHOR,
/* 14 */ YY_NO_ANCHOR,
/* 15 */ YY_NO_ANCHOR,
/* 16 */ YY_NO_ANCHOR,
/* 17 */ YY_NO_ANCHOR,
/* 18 */ YY_NO_ANCHOR,
/* 19 */ YY_NO_ANCHOR,
/* 20 */ YY_NO_ANCHOR,
/* 21 */ YY_NO_ANCHOR,
/* 22 */ YY_NO_ANCHOR,
/* 23 */ YY_NO_ANCHOR,
```

```
/* 24 */ YY_NO_ANCHOR,
/* 25 */ YY_NO_ANCHOR,
/* 26 */ YY_NO_ANCHOR,
/* 27 */ YY_NOT_ACCEPT,
/* 28 */ YY_NO_ANCHOR,
/* 29 */ YY_NO_ANCHOR,
/* 30 */ YY_NOT_ACCEPT,
/* 31 */ YY_NO_ANCHOR,
/* 32 */ YY_NOT_ACCEPT,
/* 33 */ YY_NO_ANCHOR,
/* 34 */ YY_NOT_ACCEPT,
/* 35 */ YY_NO_ANCHOR,
/* 36 */ YY_NOT_ACCEPT,
/* 37 */ YY_NO_ANCHOR,
/* 38 */ YY_NOT_ACCEPT,
/* 39 */ YY_NO_ANCHOR,
/* 40 */ YY_NOT_ACCEPT,
/* 41 */ YY_NO_ANCHOR,
/* 42 */ YY_NOT_ACCEPT
};
private int yy_cmap[] = unpackFromString(1,65538,
"25:9,26,1,25,26:2,25:18,26,25,4,24,25:4,8,9,25,7,12,25:2,10,2:10,
25,5,25,11" + ",25:3,3:23,15:3,25:6,20,3:3,17,13,19,13,3:2,22,18,
3,23,3:4,21,16,3:3,14:3,2" + "5,6,25:65411,0:2")[0];

private int yy_rmap[] = unpackFromString(1,43,
"0,1,2,3,1,4,1,5,1:3,6,1:6,7,8,9,1:6,9,10,1,11,7,12,8,13,14,15,16,
17,18,19,2" + "0,21")[0];

private int yy_nxt[][] = unpackFromString(22,27,
"1,2,3,4,5,6,7,8,9,10,11,12,13,28,31,33,35,4:2,37,39,4:2,41,14,29,
2,-1:28,2," + "-1:24,2,-1:2,3,-1:27,5,15,-1:8,5:11,-1:9,16,-1:30,
17,-1:18,18,-1:26,19,-1:2" + "6,20,-1:34,27,-1:34,38,-1:24,21,-1:32,
22,-1:21,30,-1:30,40,-1:15,27,-1:6,32" + ",-1:27,23,-1:29,34,-1:27,
24,-1:24,36,-1:6,1,25,26:25");

public java_cup.runtime.Symbol next_token ()
throws java.io.IOException {
int yy_lookahead;
int yy_anchor = YY_NO_ANCHOR;
int yy_state = yy_state_dtrans[yy_lexical_state];
```

```
int yy_next_state = YY_NO_STATE;
int yy_last_accept_state = YY_NO_STATE;
boolean yy_initial = true;
int yy_this_accept;

yy_mark_start();
yy_this_accept = yy_acpt[yy_state];
if (YY_NOT_ACCEPT != yy_this_accept) {
yy_last_accept_state = yy_state;
yy_mark_end();
}
while (true) {
if (yy_initial && yy_at_bol) yy_lookahead = YY_BOL;
else yy_lookahead = yy_advance();
yy_next_state = YY_F;
yy_next_state = yy_nxt[yy_rmap[yy_state]][yy_cmap[yy_lookahead]];
if (YY_EOF == yy_lookahead && true == yy_initial) {

return new Symbol(sym.EOF, null);
}
if (YY_F != yy_next_state) {
yy_state = yy_next_state;
yy_initial = false;
yy_this_accept = yy_acpt[yy_state];
if (YY_NOT_ACCEPT != yy_this_accept) {
yy_last_accept_state = yy_state;
yy_mark_end();
}
}
else {
if (YY_NO_STATE == yy_last_accept_state) {
throw (new Error("Lexical Error: Unmatched Input."));
}
else {
yy_anchor = yy_acpt[yy_last_accept_state];
if (0 != (YY_END & yy_anchor)) {
yy_move_end();
}
yy_to_mark();
switch (yy_last_accept_state) {
case 1:
```

```
case -2:
break;
case 2:
{
}
case -3:
break;
case 3:
{
return new Symbol(sym.NUMBER, new MyNumber(yytext()));
}
case -4:
break;
case 4:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -5:
break;
case 5:
{
String str = yytext().substring(1, yytext().length());
ErrorManaging.error(ErrorManaging.E_UNCLOSEDSTRING);
return new Symbol(sym.MY_STRING, new MyString(str));
}
case -6:
break;
case 6:
{
return new Symbol(sym.SEQ, new TokenValue(yytext()));
}
case -7:
break;
case 7:
{
ErrorManaging.error(ErrorManaging.E_UNMATCHED);
return new Symbol(sym.error, null);
}
case -8:
break;
```

```
case 8:
{
return new Symbol(sym.ALT, new TokenValue(yytext()));
}
case -9:
break;
case 9:
{
return new Symbol(sym.LPAREN, new TokenValue(yytext()));
}
case -10:
break;
case 10:
{
return new Symbol(sym.RPAREN, new TokenValue(yytext()));
}
case -11:
break;
case 11:
{
return new Symbol(sym.SLASH, new TokenValue(yytext()));
}
case -12:
break;
case 12:
{
return new Symbol(sym.EQUAL_SIGN, new TokenValue(yytext()));
}
case -13:
break;
case 13:
{
return new Symbol(sym.COMMA, new TokenValue(yytext()));
}
case -14:
break;
case 14:
{
return new Symbol(sym.SHARP, new TokenValue(yytext()));
}
case -15:
```

```
break;
case 15:
{
String str = yytext().substring(1, yytext().length()-1);
return new Symbol(sym.MY_STRING, new MyString(str));
}
case -16:
break;
case 16:
{
return new Symbol(sym.PAR, new TokenValue(yytext()));
}
case -17:
break;
case 17:
{
yybegin(COMMENTS);
}
case -18:
break;
case 18:
{
return new Symbol(sym.ITEM, new Item(yytext()));
}
case -19:
break;
case 19:
{
return new Symbol(sym.COMM_VAR, new CommunicationVariable(yytext()));
}
case -20:
break;
case 20:
{
return new Symbol(sym.FUNCTOR, new Functor(yytext()));
}
case -21:
break;
case 21:
{
return new Symbol(sym.GET, new TokenValue(yytext()));
```

```
}
case -22:
break;
case 22:
{
return new Symbol(sym.ASK, new TokenValue(yytext()));
}
case -23:
break;
case 23:
{
return new Symbol(sym.TELL, new TokenValue(yytext()));
}
case -24:
break;
case 24:
{
return new Symbol(sym.NASK, new TokenValue(yytext()));
}
case -25:
break;
case 25:
{
yybegin(YYINITIAL);
}
case -26:
break;
case 26:
{}
case -27:
break;
case 28:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -28:
break;
case 29:
{
ErrorManaging.error(ErrorManaging.E_UNMATCHED);
return new Symbol(sym.error, null);
```

```
}
case -29:
break;
case 31:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -30:
break;
case 33:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -31:
break;
case 35:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -32:
break;
case 37:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -33:
break;
case 39:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -34:
break;
case 41:
{
return new Symbol(sym.CHARACTER, new TokenValue(yytext()));
}
case -35:
break;
default:
yy_error(YY_E_INTERNAL,false);
```

```
case -1:
}
yy_initial = true;
yy_state = yy_state_dtrans[yy_lexical_state];
yy_next_state = YY_NO_STATE;
yy_last_accept_state = YY_NO_STATE;
yy_mark_start();
yy_this_accept = yy_acpt[yy_state];
if (YY_NOT_ACCEPT != yy_this_accept) {
yy_last_accept_state = yy_state;
yy_mark_end();
}
}
}
}
}
}
```

Annexe D

Le parseur de NFG généré par Cup

D.1 Le parseur dans un fichier

Voici le fichier `parser.java` généré par Cup à partir des spécifications présentées dans `MyParser.cup` de l'annexe B.

```
//----------------------------------------------------
// The following code was generated by CUP v0.10k
// Mon Aug 29 02:40:38 CEST 2005
//----------------------------------------------------

package NFGTool;

import java_cup.runtime.*;

/** CUP v0.10k generated parser.
  * @version Mon Aug 29 02:40:38 CEST 2005
  */
public class parser extends java_cup.runtime.lr_parser {

  /** Default constructor. */
  public parser() {super();}

  /** Constructor which sets the default scanner. */
  public parser(java_cup.runtime.Scanner s) {super(s);}
```

```
/** Production table. */
protected static final short _production_table[][] =
  unpackFromStrings(new String[] {
  "\000\025\000\002\002\004\000\002\003\004\000\002\003" +
  "\003\000\002\012\002\000\002\004\005\000\002\005\003" +
  "\000\002\005\005\000\002\005\005\000\002\005\005\000" +
  "\002\005\005\000\002\006\006\000\002\006\006\000\002" +
  "\006\006\000\002\006\006\000\002\007\006\000\002\010" +
  "\005\000\002\010\003\000\002\011\005\000\002\011\005" +
  "\000\002\011\005\000\002\011\005" });

/** Access to production table. */
public short[][] production_table() {return _production_table;}

/** Parse-action table. */
protected static final short[][] _action_table =
  unpackFromStrings(new String[] {
  "\000\057\000\014\012\012\025\005\026\011\027\014\030" +
  "\010\001\002\000\014\006\ufffc\007\ufffc\010\ufffc\011\
  ufffc" + "\013\ufffc\001\002\000\004\012\057\001\002\000\
  012\006" + "\043\007\040\010\042\011\ufffe\001\002\000\
  016\002\uffff" + "\012\uffff\025\uffff\026\uffff\027\
  uffff\030\uffff\001\002\000" + "\004\012\052\001\002\000\
  004\012\047\001\002\000\014" + "\012\012\025\005\026\011\
  027\014\030\010\001\002\000" + "\016\002\035\012\012\025\
  005\026\011\027\014\030\010" + "\001\002\000\004\012\015\
  001\002\000\004\021\016\001" + "\002\000\004\012\021\001\
  002\000\004\013\020\001\002" + "\000\014\006\ufff5\007\
  ufff5\010\ufff5\011\ufff5\013\ufff5\001" + "\002\000\004\
  017\022\001\002\000\004\024\030\001\002" + "\000\006\013\
  ufff1\031\026\001\002\000\004\013\025\001" + "\002\000\006\
  013\ufff3\031\ufff3\001\002\000\004\017\022" + "\001\002\
  000\004\013\ufff2\001\002\000\012\015\032\016" + "\034\020\
  031\021\016\001\002\000\006\013\uffee\031\uffee" + "\001\
  002\000\006\013\ufff0\031\ufff0\001\002\000\006\013" +
  "\uffed\031\uffed\001\002\000\006\013\uffef\031\uffef\001\
  002" + "\000\004\002\001\001\002\000\016\002\000\012\000\
  025" + "\000\026\000\027\000\030\000\001\002\000\012\006\
  043" + "\007\040\010\042\013\041\001\002\000\014\012\012\
  025" + "\005\026\011\027\014\030\010\001\002\000\014\006\
  ufffb" + "\007\ufffb\010\ufffb\011\ufffb\013\ufffb\001\002\
```

```
  000\014\012" + "\012\025\005\026\011\027\014\030\010\001\
  002\000\014" + "\012\012\025\005\026\011\027\014\030\010\
  001\002\000" + "\014\006\ufffa\007\ufffa\010\ufffa\011\
  ufffa\013\ufffa\001\002" + "\000\014\006\ufff8\007\ufff8\
  010\ufff8\011\ufff8\013\ufff8\001" + "\002\000\014\006\
  ufff9\007\ufff9\010\ufff9\011\ufff9\013\ufff9" + "\001\
  002\000\004\021\016\001\002\000\004\013\051\001" + "\002\
  000\014\006\ufff6\007\ufff6\010\ufff6\011\ufff6\013\ufff6"
  + "\001\002\000\004\021\016\001\002\000\004\013\054\001" +
  "\002\000\014\006\ufff4\007\ufff4\010\ufff4\011\ufff4\013\
  ufff4" + "\001\002\000\004\011\056\001\002\000\016\002\
  ufffd\012" + "\ufffd\025\ufffd\026\ufffd\027\ufffd\030\
  ufffd\001\002\000\004" + "\021\016\001\002\000\004\013\
  061\001\002\000\014\006" + "\ufff7\007\ufff7\010\ufff7\
  011\ufff7\013\ufff7\001\002" });

/** Access to parse-action table. */
public short[][] action_table() {return _action_table;}

/** <code>reduce_goto</code> table. */
protected static final short[][] _reduce_table =
  unpackFromStrings(new String[] {
  "\000\057\000\012\003\012\004\006\005\005\006\003\001" +
  "\001\000\002\001\001\000\002\001\001\000\004\012\054" +
  "\001\001\000\002\001\001\000\002\001\001\000\002\001" +
  "\001\000\006\005\036\006\003\001\001\000\010\004\035" +
  "\005\005\006\003\001\001\000\002\001\001\000\004\007" +
  "\016\001\001\000\002\001\001\000\002\001\001\000\002" +
  "\001\001\000\006\010\023\011\022\001\001\000\002\001" +
  "\001\000\002\001\001\000\002\001\001\000\002\001\001" +
  "\000\006\010\026\011\022\001\001\000\002\001\001\000" +
  "\004\007\032\001\001\000\002\001\001\000\002\001\001" +
  "\000\002\001\001\000\002\001\001\000\002\001\001\000" +
  "\002\001\001\000\002\001\001\000\006\005\045\006\003" +
  "\001\001\000\002\001\001\000\006\005\044\006\003\001" +
  "\001\000\006\005\043\006\003\001\001\000\002\001\001" +
  "\000\002\001\001\000\002\001\001\000\004\007\047\001" +
  "\001\000\002\001\001\000\002\001\001\000\004\007\052" +
  "\001\001\000\002\001\001\000\002\001\001\000\002\001" +
  "\001\000\002\001\001\000\004\007\057\001\001\000\002" +
  "\001\001\000\002\001\001" });
```

```
/** Access to <code>reduce_goto</code> table. */
public short[][] reduce_table() {return _reduce_table;}

/** Instance of action encapsulation class. */
protected CUP$parser$actions action_obj;

/** Action encapsulation object initializer. */
protected void init_actions()
  {
    action_obj = new CUP$parser$actions(this);
  }

/** Invoke a user supplied parse action. */
public java_cup.runtime.Symbol do_action(
  int                        act_num,
  java_cup.runtime.lr_parser parser,
  java.util.Stack            stack,
  int                        top)
  throws java.lang.Exception
{
  /* call code in generated class */
  return action_obj.CUP$parser$do_action(act_num, parser,
  stack, top);
}

/** Indicates start state. */
public int start_state() {return 0;}
/** Indicates start production. */
public int start_production() {return 0;}

/** <code>EOF</code> Symbol index. */
public int EOF_sym() {return 0;}

/** <code>error</code> Symbol index. */
public int error_sym() {return 1;}


/** Scan to get the next Symbol. */
public java_cup.runtime.Symbol scan()
  throws java.lang.Exception
```

```
    {
 return lexer.next_token();
    }


Lexer lexer;

public parser(Lexer lexer) {
this.lexer = lexer;
}

/* Une méthode par gérer les messages d'erreur émis par le parseur */
public void report_error(String message, Object info) {
StringBuffer m = new StringBuffer("Error");

if (info instanceof java_cup.runtime.Symbol) {
java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
if (s.left >= 0) {
m.append(" in line "+(s.left+1));
if (s.right >= 0)
m.append(", column "+(s.right+1));
}
}
m.append(" : "+message);
System.err.println(m);
}

public void report_fatal_error(String message, Object info) {
report_error(message, info);
System.exit(1);
}

}

/** Cup generated class to encapsulate user supplied action code.*/
class CUP$parser$actions {
  private final parser parser;

  /** Constructor */
  CUP$parser$actions(parser parser) {
    this.parser = parser;
```

```
}

/** Method with the actual generated action code. */
public final java_cup.runtime.Symbol CUP$parser$do_action(
  int                        CUP$parser$act_num,
  java_cup.runtime.lr_parser CUP$parser$parser,
  java.util.Stack            CUP$parser$stack,
  int                        CUP$parser$top)
  throws java.lang.Exception
  {
    /* Symbol object for return from actions */
    java_cup.runtime.Symbol CUP$parser$result;

    /* select the action based on the action number */
    switch (CUP$parser$act_num)
      {
        /*. . . . . .*/
        case 20: //value_funct_arg::=ITEM EQUAL_SIGN psi_term
          {
            Value RESULT = null;
            int ileft = ((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-2)).left;
            int iright = ((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-2)).right;
            Item i = (Item)((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-2)).value;
            int pleft = ((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-0)).left;
            int pright = ((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-0)).right;
            PsiTerm p = (PsiTerm)((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
            RESULT = new Value(i,p);

            CUP$parser$result = new java_cup.runtime.Symbol(7/
            *value_funct_arg*/, ((java_cup.runtime.Symbol)
            CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
            ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
            (CUP$parser$top-0)).right, RESULT);
          }
        return CUP$parser$result;
```

```
/*. . . . . .*/
case 19: //value_funct_arg ::= ITEM EQUAL_SIGN COMM_VAR
  {
    Value RESULT = null;
    int ileft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).left;
    int iright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).right;
    Item i = (Item)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).value;
    int cleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int cright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).right;
    CommunicationVariable c = (CommunicationVariable)
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).value;
    RESULT = new Value(i, c.getValue());

    CUP$parser$result = new java_cup.runtime.Symbol(7
    /*value_funct_arg*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 18: // value_funct_arg ::= ITEM EQUAL_SIGN MY_STRING
  {
    Value RESULT = null;
    int ileft = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).left;
    int iright = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).right;
    Item i = (Item)((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).value;
    int sleft = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int sright = ((java_cup.runtime.Symbol)CUP$parser$stack.
```

```
    elementAt(CUP$parser$top-0)).right;
    MyString s = (MyString)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    RESULT = new Value(i, s.getValue());

    CUP$parser$result = new java_cup.runtime.Symbol(7
    /*value_funct_arg*/,((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 17: // value_funct_arg ::= ITEM EQUAL_SIGN NUMBER
  {
    Value RESULT = null;
    int ileft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).left;
    int iright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).right;
    Item i=(Item)((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).value;
    int nleft = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int nright = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).right;
    MyNumber n = (MyNumber)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    RESULT = new Value(i, n.getValue());

    CUP$parser$result = new java_cup.runtime.Symbol(7
    /*value_funct_arg*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 16: // value_funct_arg_list ::= value_funct_arg
```

```
    {
      ValueList RESULT = null;
      int vleft=((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-0)).left;
      int vright=((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-0)).right;
      Value v = (Value)((java_cup.runtime.Symbol)
      CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
      RESULT = new ValueList(v);

      CUP$parser$result = new java_cup.runtime.Symbol(6
      /*value_funct_arg_list*/, ((java_cup.runtime.Symbol)
      CUP$parser$stack.elementAt(CUP$parser$top-0)).left,
      ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
      (CUP$parser$top-0)).right, RESULT);
    }
  return CUP$parser$result;

  /*. . . . . .*/
  case 15: // value_funct_arg_list ::= value_funct_arg COMMA
  value_funct_arg_list
    {
      ValueList RESULT = null;
      int vleft = ((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-2)).left;
      int vright =((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-2)).right;
      Value v=(Value)((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-2)).value;
      int lleft = ((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-0)).left;
      int lright = ((java_cup.runtime.Symbol)CUP$parser$stack.
      elementAt(CUP$parser$top-0)).right;
      ValueList l = (ValueList)((java_cup.runtime.Symbol)
      CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
      RESULT = new ValueList(v, l);

      CUP$parser$result = new java_cup.runtime.Symbol(6
      /*value_funct_arg_list*/,((java_cup.runtime.Symbol)
      CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
      ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
```

```
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 14: // psi_term ::= FUNCTOR LPAREN
    value_funct_arg_list RPAREN
  {
    PsiTerm RESULT = null;
    int fleft = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-3)).left;
    int fright =((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-3)).right;
    Functor f = (Functor)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-3)).value;
    int vlleft = ((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).left;
    int vlright =((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).right;
    ValueList vl = (ValueList)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    RESULT = new PsiTerm(f, ValueList.buildValueArray(vl));

    CUP$parser$result = new java_cup.runtime.Symbol(5
    /*psi_term*/,((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-3)).left,((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 13: // comm_action ::= NASK LPAREN psi_term RPAREN
  {
    Agent RESULT = null;
    int pleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).left;
    int pright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).right;
    PsiTerm p = (PsiTerm)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    RESULT = new Agent("nask", p);
```

```
    CUP$parser$result = new java_cup.runtime.Symbol(4
    /*comm_action*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-3)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 12: // comm_action ::= ASK LPAREN psi_term RPAREN
  {
    Agent RESULT = null;
    int pleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).left;
    int pright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).right;
    PsiTerm p = (PsiTerm)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    RESULT = new Agent("ask", p);

    CUP$parser$result = new java_cup.runtime.Symbol(4
    /*comm_action*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-3)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 11: // comm_action ::= GET LPAREN psi_term RPAREN
  {
    Agent RESULT = null;
    int pleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).left;
    int pright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).right;
    PsiTerm p = (PsiTerm)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    RESULT = new Agent("get", p);
```

```
        CUP$parser$result = new java_cup.runtime.Symbol(4
        /*comm_action*/, ((java_cup.runtime.Symbol)
        CUP$parser$stack.elementAt(CUP$parser$top-3)).left,
        ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
        (CUP$parser$top-0)).right, RESULT);
      }
    return CUP$parser$result;

    /*. . . . . .*/
    case 10: // comm_action ::= TELL LPAREN psi_term RPAREN
      {
        Agent RESULT = null;
        int pleft=((java_cup.runtime.Symbol)CUP$parser$stack.
        elementAt(CUP$parser$top-1)).left;
        int pright=((java_cup.runtime.Symbol)CUP$parser$stack.
        elementAt(CUP$parser$top-1)).right;
        PsiTerm p = (PsiTerm)((java_cup.runtime.Symbol)
        CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
        RESULT = new Agent("tell", p);

        CUP$parser$result = new java_cup.runtime.Symbol(4
        /*comm_action*/, ((java_cup.runtime.Symbol)
        CUP$parser$stack.elementAt(CUP$parser$top-3)).left,
        ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
        (CUP$parser$top-0)).right, RESULT);
      }
    return CUP$parser$result;

    /*. . . . . .*/
    case 9: // agent ::= agent ALT agent
      {
        Agent RESULT = null;
        int aleft=((java_cup.runtime.Symbol)CUP$parser$stack.
        elementAt(CUP$parser$top-2)).left;
        int aright=((java_cup.runtime.Symbol)CUP$parser$stack.
        elementAt(CUP$parser$top-2)).right;
        Agent a = (Agent)((java_cup.runtime.Symbol)
        CUP$parser$stack.elementAt(CUP$parser$top-2)).value;
        int bleft=((java_cup.runtime.Symbol)CUP$parser$stack.
        elementAt(CUP$parser$top-0)).left;
        int bright=((java_cup.runtime.Symbol)CUP$parser$stack.
```

```
    elementAt(CUP$parser$top-0)).right;
    Agent b = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    Agent res = new Agent(a, b);
    Agent.alternativeComposition(res, a, b);
    RESULT = res;

    CUP$parser$result = new java_cup.runtime.Symbol(3
    /*agent*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 8: // agent ::= agent PAR agent
  {
    Agent RESULT = null;
    int aleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).left;
    int aright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).right;
    Agent a = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).value;
    int bleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int bright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).right;
    Agent b = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    Agent res = new Agent(a, b);
    Agent.parallelComposition(res, a, b);
    RESULT = res;

    CUP$parser$result = new java_cup.runtime.Symbol(3
    /*agent*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
```

```
return CUP$parser$result;

/*. . . . . .*/
case 7: // agent ::= agent SEQ agent
  {
    Agent RESULT = null;
    int aleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).left;
    int aright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).right;
    Agent a=(Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).value;
    int bleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int bright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).right;
    Agent b = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    Agent res = new Agent(a, b);
    Agent.sequentialComposition(res, a, b);
    RESULT = res;

    CUP$parser$result = new java_cup.runtime.Symbol(3
    /*agent*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 6: // agent ::= LPAREN agent RPAREN
  {
    Agent RESULT = null;
    int aleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).left;
    int aright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-1)).right;
    Agent a = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    RESULT = a;
```

```
    CUP$parser$result = new java_cup.runtime.Symbol(3
    /*agent*/,((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 5: // agent ::= comm_action
  {
    Agent RESULT = null;
    int cleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int cright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).right;
    Agent c = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    Agent.processPrimitive(c);
    RESULT = c;

    CUP$parser$result = new java_cup.runtime.Symbol(3
    /*agent*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 4: // agent_part ::= agent NT$0 SHARP
  {
    Object RESULT = null;
    // propagate RESULT from NT$0
    if ( ((java_cup.runtime.Symbol) CUP$parser$stack.
         elementAt(CUP$parser$top-1)).value != null )
    RESULT = (Object) ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    int aleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).left;
```

```
    int aright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-2)).right;
    Agent a = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).value;

    CUP$parser$result = new java_cup.runtime.Symbol(2
    /*agent_part*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-2)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 3: // NT$0 ::=
  {
    Object RESULT = null;
    int aleft=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).left;
    int aright=((java_cup.runtime.Symbol)CUP$parser$stack.
    elementAt(CUP$parser$top-0)).right;
    Agent a = (Agent)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).value;
    Agent.generateFileForAgent(a);

    CUP$parser$result = new java_cup.runtime.Symbol(8
    /*NT$0*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).right,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 2: // agent_list ::= agent_part
  {
    Object RESULT = null;

    CUP$parser$result = new java_cup.runtime.Symbol(1
    /*agent_list*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-0)).left,
```

```
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 1: // agent_list ::= agent_list agent_part
  {
    Object RESULT = null;

    CUP$parser$result = new java_cup.runtime.Symbol(1
    /*agent_list*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
return CUP$parser$result;

/*. . . . . .*/
case 0: // $START ::= agent_list EOF
  {
    Object RESULT = null;
    int start_valleft = ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).left;
    int start_valright = ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).right;
    Object start_val = (Object)((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).value;
    RESULT = start_val;

    CUP$parser$result = new java_cup.runtime.Symbol(0
    /*$START*/, ((java_cup.runtime.Symbol)
    CUP$parser$stack.elementAt(CUP$parser$top-1)).left,
    ((java_cup.runtime.Symbol)CUP$parser$stack.elementAt
    (CUP$parser$top-0)).right, RESULT);
  }
/* ACCEPT */
CUP$parser$parser.done_parsing();
return CUP$parser$result;

/* . . . . . .*/
```

```
            default:
              throw new Exception(
                 "Invalid action number found in internal parse
                  table");
          }
      }
}
```

D.2 Un fichier associé au parseur

Voici le fichier `sym.java` généré par Cup en même temps que `parser.java`. Ce fichier contient un identifiant pour chaque terminal figurant dans les spécifications contenues dans `MyParser.cup`.

```
//----------------------------------------------------
// The following code was generated by CUP v0.10k
// Mon Aug 29 02:40:38 CEST 2005
//----------------------------------------------------

package NFGTool;

/** CUP generated class containing symbol constants. */
public class sym {
  /* terminals */
  public static final int ALT = 6;
  public static final int SHARP = 7;
  public static final int RPAREN = 9;
  public static final int ITEM = 13;
  public static final int CHARACTER = 2;
  public static final int SLASH = 10;
  public static final int NASK = 22;
  public static final int QUOTATION = 17;
  public static final int PRIMITIVE = 16;
  public static final int COMM_VAR = 14;
  public static final int LPAREN = 8;
  public static final int COMMA = 23;
  public static final int EOF = 0;
  public static final int NUMBER = 11;
  public static final int ASK = 21;
```

```
  public static final int PAR = 5;
  public static final int SEQ = 4;
  public static final int error = 1;
  public static final int MY_STRING = 12;
  public static final int TELL = 19;
  public static final int WHITE_SPACE = 3;
  public static final int GET = 20;
  public static final int EQUAL_SIGN = 18;
  public static final int FUNCTOR = 15;
}
```

Annexe E

Le code Java de NFG

Voici le code source des différentes classes qui composent l'application NFG.

E.1 Le fichier `MyNumber.java`

```
package NFGTool;

/**
 * Cette classe définit le lexème NUMBER
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

class MyNumber extends TokenValue
{
Integer value;

MyNumber(String val) {
super(val);
value = new Integer(val);
}

public Integer getValue() {
return value;
}
```

```
public void setValue(Integer numb) {
value = numb;
}
}
```

E.2 Le fichier MyString.java

```
package NFGTool;

/**
 * Cette classe définit le lexème MY_STRING
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

class MyString extends TokenValue {
String value;

MyString(String txt) {
super(txt);
value = txt;
}

public String getValue() {
return value;
}

public void setValue(String text) {
value = text;
}
}
```

E.3 Le fichier CommunicationVariable.java

```
package NFGTool;
```

```
/**
 * Cette classe définit un variable de communication
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

class CommunicationVariable extends TokenValue
{
private String value;
private String variable;
private int position;
private String type;

CommunicationVariable(String text) {
super(text);
this.value = text;
this.variable = text.substring(0,1);
this.position = new Integer(text.substring(1)).intValue();
}

CommunicationVariable(String var, int pos) {
this.variable = var;
this.position = pos;
}

public String getValue() {
return value;
}

public void setValue(String val) {
value = val;
}

public String getVariable() {
return variable;
}

public void setVariable(String var) {
variable = var;
}
```

```
public int getPosition() {
return position;
}

public void setPosition(int pos) {
position = pos;
}

public String getType() {
return type;
}

public void setType(String typ) {
type = typ;
}

public boolean isEqualsTo(CommunicationVariable comm_var2) {
return (this.variable.equals(comm_var2.getVariable())) &&
(this.position == comm_var2.getPosition());
}
}
```

E.4 Le fichier PsiTerm.java

```
package NFGTool;

/**
 * Cette classe définit le psi-terme
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

import java.util.*;

class PsiTerm extends TokenValue
{
private Functor functor;
```

```
private Value[] values_of_psiterm;
private PsiTerm parent;
private boolean is_child;

private boolean is_closed;
private Place place_assoc;
private Agent agent_assoc;
private int psi_number;

PsiTerm(Functor functor, Value[] values) {
super(functor, values);
this.functor = functor;
this.values_of_psiterm = values;
Agent.psiterm_number++;
this.psi_number = Agent.psiterm_number;
}

public Functor getFunctor() {
return functor;
}

public void setFunctor(Functor func) {
functor = func;
}

public Value[] getValuesOfPsiTerm() {
return this.values_of_psiterm;
}

public void setValuesOfPsiTerm(Value[] values) {
this.values_of_psiterm = values;
}

public PsiTerm getParent() {
return parent;
}

public void setParent(PsiTerm psi) {
parent = psi;
}
```

```
public boolean getIsChild() {
return is_child;
}

public void setIsChild(boolean bool) {
is_child = bool;
}

public Place getPlaceAssoc() {
return place_assoc;
}

public void setPlaceAssoc(Place place) {
place_assoc = place;
}

public int getPsiNumber() {
return psi_number;
}

public void setPsiNumber(int number) {
psi_number = number;
}

public boolean isClosed() {
Value[] tab = this.values_of_psiterm;
boolean isclosed = true;
for (int i = 0; i < tab.length ; i++)
{
isclosed = isclosed && (!tab[i].getIsCommVariable());
}
return isclosed;
}

/* Méthode qui compare deux psi-termes */
public boolean isEqualsTo(PsiTerm psi2) {
boolean tmp1 = (this.functor).isEqualsTo(psi2.getFunctor());
boolean tmp2 = true;
if (this.values_of_psiterm.length == psi2.getValuesOfPsiTerm().
length)
{
```

```
for (int i = 0; i < psi2.getValuesOfPsiTerm().length; i++)
{
tmp2 = tmp2 && this.values_of_psiterm[i].isEqualsTo(psi2.
getValuesOfPsiTerm()[i]);
}
return tmp1 && tmp2;
}
else
{
if (this.values_of_psiterm.length < psi2.getValuesOfPsiTerm().length)
{
boolean tmp3 = false;
for (int j = 0; j < this.values_of_psiterm.length; j++)
{
Item tmp_item = this.values_of_psiterm[j].getItem();
for (int k = 0; k < psi2.getValuesOfPsiTerm().length; k++)
{
Item tmp_item_2 = psi2.getValuesOfPsiTerm()[k].getItem();
if (tmp_item.isEqualsTo(tmp_item_2))
{
tmp3 = true;
tmp2 = tmp2 && this.values_of_psiterm[j].isEqualsTo(psi2.
getValuesOfPsiTerm()[k]);
}
}
}
return tmp1 && tmp2 && tmp3;
}
}
return false;
}

public Value[] makeArrayFromAValue(Value v) {
Value[] result = new Value[1];
result[0] = v;

return result;
}

public void addValue(Value value) {
int size = this.values_of_psiterm.length;
```

```
Value[] result_array = new Value[size + 1];
result_array[size] = value;

this.setValuesOfPsiTerm(result_array);
}
}
```

E.5 Le fichier Item.java

```
package NFGTool;

/**
 * Cette classe définit les items du langage
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

class Item extends TokenValue
{
private String itemname;
private int itemposition;

Item(String itemname) {
this(itemname, -1);
}

Item(String itemname, int itemposition) {
super(itemname);
this.itemname = itemname;
this.itemposition = itemposition;
}

public String getItemName() {
return itemname;
}

public int getItemPosition() {
return itemposition;
```

```
}

public void setItemName(String name) {
itemname = name;
}

public void setItemPosition(int position) {
itemposition = position;
}

public boolean isEqualsTo(Item item2) {
return this.itemname.equals(item2.getItemName()) &&
this.itemposition == item2.getItemPosition();
}

public boolean isWellFormedItem() {
String tmp0 = (new Integer(this.itemposition)).toString();
try
{
Integer tmp1 = new Integer(tmp0);
/* On a imposé que les noms d'items commencent pas
'x', 'y' ou 'z' */
if (this.itemname.equals("x") || this.itemname.equals("y") ||
this.itemname.equals("z"))
{
return true;
}
return false;
}
catch (Exception e)
{
e.printStackTrace();
}
return false;
}
}
```

E.6 Le fichier Functor.java

```
package NFGTool;

/**
 * Cette classe définit un foncteur tel qu'il apparaît
 * dans le formalisme
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

class Functor extends TokenValue
{
private String functorname;
private int functorarity;

Functor(String text) {
super(text);
this.functorname = text.substring(0,1);
this.functorarity = new Integer(text.substring(2)).intValue();
}

public String getFunctorName() {
return functorname;
}

public void setFunctorName(String name) {
functorname = name;
}

public int getFunctorArity() {
return functorarity;
}

public void setFunctorArity(int arity) {
functorarity = arity;
}

public boolean isEqualsTo(Functor funct2) {
return this.functorname.equals(funct2.getFunctorName()) &&
```

```
this.functorarity == funct2.getFunctorArity();
}
}
```

E.7 Le fichier Value.java

```
package NFGTool;

/**
 * Cette classe représente décrit les valeurs
 * que peuvent prendre les items d'un psi-terme
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */
class Value extends TokenValue
{
/* Les champs d'une valeur */
private Item item;
private Object value;

/* Indication du type de la valeur */
private boolean isnumber = false;
private boolean isstring = false;
private boolean ispsiterm = false;
private boolean iscomm_variable = false;

/* Peut donner l'info concernant le psi-terme
qui contient cette valeur */
private PsiTerm psiterm_assoc;

Value(Item item, String value) {
super(item, value);
this.item = item;
this.value = value;
this.isstring = true;
}

Value(Item item, Integer value) {
super(item, value);
```

```
this.item = item;
this.value = value;
this.isnumber = true;
}

Value(Item item, PsiTerm value) {
super(item, value);
this.item = item;
this.value = value;
this.ispsiterm = true;
}

Value(Item item, CommunicationVariable value) {
super(item, value);
this.item = item;
this.value = value;
this.iscomm_variable = true;
}

public Object getValue() {
return value;
}

public void setValue(Object value) {
this.value = value;
}

public Item getItem() {
return item;
}

public void setItem(Item item) {
this.item = item;
}

public boolean getIsNumber() {
return isnumber;
}

public void setIsNumber(boolean number) {
isnumber = number;
```

```
}

public boolean getIsString() {
return isstring;
}

public void setIsString(boolean string) {
isstring = string;
}

public boolean getIsPsiTerm() {
return ispsiterm;
}

public void setIsPsiTerm(boolean psi) {
ispsiterm = psi;
}

public boolean getIsCommVariable() {
return iscomm_variable;
}

public void setIsCommVariable(boolean comm_var) {
iscomm_variable = comm_var;
}

public PsiTerm getPsiTermAssoc() {
return psiterm_assoc;
}

public void setPsiTermAssoc(PsiTerm psi) {
psiterm_assoc = psi;
}

public String findClass(Object obj) {
String tmp = obj.getClass().getName();
int limit = tmp.lastIndexOf('.');
if (limit != -1)
{
return tmp.substring(limit + 1);
}
```

```
return tmp;
}

public boolean isNumber() {
String tmp = findClass(this);
if (tmp.equals("Integer"))
{
return true;
}
return false;
}

public boolean isNumber(Object value) {
return value instanceof Integer;
}

public boolean isCommunicationVariable() {
String tmp = findClass(this);
if (tmp.equals("CommunicationVariable"))
{
return true;
}
return false;
}

public boolean isCommunicationVariable(Object value) {
return value instanceof CommunicationVariable;
}

public boolean isPsiTerm() {
String tmp = findClass(this);
if (tmp.equals("PsiTerm"))
{
return true;
}
return false;
}

public boolean isPsiTerm(Object value) {
return value instanceof PsiTerm;
}
```

```
public boolean isString() {
String tmp = findClass(this);
if (tmp.equals("String"))
{
return true;
}
return false;
}

public boolean isString(Object value) {
return value instanceof String;
}

/* Méthode qui compare deux valeurs */
public boolean isEqualsTo(Value value2) {
if (this.isstring && value2.getIsString())
{
String tmp1 = (String) value2.getValue();
String tmp11 = (String) this.value;
return this.item.isEqualsTo(value2.getItem())
&& tmp11.equals(tmp1);
}
else if (this.isnumber && value2.getIsNumber())
{
int tmp2 = ((Integer) value2.getValue()).intValue();
int tmp22 = ((Integer) this.value).intValue();
return this.item.isEqualsTo(value2.getItem())
&& tmp22 == tmp2;
}
else if (this.ispsiterm && value2.getIsPsiTerm())
{
PsiTerm tmp3 = (PsiTerm) value2.getValue();
PsiTerm tmp33 = (PsiTerm) this.value;
return this.item.isEqualsTo(value2.getItem())
&& tmp33.isEqualsTo(tmp3);
}
else if (this.iscomm_variable && value2.getIsCommVariable())
{
CommunicationVariable tmp4 = (CommunicationVariable)
value2.getValue();
```

```
CommunicationVariable tmp44 = (CommunicationVariable)
this.value;
return this.item.isEqualsTo(value2.getItem()) &&
tmp44.isEqualsTo(tmp4);
}
else return false;
}
}
```

E.8 Le fichier ValueList.java

```
package NFGTool;

/**
 * Cette classe définit des listes de valeurs
 * acceptées dans la définition d'un item
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

class ValueList extends TokenValue
{
Value head;
ValueList tail;

ValueList(Value h) {
this(h, null);
}

ValueList(Value h, ValueList t) {
super(h, t);
head = h;
tail = t;
}

static Value[] buildValueArray(ValueList t) {
if (t.tail == null)
{
Value[] result = new Value[1];
```

```
result[0] = t.head;
return result;
}
else
{
return appendValue(t.head, buildValueArray(t.tail));
}
}

static Value[] appendValue(Value v, Value[] list) {
if (list == null)
{
Value[] res = new Value[1];
res[0] = v;
return res;
}
else
{
Value[] res = new Value[list.length + 1];
for (int i = 0; i < list.length; i++)
{
res[i] = list[i];
}
res[list.length] = v;
return res;
}
}
}
```

E.9 Le fichier Place.java

```
package NFGTool;

/**
 * Cette classe définit une place dans un réseau
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */
```

```
class Place
{
private String placename;
private int placenumber;
private int number_of_tokens = 0;
private int capacity;
private Transition[] pre_transitions;
private Transition[] post_transitions;
private PsiTerm psi_term_assoc;
private boolean is_launching_place = false;
private boolean is_terminal_place = false;
private boolean is_variable_assoc_place = false;
private boolean is_psi_term_assoc_place = false;
private boolean is_agent_assoc_place = false;

Place () {
}

Place (String placename, int number_of_tokens,
Transition[] pre_transitions,
Transition[] post_transitions) {
this.placename = placename;
this.number_of_tokens = number_of_tokens;
this.pre_transitions = pre_transitions;
this.post_transitions = post_transitions;
Agent.place_number += 1;
this.setPlaceNumber(Agent.place_number);
}

Place (Transition[] pre_transitions,
Transition[] post_transitions) {
this.pre_transitions = pre_transitions;
this.post_transitions = post_transitions;
this.is_agent_assoc_place = true;
Agent.place_number += 1;
this.setPlaceNumber(Agent.place_number);
this.setPlaceName("place_"+getPlaceNumber());
}

Place (PsiTerm psiterm) {
this.psi_term_assoc = psiterm;
```

```
this.is_psi_term_assoc_place = true;
Agent.place_number += 1;
this.setPlaceNumber(Agent.place_number);
this.setPlaceName("psi_place_"+psiterm.getPsiNumber());
}

public String getPlaceName() {
return placename;
}

public void setPlaceName(String name){
this.placename = name;
}

public int getPlaceNumber() {
return placenumber;
}

public void setPlaceNumber(int number){
this.placenumber = number;
}

public boolean getIsLaunchingPlace() {
return this.is_launching_place;
}

public void setIsLaunchingPlace(boolean placetype) {
this.is_launching_place = placetype;
}

public boolean getIsTerminalPlace() {
return this.is_terminal_place;
}

public void setIsTerminalPlace(boolean placetype) {
this.is_terminal_place = placetype;
}

public boolean getIsVariableAssociatedPlace() {
return this.is_variable_assoc_place;
}
```

```
public void setIsVariableAssociatedPlace(boolean placetype) {
this.is_variable_assoc_place = placetype;
}

public boolean getIsPsiermAssociatedPlace() {
return this.is_psi_term_assoc_place;
}

public void setIsPsiermAssociatedPlace(boolean placetype) {
this.is_psi_term_assoc_place = placetype;
}

public boolean getIsAgentAssociatedPlace() {
return this.is_agent_assoc_place;
}

public void setIsAgentAssociatedPlace(boolean placetype) {
this.is_agent_assoc_place = placetype;
}

public void setNumberOfTokens(int number) {
this.number_of_tokens = number;
}

public int getNumberOfTokens() {
return number_of_tokens;
}

public Transition[] getPreTransitions() {
return this.pre_transitions;
}

public void setPreTransitions(Transition[] transitions) {
this.pre_transitions = transitions;
}

public Transition[] getPostTransitions() {
return this.post_transitions;
}
```

```
public void setPostTransitions(Transition[] transitions) {
this.post_transitions = transitions;
}

public PsiTerm getPsiTermAssoc() {
if (this.is_psi_term_assoc_place)
return psi_term_assoc;
else
return null;
}

public void setPsiTermAssoc(PsiTerm psi) {
psi_term_assoc = psi;
}

public void addTokens(int tokens_added) {
this.number_of_tokens += tokens_added;
}

public void removeTokens(int tokens_removed) {
this.number_of_tokens -= tokens_removed;
}

public void removeAllTokens() {
setNumberOfTokens(0);
}

public void addToPreTransition(Transition transition_to_add){
int size;
if (this.pre_transitions != null)
size = this.pre_transitions.length;
else
size = 0;
Transition[] result_array = new Transition[size + 1];
for (int i = 0; i < size; i++)
{
result_array[i] = this.pre_transitions[i];
}

// A vérifier à l'exécution...
result_array[size] = transition_to_add;
```

```
this.setPreTransitions(result_array);
}

public void addToPostTransition(Transition transition_to_add){
int size;
if (this.post_transitions != null)
size = this.post_transitions.length;
else
size = 0;

Transition[] result_array = new Transition[size + 1];
for (int i = 0; i < size; i++)
{
result_array[i] = this.post_transitions[i];
}

result_array[size] = transition_to_add;
this.setPostTransitions(result_array);
}

public String buildName() {
if (this.getPlaceName().equals(null))
{
return "place_"+this.getPlaceNumber();
}
return this.getPlaceName();
}
}
```

E.10 Le fichier `Transition.java`

```
package NFGTool;
/**
 * Cette classe définit une transition dans le formalisme
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */
```

```
import java.util.*;

class Transition
{
/* Le nom de la transition. On le crée s'il n'est pas donné */
String transitionname;
int transitionnumber;

/* Le tableau indiquant les nombres de tokens provenant
de différentes pré-places de la transition.
Ceci pourrait servir par la suite */
int[] tokens_received;//Faire gaffe à l'ordre !

/* Le tableaux indiquant les nombres de tokens produits par la
transition pour ses différentes post-places */
int[] tokens_produced;

boolean is_launching_transition = false;
boolean is_terminal_transition = false;

Place[] pre_places;
Place[] post_places;
Place[] contextual_places;
Place[] inhibitor_places;

Transition() {
}

Transition(String transitionname, Place[] pre_places,
Place[] post_places) {
this.transitionname = transitionname;
this.pre_places = pre_places;
this.post_places = post_places;
Agent.transition_number += 1;
this.transitionnumber = Agent.transition_number;
this.setTransitionName("trans_"+getTransitionNumber());
}

Transition(Place[] pre_places, Place[] post_places) {
this(pre_places, null, null, post_places);
}
```

```
Transition(Place[] pre_places, Place[] contextual_places,
Place[] inhibitor_places, Place[] post_places) {
this.pre_places = pre_places;
this.contextual_places = contextual_places;
this.inhibitor_places = inhibitor_places;
this.post_places = post_places;
Agent.transition_number += 1;
this.transitionnumber = Agent.transition_number;
this.setTransitionName("trans_"+getTransitionNumber());
}

public String getTransitionName() {
return transitionname;
}

public void setTransitionName(String name){
this.transitionname = name;
}

public int getTransitionNumber() {
return transitionnumber;
}

public void setTransitionNumber(int number){
this.transitionnumber = number;
}

public void setLaunchingTransition(boolean transitiontype) {
this.is_launching_transition = transitiontype;
}

public boolean isLaunchingTransition() {
Place[] preplaces = this.getPrePlaces();
boolean tmp_bool = false;

if (preplaces != null)
{
for (int i = 0; i < preplaces.length; i++)
{
tmp_bool = tmp_bool ||
```

```
preplaces[i].getIsLaunchingPlace();
}
}
return tmp_bool;
}

public void setTerminalTransition(boolean transitiontype) {
this.is_terminal_transition = transitiontype;
}

public Place[] getPrePlaces(){
return pre_places;
}

public Place[] getPostPlaces(){
return post_places;
}

public Place[] getContextualPlaces(){
return contextual_places;
}

public Place[] getInhibitorPlaces(){
return inhibitor_places;
}

public void setPrePlaces(Place[] pre_set){
this.pre_places = pre_set;
for (int i = 0; i < pre_set.length; i++)
{
pre_set[i].addToPostTransition(this);
}
}

public void setPostPlaces(Place[] post_set){
this.post_places = post_set;
for (int i = 0; i < post_set.length; i++)
{
post_set[i].addToPreTransition(this);
}
}
```

```
public void addPrePlace(Place place_to_add){
Vector tmp_array = new Vector();

for (int i = 0; i < this.pre_places.length; i++)
{
tmp_array.addElement(this.pre_places[i]);
}

tmp_array.addElement(place_to_add);

for (int i = 0; i < tmp_array.size(); i++)
{
this.pre_places[i] = (Place) tmp_array.elementAt(i);
}
}

public void addPostPlace(Place place_to_add){
int size = this.post_places.length;
this.post_places[size] = place_to_add;
}

public String buildName() {
if (this.getTransitionName().equals(null))
{
return "trans_"+this.getTransitionNumber();
}
return this.getTransitionName();
}
}
```

E.11 Le fichier Agent.java

```
package NFGTool;

/**
 * Cette classe représente le traitement des différents
 * types d'agents
 *
 * @version last update: 24/08/2005
```

```
 * @author Mayala Lusilabo M. Frumence
 */

import javax.swing.*;
import java.util.*;
import java.io.*;
import java.text.*;

public class  Agent extends TokenValue
{
/* Compteurs pour les agents de différents types */
static int agent_number = 0;
static int tell_primitive = 0;
static int get_primitive = 0;
static int ask_primitive = 0;
static int nask_primitive = 0;
static int sequential_agent = 0;
static int parallel_agent = 0;
static int alternative_agent = 0;

/* Compteurs pour toutes les places, toutes les
transitions et de tous mes psi-termes respectivement */
static int place_number = 0;
static int transition_number = 0;
static int psiterm_number = 0;


/* Champ contenant le nom de la primitive principale
de l'agent si elle existe */
String primitive_type;
PsiTerm psiterm_assoc;

/* Le sous-agent gauche et le sous-agent droit */
Agent leftagent;
Agent rightagent;

/* Indication si l'agent est une primitive ou pas */
boolean isprimitive;

/* Tableau de toutes les places de l'agent */
Place[] allplaces;
```

```
/* Tableau  pour toutes les transitions de l'agent */
Transition[] alltransitions;

/* La place de lancement de l'agent */
Place launching_place;

/* La place terminale de l'agent */
Place terminal_place;

/* Tableau des transitions de lancement */
Transition[] launching_transitions;

/* Tableau des transitions terminales */
Transition[] terminal_transitions;

/** Un constructeur avec des sous-agents à utiliser pour
* l'instanciation d'un agent complexe défini au moyen d'une
* opération de composition
* @param leftagent l'agent de gauche.
* @param rightagent l'agent de droite
    */
Agent(Agent leftagent, Agent rightagent) {
this(leftagent, rightagent, "agent_"+agent_number);
}

/** Un constructeur avec des sous-agents à utiliser pour
* l'instanciation d'un agent ... (Peut-être inutile)
* @param leftagent l'agent de gauche.
* @param rightagent l'agent de droite
* @param primitive la primitive de l'agent pour le cas élémentaire
    */
Agent(Agent leftagent, Agent rightagent, String primitive) {
super(primitive, leftagent, rightagent);
this.primitive_type = primitive;
this.leftagent = leftagent;
this.rightagent = rightagent;
}

/** Un constructeur avec des sous-agents à utiliser pour
* l'instanciation d'un agent élémentaire défini avec un
```

```
* psi-terme
* @param primitive la primitive pour construire un agent
* élémentaire
* @param psiterm le psi-terme que contient l'agent
    */
Agent(String primitive, PsiTerm psiterm) {
super(primitive, psiterm, null);
this.primitive_type = primitive;
this.psiterm_assoc = psiterm;
this.allplaces = buildAllPsiTermPlaces(psiterm);
this.isprimitive = true;
}

public String getPrimitiveType() {
return primitive_type;
}

public void setPrimitiveType(String prim) {
this.primitive_type = prim;
}

public PsiTerm getPsiTermAssoc() {
return psiterm_assoc;
}

public void setPsiTermAssoc(PsiTerm psi) {
psiterm_assoc = psi;
}

public Agent getLeftAgent() {
return leftagent;
}

public void setLeftAgent(Agent leftagent) {
this.leftagent = leftagent;
}

public Agent getRightAgent() {
return rightagent;
}
```

```
public void setRightAgent(Agent rightagent) {
 this.rightagent =  rightagent;
}

public void isPrimitive(boolean agenttype) {
isprimitive = agenttype;
}

public Place[] getAllPlaces() {
return allplaces;
}

public void setAllPlaces(Place[] places) {
allplaces = places;
}

public Transition[] getAllTransitions() {
return alltransitions;
}

public void setAllTransitions(Transition[] transitions) {
alltransitions = transitions;
}

public Place getLaunchingPlace() {
return launching_place;
}

public void setLaunchingPlace(Place place) {
this.launching_place = place;
}

public Place getTerminalPlace() {
return terminal_place;
}

public void setTerminalPlace(Place place) {
this.terminal_place = place;
}

public Transition[] getLaunchingTransitions() {
```

```
return this.getLaunchingPlace().getPostTransitions();
}

public void setLaunchingTransitions(Transition[] transitions) {
launching_transitions = transitions;
}

public Transition[] getTerminalTransitions() {
return this.getTerminalPlace().getPreTransitions();
}

public void setTerminalTransitions(Transition[] transitions) {
terminal_transitions = transitions;
}

public boolean isPrimitiveAgent() {

//En attendant
return true;
}

/* Construction d'un tableau des places à partir des
deux tableaux */
static public Place[] createAllPlacesArray(Place[] tab1,
Place[] tab2) {
Vector tab = new Vector();

for (int i = 0; i < tab1.length; i++ )
{
//Une condition afin d'éviter le dédoublement des places
if (!(tab.contains(tab1[i])))
{
tab.addElement(tab1[i]);
}
}

for (int i = 0; i < tab2.length; i++ )
{
boolean bool = false;
for (int j = 0; j < tab.size(); j++)
{
```

```
Place place_0 = (Place) tab.elementAt(j);
if (place_0.getIsPsiermAssociatedPlace() &&
tab2[i].getIsPsiermAssociatedPlace())
{
if (place_0.getPsiTermAssoc().isEqualsTo(
tab2[i].getPsiTermAssoc()))
{
place_0.setPreTransitions(createTransitionsArray(
tab2[i].getPreTransitions(),
place_0.getPreTransitions()));
place_0.setPostTransitions(createTransitionsArray(
tab2[i].getPostTransitions(),
place_0.getPostTransitions()));
}
bool = bool || (place_0.getPsiTermAssoc().isEqualsTo(
tab2[i].getPsiTermAssoc()));
}
}
if (!bool)
{
tab.addElement(tab2[i]);
}
}

Place[] array_created = new Place[tab.size()];

for (int i = 0; i < tab.size(); i++ )
{
array_created[i] = (Place) tab.elementAt(i);
}

return array_created;
}

/* Construction d'un tableau de transitions à partir
des deux tableaux */
static public Transition[] createTransitionsArray(Transition[] tab1,
Transition[] tab2) {
Vector tab = new Vector();

if (tab1 != null)
```

```
{
for (int i = 0; i < tab1.length; i++ )
{
//Une condition afin d'éviter le dédoublement des transitions
if (!(tab.contains(tab1[i])))
{
tab.addElement(tab1[i]);
}

}
}

if (tab2 != null)
{
for (int i = 0; i < tab2.length; i++ )
{
if (!(tab.contains(tab2[i])))
{
tab.addElement(tab2[i]);
}

}
}

Transition[] array_created = new Transition[tab.size()];

for (int i = 0; i < tab.size(); i++ )
{
array_created[i] = (Transition) tab.elementAt(i);
}

return array_created;
}

/* Une méthode pour ajouter une place dans une liste de places */
static Place[] addPlace(Place[] places_array, Place place_to_add) {
int size;
if (places_array != null) {
size = places_array.length;
Place[] result_array = new Place[size + 1];
for (int i = 0; i < size; i++)
```

```
{
result_array[i] = places_array[i];
}
result_array[size] = place_to_add;
return result_array;
}

else {
size = 0;
Place[] result_array = new Place[size + 1];
result_array[size] = place_to_add;
return result_array;
}
}

/* Une méthode pour ajouter une transition dans une
liste de transitions */
static Transition[] addTransition(Transition[] transitions_array,
Transition transition_to_add) {
int size;
if (transitions_array != null) {
size = transitions_array.length;
Transition[] result_array = new Transition[size + 1];
for (int i = 0; i < size; i++)
{
result_array[i] = transitions_array[i];
}
result_array[size] = transition_to_add;
return result_array;
}
else {
size = 0;
Transition[] result_array = new Transition[size + 1];
result_array[size] = transition_to_add;
return result_array;
}
}

/* Méthode pour effacer une place dans une liste */
static Place[] removePlace(Place[] places, Place place) {
Vector tmp_places = toVector(places);
```

```
if (tmp_places.contains(place))
{
int pos  = tmp_places.indexOf(place);
tmp_places.removeElementAt(pos);

Agent.place_number --;
}

Place[] tmp_array = new Place[tmp_places.size()];

for (int i = 0; i < tmp_places.size(); i++)
{
tmp_array[i] = (Place) tmp_places.elementAt(i);
}
return tmp_array;
}

public static Vector toVector(Object[] objects) {
Vector result_vector = new Vector();
if (objects != null)
{
int size = objects.length;
for (int i = 0; i < size; i++)
{
result_vector.addElement(objects[i]);
}
}
return result_vector;
}

public static Place[] toPlacesArray(Vector vector) {
Place[] result_array = null;
int size = vector.size();
if (size != 0)
{
result_array = new Place[size];
for (int i = 0; i < size; i++)
{
result_array[i] = (Place) vector.elementAt(i);
}
}
```

```
return result_array;
}

public static Transition[] toTransitionsArray(Vector vector) {
Transition[] result_array = null;
int size = vector.size();
if (size != 0)
{
result_array = new Transition[size];
for (int i = 0; i < size; i++)
{
result_array[i] = (Transition) vector.elementAt(i);
}
}
return result_array;
}

public static void arrangePlacesNumber(Place[] places) {
for (int i = 0; i < places.length; i++)
{
places[i].setPlaceNumber(i + 1);
}
}

public static void arrangeTransitionsNumber(Transition[]
transitions) {
for (int i = 0; i < transitions.length; i++)
{
transitions[i].setTransitionNumber(i + 1);
}
}

/* Pour effacer une transition dans une liste */
public Transition[] removeTransition(Transition[] transitions,
Transition transition) {
Vector tmp_transitions = toVector(transitions);
if (tmp_transitions.contains(transition))
{
Place[] tmp_pre_places = transition.getPrePlaces();
if (tmp_pre_places != null)
{
```

```
for (int i = 0; i < tmp_pre_places.length; i++)
{
Vector tmp_vector = toVector(tmp_pre_places[i].
getPostTransitions());
tmp_pre_places[i].setPostTransitions(
toTransitionsArray(tmp_vector));
}
}

Place[] tmp_post_places = transition.getPostPlaces();
if (tmp_post_places != null)
{
for (int i = 0; i < tmp_post_places.length; i++)
{
Vector tmp_vector = toVector(tmp_post_places[i].
getPreTransitions());
tmp_post_places[i].setPreTransitions(
toTransitionsArray(tmp_vector));
}
}

int pos = tmp_transitions.indexOf(transition);
tmp_transitions.removeElementAt(pos);
Agent.transition_number --;
for (int j = pos + 1; j < tmp_transitions.size(); j++)
{
Transition tmp_transition = (Transition) tmp_transitions.
elementAt(j);
tmp_transition.setTransitionNumber(tmp_transition.
getTransitionNumber() - 1);
}
}

Transition[] tmp_array = new Transition[tmp_transitions.size()];

for (int i = 0; i < tmp_transitions.size(); i++)
{
tmp_array[i] = (Transition) tmp_transitions.elementAt(i);
}
return tmp_array;
}
```

```
/* Méthode pour supprimer une liste de transitions
d'une autre liste */
public Transition[] removeTransitionsArray(Transition[] transitions1,
Transition[] transitions2) {
Transition[] tmp_transitions = transitions1;
for (int i = 0; i < transitions2.length; i++)
{
tmp_transitions = removeTransition(tmp_transitions,
 transitions2[i]);
}
return tmp_transitions;
}

/* Méthode pour supprimer le lien entre une transition et les
places qui y sont connectées */
public static void cleanAllRelationsForTransition(
Transition transition) {
Place[] tmp_pre_places = transition.getPrePlaces();
if (tmp_pre_places != null)
{
for (int i = 0; i < tmp_pre_places.length; i++)
{
Vector tmp_vector = toVector(tmp_pre_places[i].
getPostTransitions());
tmp_pre_places[i].setPostTransitions(toTransitionsArray(
tmp_vector));
}
}

Place[] tmp_post_places = transition.getPostPlaces();
if (tmp_post_places != null)
{
for (int i = 0; i < tmp_post_places.length; i++)
{
Vector tmp_vector = toVector(tmp_post_places[i].
getPreTransitions());
tmp_post_places[i].setPreTransitions(toTransitionsArray(
tmp_vector));
}
}
```

```
}

/* Méthode pour construire la liste des transitions de lancement */
public Transition[] buildLaunchingTransitionsArray() {
Vector tmp_launching_transitions = new Vector();
for (int i = 0; i < this.alltransitions.length; i++)
{
if ((this.getAllTransitions())[i].isLaunchingTransition())
{
tmp_launching_transitions.addElement(this.alltransitions[i]);
}
}

int lg = tmp_launching_transitions.size();
Transition[] tmp_array = new Transition[lg];

for (int i = 0; i < lg; i++)
{
tmp_array[i] = (Transition) tmp_launching_transitions.elementAt(i);
}
return tmp_array;
}

static Place[] makeArrayFromAPlace(Place place) {
Place[] array_for_place = new Place[1];
array_for_place[0] = place;
return array_for_place;
}

/* Une méthode pour défaire une launchingPlace d'agent */
public void makeLaunchingPlaceFalse() {
this.getLaunchingPlace().setIsLaunchingPlace(false);
}

/* Une méthode pour défaire une terminalPlace d'un agent */
public void makeTerminalPlaceFalse() {
this.getTerminalPlace().setIsTerminalPlace(false);
}

/* Une méthode pour défaire les launchingPlaces existantes */
public void makeAllLaunchingTransitionsFalse(Agent agent) {
```

```
for (int i = 0; i < agent.getAllTransitions().length; i++ )
{
agent.getAllTransitions()[i].setLaunchingTransition(false);
}
}

/* Méthode pour traiter les agents élémentaires et construire
les arbres associés */
static void processPrimitive(Agent agent) {
if (agent.getPrimitiveType().equals("tell") ||
agent.getPrimitiveType().equals("Tell") ||
agent.getPrimitiveType().equals("TELL"))
{
Agent.agent_number += 1;
Agent.tell_primitive += 1;

Place place_for_launch = new Place("tell_place_"+
Agent.tell_primitive, 0, null, null);
place_for_launch.setIsLaunchingPlace(true);
agent.launching_place = place_for_launch;
agent.allplaces = addPlace(agent.allplaces, place_for_launch);

Place place_for_terminate = new Place("tell_term_"+
Agent.tell_primitive, 0, null, null);
place_for_launch.setIsTerminalPlace(true);
agent.terminal_place = place_for_terminate;
agent.allplaces = addPlace(agent.allplaces, place_for_terminate);

if (agent.getPsiTermAssoc().isClosed())
{
Transition new_transition = new Transition(makeArrayFromAPlace
(place_for_launch), addPlace(makeArrayFromAPlace(
agent.getPsiTermAssoc().getPlaceAssoc()),
place_for_terminate));
agent.alltransitions = addTransition(agent.alltransitions,
new_transition);
new_transition.setPrePlaces(makeArrayFromAPlace(place_for_launch));
new_transition.setPostPlaces(addPlace(makeArrayFromAPlace(agent.
getPsiTermAssoc().getPlaceAssoc()), place_for_terminate));
}
else
```

```
{
/* TRAITER LE CAS OU LE PSI-TERME N'EST PAS CLOS */
}

}
else if (agent.getPrimitiveType().equals("get") ||
agent.getPrimitiveType().equals("Get") ||
agent.getPrimitiveType().equals("GET"))
{
Agent.agent_number += 1;
Agent.get_primitive += 1;

Place place_for_launch=new Place("get_place_"+Agent.get_primitive,
0, null, null);
place_for_launch.setIsLaunchingPlace(true);
agent.launching_place = place_for_launch;
Place[] tmp_places0 = agent.allplaces;
agent.setAllPlaces(addPlace(tmp_places0, place_for_launch));

Place place_for_terminate=new Place("get_term_"+Agent.get_primitive,
0, null, null);
place_for_launch.setIsTerminalPlace(true);
agent.terminal_place = place_for_terminate;
Place[] tmp_places1 = agent.allplaces;
agent.setAllPlaces(addPlace(tmp_places1, place_for_terminate));

if (agent.getPsiTermAssoc().isClosed())
{
Transition new_transition = new Transition(addPlace(
makeArrayFromAPlace(agent.getPsiTermAssoc().getPlaceAssoc()),
place_for_launch), makeArrayFromAPlace(place_for_terminate));
agent.alltransitions = addTransition(agent.alltransitions,
new_transition);
new_transition.setPrePlaces(addPlace(makeArrayFromAPlace(
agent.getPsiTermAssoc().getPlaceAssoc()), place_for_launch));
new_transition.setPostPlaces(makeArrayFromAPlace(
place_for_terminate));
}
else
{
/* TRAITER LE CAS OU LE PSI-TERME N'EST PAS CLOS */
```

```
}
}
else if (agent.getPrimitiveType().equals("ask") ||
agent.getPrimitiveType().equals("Ask") ||
agent.getPrimitiveType().equals("ASK"))
{
Agent.agent_number += 1;
Agent.ask_primitive += 1;

Place place_for_launch = new Place("ask_place_"+
Agent.ask_primitive, 0,null, null);
place_for_launch.setIsLaunchingPlace(true);
agent.launching_place = place_for_launch;
agent.allplaces = addPlace(agent.allplaces, place_for_launch);

Place place_for_terminate = new Place("ask_term_"+
Agent.ask_primitive, 0,null, null);
place_for_launch.setIsTerminalPlace(true);
agent.terminal_place = place_for_terminate;
agent.allplaces = addPlace(agent.allplaces, place_for_terminate);

if (agent.getPsiTermAssoc().isClosed())
{
/* Pour tenir compte de la transformation de l'arc contextuel
en deux arcs normaux */
Transition new_transition = new Transition(createAllPlacesArray(
makeArrayFromAPlace(place_for_launch), makeArrayFromAPlace(
agent.getPsiTermAssoc().getPlaceAssoc())), null, null,
createAllPlacesArray(makeArrayFromAPlace(place_for_terminate),
makeArrayFromAPlace(agent.getPsiTermAssoc().getPlaceAssoc())));
agent.alltransitions = addTransition(agent.alltransitions,
new_transition);
new_transition.setPrePlaces(createAllPlacesArray(
makeArrayFromAPlace(place_for_launch), makeArrayFromAPlace(
agent.getPsiTermAssoc().getPlaceAssoc())));
new_transition.setPostPlaces(createAllPlacesArray(
makeArrayFromAPlace(place_for_terminate), makeArrayFromAPlace(
agent.getPsiTermAssoc().getPlaceAssoc())));
}
else
{
```

```
/* TRAITER LE CAS OU LE PSI-TERME N'EST PAS CLOS */
}
}
else if (agent.getPrimitiveType().equals("nask") ||
agent.getPrimitiveType().equals("Nask") ||
agent.getPrimitiveType().equals("NASK"))
{
Agent.agent_number += 1;
Agent.nask_primitive += 1;

Place place_for_launch = new Place("nask_place_"+
Agent.nask_primitive, 0, null, null);
place_for_launch.setIsLaunchingPlace(true);
agent.launching_place = place_for_launch;
agent.allplaces = addPlace(agent.allplaces, place_for_launch);

Place place_for_terminate = new Place("nask_term_"
+Agent.nask_primitive, 0, null, null);
place_for_launch.setIsTerminalPlace(true);
agent.terminal_place = place_for_terminate;
agent.allplaces = addPlace(agent.allplaces, place_for_terminate);

if (agent.getPsiTermAssoc().isClosed())
{
Transition new_transition = new Transition(makeArrayFromAPlace(
place_for_launch), null, makeArrayFromAPlace(
agent.getPsiTermAssoc().getPlaceAssoc()), makeArrayFromAPlace(
place_for_terminate));
agent.alltransitions = addTransition(agent.alltransitions,
new_transition);
}
else
{
/* TRAITER LE CAS OU LE PSI-TERME N'EST PAS CLOS */
}
}
}

/* Méthode pour faire la composition séquentielle telle que définie
dans le formalisme utilisé */
static void sequentialComposition(Agent agent, Agent agent1,
```

```
Agent agent2) {
Agent.agent_number += 1;
Agent.sequential_agent += 1;

/* Toutes les places de l'agent après la composition séquentielle
de deux agents agent1 et agent2 */
agent.setAllPlaces(agent.createAllPlacesArray(agent1.getAllPlaces(),
agent2.getAllPlaces()));

/* Toutes les transitions de l'agent après la composition séquentielle
de deux agents agent1 et agent2 */
agent.setAllTransitions(agent.createTransitionsArray(
agent1.getAllTransitions(), agent2.getAllTransitions()));

/* Ajout de la transition qui relie les deux agents en une
séquence */
Place[] tmp_pre_places = new Place[1];
tmp_pre_places[0] = agent1.getTerminalPlace();
Place[] tmp_post_places = new Place[1];
tmp_post_places[0] = agent2.getLaunchingPlace();
Transition transition_to_add = new Transition(tmp_pre_places,
tmp_post_places);
agent.alltransitions = addTransition(agent.alltransitions,
transition_to_add);

transition_to_add.setPrePlaces(tmp_pre_places);
transition_to_add.setPostPlaces(tmp_post_places);

agent.setLaunchingPlace(agent1.getLaunchingPlace());
agent.setTerminalPlace(agent2.getTerminalPlace());
agent.setLaunchingTransitions(agent1.getLaunchingTransitions());
agent.setTerminalTransitions(agent2.getTerminalTransitions());
arrangePlacesNumber(agent.allplaces);
}

static void parallelComposition(Agent agent, Agent agent1,
 Agent agent2) {
Agent.agent_number += 1;
Agent.parallel_agent += 1;

/* Toutes les places de l'agent après la composition parallèle
```

```
de deux agents agent1 et agent2 */
agent.setAllPlaces(agent.createAllPlacesArray(
agent1.getAllPlaces(),  agent2.getAllPlaces()));
Place place_for_launching = new Place("par_launch_"+
Agent.parallel_agent, 0, null, null);
agent.setLaunchingPlace(place_for_launching);
Place place_for_ending = new Place("par_term_"+
Agent.parallel_agent, 0, null, null);
agent.setTerminalPlace( place_for_ending);
agent.allplaces = addPlace(agent.allplaces, place_for_launching);
agent.allplaces = addPlace(agent.allplaces, place_for_ending);

/* Toutes les transitions de l'agent après la composition
parallèle de deux agents agent1 et agent2 */
Transition[] tmp_transitions = agent.createTransitionsArray(
agent1. getAllTransitions(), agent2.getAllTransitions());
agent.setAllTransitions(tmp_transitions);

for (int i = 0; i < agent1.getLaunchingTransitions().length; i++)
{
Transition transition_to_add = new Transition(
makeArrayFromAPlace(agent.getLaunchingPlace()),
((agent1.getLaunchingTransitions())[i]).
getContextualPlaces(), ((agent1.getLaunchingTransitions())[i]).
getInhibitorPlaces(), addPlace((agent1.
getLaunchingTransitions())[i].getPostPlaces(),
agent2.getLaunchingPlace()));
agent.alltransitions = addTransition(agent.alltransitions,
transition_to_add);
transition_to_add.setPrePlaces(makeArrayFromAPlace(
agent.getLaunchingPlace()));
transition_to_add.setPostPlaces(addPlace((
agent1.getLaunchingTransitions())[i].getPostPlaces(),
agent2.getLaunchingPlace()));
}

for (int i = 0; i < agent2.getLaunchingTransitions().length; i++)
{
Transition transition_to_add = new Transition(
makeArrayFromAPlace(agent.getLaunchingPlace()),
((agent2.getLaunchingTransitions())[i]).getContextualPlaces(),
```

```
((agent2.getLaunchingTransitions())[i]).getInhibitorPlaces(),
addPlace((agent2.getLaunchingTransitions())[i].getPostPlaces(),
agent1.getLaunchingPlace()));
agent.alltransitions = addTransition(agent.alltransitions,
transition_to_add);
transition_to_add.setPrePlaces(makeArrayFromAPlace(agent.
getLaunchingPlace()));
transition_to_add.setPostPlaces(addPlace((
agent2.getLaunchingTransitions())[i].getPostPlaces(),
agent1.getLaunchingPlace()));
}

/* La place terminale du nouvel agent et les transitions qui la
relient aux anciennes places terminales */
Place[] tmp_two_terminal_places = addPlace(makeArrayFromAPlace(
agent1.getTerminalPlace()), agent2.getTerminalPlace());
Transition transition_to_add = new Transition(tmp_two_terminal_places,
makeArrayFromAPlace(agent.getTerminalPlace()));
agent.alltransitions = addTransition(agent.alltransitions,
transition_to_add);
transition_to_add.setPrePlaces(tmp_two_terminal_places);
transition_to_add.setPostPlaces(makeArrayFromAPlace(
agent.getTerminalPlace()));
arrangePlacesNumber(agent.allplaces);
}

/* Méthode pour effectuer la composition alternative et
reconstruire l'arbre de l'agent*/
static void alternativeComposition(Agent agent, Agent agent1,
Agent agent2) {
Agent.agent_number += 1;
Agent.alternative_agent += 1;

Place place_for_launching = new Place("alt_launch_"+
Agent.alternative_agent, 0, null, null);
agent.setLaunchingPlace(place_for_launching);
Place place_for_ending = new Place("alt_term_"+
Agent.alternative_agent, 0, null, null);
agent.setTerminalPlace( place_for_ending);

/* Suppression de la place de lancement du premier agent */
```

```
Place[] tmp_places1 = agent.removePlace(agent1.getAllPlaces(),
agent1.getLaunchingPlace());

/* Suppression de la place de lancement du deuxième agent */
Place[] tmp_places2 = agent.removePlace(agent2.getAllPlaces(),
agent2.getLaunchingPlace());

/* Construction de la liste de toutes les places de l'agent
obtenu par composition alternative */
agent.setAllPlaces(agent.createAllPlacesArray(tmp_places1,
tmp_places2));

/* Ajout de la nouvelle place de lancement dans la liste
de toutes les places */
agent.allplaces = addPlace(agent.allplaces, place_for_launching);

/* Ajout de la nouvelle place de terminaison dans la liste de
toutes les places */
agent.allplaces = addPlace(agent.allplaces, place_for_ending);
arrangePlacesNumber(agent.allplaces);
/* Construction de l'ensemble de toutes les transitions de
l'agent obtenu par composition alternative */
Transition[] tmp_transitions_0 = agent.createTransitionsArray(
agent1. buildLaunchingTransitionsArray(),
agent2.buildLaunchingTransitionsArray());
Transition[] tmp_transitions_1 = agent.removeTransitionsArray(
agent1.getAllTransitions(), agent1.buildLaunchingTransitionsArray());
Transition[] tmp_transitions_2 = agent.removeTransitionsArray(
agent2.getAllTransitions(), agent2.buildLaunchingTransitionsArray());
Transition[] tmp_transitions = agent.createTransitionsArray(
tmp_transitions_1, tmp_transitions_2);

agent.setAllTransitions(tmp_transitions);

/* Ajout de nouvelles transitions partant de la nouvelle place de
lancement */
for (int i = 0; i < tmp_transitions_0.length; i++)
{
Place[] tmp_preplaces_minus_launching = removePlace(
tmp_transitions_0[i].getPrePlaces(), agent1.getLaunchingPlace());
tmp_preplaces_minus_launching = removePlace(
```

```
tmp_preplaces_minus_launching, agent2.getLaunchingPlace());
Transition new_transition = new Transition(addPlace(
tmp_preplaces_minus_launching, agent.getLaunchingPlace()),
tmp_transitions_0[i].getContextualPlaces(), tmp_transitions_0[i].
getInhibitorPlaces(), tmp_transitions_0[i].getPostPlaces());

agent.alltransitions = addTransition(agent.alltransitions,
new_transition);
new_transition.setPrePlaces(addPlace(
tmp_preplaces_minus_launching, agent.getLaunchingPlace()));
new_transition.setPostPlaces(tmp_transitions_0[i].
getPostPlaces());
}

/* Ajout de la nouvelle transition finale */
Place[] tmp_two_terminal_places = addPlace(makeArrayFromAPlace(
agent1. getTerminalPlace()), agent2.getTerminalPlace());
Transition transition_to_add = new Transition(
tmp_two_terminal_places, makeArrayFromAPlace(
agent.getTerminalPlace()));
agent.alltransitions = addTransition(agent.alltransitions,
transition_to_add);
transition_to_add.setPrePlaces(tmp_two_terminal_places);
transition_to_add.setPostPlaces(makeArrayFromAPlace(agent.
getTerminalPlace()));
arrangePlacesNumber(agent.allplaces);
}

/* Je doute que ceci soit utile. Mais je le garde d'abord... */
public void addTokens(Place place, int tokens) {
place.addTokens(tokens);
}

/* Pour retrouver toutes les places liées aux psi-termes
contenues dans un agent construit à partir d'un psi-terme */
public Place[] buildAllPsiTermPlaces(PsiTerm psiterm_parent) {
Place current_psiterm_place = new Place(psiterm_parent);
psiterm_parent.setPlaceAssoc(current_psiterm_place);

int size = psiterm_parent.getValuesOfPsiTerm().length;
Vector tmp_vect = new Vector();
```

```
tmp_vect.addElement(current_psiterm_place);
for (int i = 0; i < size; i++)
{
if (psiterm_parent.getValuesOfPsiTerm()[i].getIsPsiTerm())
{
PsiTerm tmp_psi = (PsiTerm) (psiterm_parent.
getValuesOfPsiTerm()[i]).getValue();
tmp_psi.setIsChild(true);
tmp_psi.setParent(psiterm_parent);
Place[] tmp_places = buildAllPsiTermPlaces(tmp_psi);
if (tmp_places != null)
{
for (int j = 0; j < tmp_places.length; j++)
{
boolean tmp_bool = false;
for (int k = 0; k < tmp_vect.size(); k++)
{
Place tmp_place = (Place) tmp_vect.elementAt(k);
PsiTerm tmp_psi_0 = tmp_place.getPsiTermAssoc();
PsiTerm tmp_psi_1 = tmp_places[j].getPsiTermAssoc();
tmp_bool = tmp_bool || tmp_psi_0.isEqualsTo(tmp_psi_1);
}
if (!tmp_bool)
tmp_vect.addElement(tmp_places[j]);
}
}
}
}
int size1 = tmp_vect.size();

Place[] result_array = new Place[size1];
for (int i = 0; i < size1; i++)
{
result_array[i] = (Place) tmp_vect.elementAt(i);
}
return result_array ;
}

/* Méthode qui construit un fichier de réseau selon le format INA */
public static void generateFileForAgent(Agent agent) {
Place[] places = agent.getAllPlaces();
```

```
agent.getLaunchingPlace().setNumberOfTokens(1);
Transition[] transitions = agent.getAllTransitions();
Date date = new Date();
String format_date = "yyMMdd_HHmm";
SimpleDateFormat my_patterns = new SimpleDateFormat(format_date);
String part_of_name1 = my_patterns.format(date);
String file_name = "net"+ part_of_name1;

String content_file = "P   M   PRE,POST  NETZ "+0
+":                ";
file_name += ".pnt";

content_file += System.getProperty("line.separator") ;
for (int i = 0; i < places.length; i++)
{
Transition[] tmp_pre_transitions = places[i].getPreTransitions();
Transition[] tmp_post_transitions = places[i].getPostTransitions();
String tmp_string_0 = "";
String tmp_string_1 = "";

if (tmp_pre_transitions != null)
{
for (int j = 0; j < tmp_pre_transitions.length; j++)
{
if (j != tmp_pre_transitions.length - 1)
tmp_string_0 += tmp_pre_transitions[j].getTransitionNumber()+" ";
else
tmp_string_0 += tmp_pre_transitions[j].getTransitionNumber();
}
tmp_string_0 = "     "+tmp_string_0;
}
else
tmp_string_0 = "    ";

if (i < 9)
content_file += "  "+ places[i].getPlaceNumber();
else if (i < 99)
content_file += " "+ places[i].getPlaceNumber();
else
content_file += ""+ places[i].getPlaceNumber();
```

```
content_file += " "+places[i].getNumberOfTokens()+tmp_string_0;

if (tmp_post_transitions != null)
{
content_file += ",";
for (int k = 0; k < tmp_post_transitions.length; k++)
{
tmp_string_1 += " "+tmp_post_transitions[k].
getTransitionNumber();
}
}

content_file += tmp_string_1 + System.getProperty(
"line.separator");
}

content_file += "@" + System.getProperty("line.separator");
content_file += "place nr.              name capacity time";
content_file += System.getProperty("line.separator");

for (int i = 0; i < places.length; i++)
{
String name = places[i].buildName();
if (i < 9)
content_file += "       ";
else if (i < 99)
content_file += "      ";
else
content_file += "     ";

int reste = 23 - name.length();
for (int j = 0; j < reste; j++)
{
name +=" ";
}
content_file += places[i].getPlaceNumber()+": "+name+"oo    "+0+
System.getProperty("line.separator");
}

content_file += "@" + System.getProperty("line.separator");
content_file += "trans nr.              name priority time";
```

```
content_file += System.getProperty("line.separator");

for (int i = 0; i < transitions.length; i++)
{
String name = transitions[i].buildName();
if (i < 10)
content_file += "       ";
else
content_file += "      ";

int reste = 24 - name.length();
for (int j = 0; j < reste; j++)
{
name +=" ";
}
content_file += transitions[i].getTransitionNumber()+": "+name+""
+0+"    "+0+System.getProperty("line.separator");
}
content_file += "@" + System.getProperty("line.separator");

try {
Writer out = new BufferedWriter(new OutputStreamWriter(
new FileOutputStream(file_name), "UTF8"));
out.write(content_file);
out.close();
String border = "\n\n==================================
============\n= LE FICHIER "+ file_name;
border += " A ETE GENERE  =\n============================
====================";
System.out.println(border);
}
catch (UnsupportedEncodingException e) {
}
catch (IOException e) {
}
}
}
```

E.12 Le fichier Launch.java

```
package NFGTool;

/**
 * Cette classe sert à lancer l'application
 *
 * @version last update: 24/08/2005
 * @author Mayala Lusilabo M. Frumence
 */

import java_cup.runtime.Symbol;
import java.io.*;

class Launch {
static boolean do_debug_parse = false;

static public void main(String argv[]) {
try
{
parser p = new parser(new Lexer(new FileReader(argv[0])));
Object result = p.parse().value;
}
catch (Exception e)
{
e.printStackTrace();
} finally {
System.exit(0);
}
}
}
```