République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri Constantine

Faculté des Sciences de l'Ingénieur
Département d'informatique

N° Ordre: 139 / T.E / 2007
N° Série : 09/ INF / 2007

Thèse

pour obtenir le diplôme de
Doctorat d'Etat en Informatique

Titre

Définition d'un Langage de Spécification Formelle basé sur les ECATNets

Présentée par :
Zeghib Nadia (épouse Chaïb)

Soutenue le 18/12/2007, devant le Jury :

M. BOUFAIDA	Professeur à l'université Mentouri - Constantine	Président du Jury
M. BETTAZ	Professeur à l'INI – Alger	Directeur de thèse
Z. SAHNOUN	Professeur à l'université Mentouri - Constantine	Co-Rapporteur
M. AHMED NACER	Professeur à l' USTHB - Alger	Examinateur
N. BADACHE	Professeur à l' USTHB - Alger	Examinateur
M. BENMOHAMED	Professeur à l'université Mentouri – Constantine	Examinateur

Résumé

Cette thèse s'inscrit dans le cadre général de l'utilisation des méthodes formelles lors de la conception des systèmes concurrents. Elle concerne plus précisément, l'extension du pouvoir de spécification des réseaux de Petri à termes algébriques étendus ECATNets (« **E**xtended **C**oncurrent **A**lgebraic **T**erms **Nets** »).
Dans ce but, nous définissons un langage de spécification basé sur les ECATNets que nous appelons **CIRTA** (« **C**onstruction **I**ncrémentale des **R**éseaux de Petri à **T**ermes **A**lgébriques étendus »).
Le langage CIRTA permet la spécification des aspects statiques et dynamiques des systèmes de façon unifiée. Il est expressif grâce à une syntaxe simple et semi-graphique. Il favorise la définition de composants avec différents types de communication (synchrone et/ou asynchrone) à un haut niveau d'abstraction.
Le langage CIRTA offre des primitives de structuration permettant le développement progressif et incrémental des spécifications complexes.
La description du langage est illustrée par quelques exemples et une étude de cas : la spécification d'un système de contrôle de voies ferrées.

Abstract

This thesis is submitted within the framework of formal methods used during the design process of concurrent systems. More precisely, it concerns the extension of the specification power of the ECATNets (« **E**xtended **C**oncurrent **A**lgebraic **T**erms **Nets** »).
In this aim, we define a specification language based on ECATNets and called **CIRTA** (« **C**onstruction **I**ncrémentale des **R**éseaux de Petri à **T**ermes **A**lgébriques étendus »).
The CIRTA language allows one to specify static and dynamic aspects of systems in a unified way. It is expressive thanks to its simple and semi-graphical syntax. It promotes the components definition with various way of communication (synchronous and/or asynchronous) at a high level of abstraction.
The CIRTA language offers some structuring primitives allowing the progressive and incremental development of complex specifications.
The description of the language is illustrated by some examples and a case study: the specification of a railway control system.

ملخص

هذه الأطروحة تندرج ضمن الإطار العام لاستعمال الطرق الشكلية في عملية تصميم الأنظمة المتنافسة, و تهتم هذه الأطروحة بصفة خاصة بتوسيع قدرة الوصف الشكلي لشبكات بيتري ذات الحدود الجبرية الموسعة.

ولهذا الهدف نقترح لغة وصف تدعى "سيرتا" يكون أساسها شبكات بيتري ذات الحدود الجبرية.

تسمح لغة سيرتا بوصف الجوانب الثابتة و المتغيرة للأنظمة بطريقة موحدة. و تعد هذه اللغة معبرة حيث أن لها نحو بسيط يحتوي على رسوم بيانية. إضافة إلى ذلك فان هذه اللغة تسمح بوصف مركبات الأنظمة ومختلف الاتصالات فيما بينها على درجة عالية من التجريد.

و توفر لغة سيرتا عمليات تركيب تسهل التصميم التدريجي لوصف الأنظمة المعقدة. تقديم تعريف لغة سيرتا في هذه الأطروحة معزز بأمثلة متنوعة من بينها نظام مراقبة خطوط السكك الحديدية.

Remerciements

Enfin, l'heure est venue de rédiger ces fameux remerciements, but ultime de chaque doctorant puisqu'il s'agit bel et bien de l'événement qui marque la fin de la thèse.

En premier lieu, je tiens à exprimer ma profonde reconnaissance au Professeur *M. Bettaz* qui m'a accueilli au laboratoire LIRE, m'a formée à la recherche et a assuré avec beaucoup de gentillesse et d'attention l'encadrement de tous mes travaux : Ingéniorat, Magister, et enfin la présente Thèse d'état. Nombre des travaux réalisés ici sont le fruit de son enseignement, de ses conseils et de sa patience. J'ai pu à plusieurs reprises apprécier et profiter de sa haute compétence scientifique et pédagogique que tout le monde lui connaît.
Qu'il trouve dans ces lignes si courtes, la sincère expression de ma gratitude et le témoignage de ma reconnaissance.

Je tiens à remercier profondément le Professeur *M. Boufaida* qui a su, malgré ses lourdes responsabilités et son emploi du temps extrêmement chargé, m'apporter son soutien et me faire l'honneur de présider le jury de cette thèse.

Je suis particulièrement reconnaissante au Professeur *Z. Sahnoun* d'avoir accepté la tâche délicate d'être le co-rapporteur de cette thèse. Je tiens à le remercier chaleureusement pour les conseils dont il m'a fait bénéficier avec une bienveillante attention et une grande gentillesse.

C'est avec un grand plaisir que je compte le Professeur *N. Badache* de l'USTHB parmi les membres de ce jury. Qu'il trouve l'expression de ma sincère reconnaissance pour l'attention avec laquelle il a examiné mes travaux, et pour être venu de si loin pour participer à ce jury.

Je suis également tout à fait honorée de la présence à ce jury du Professeur *M. Ahmed Nacer* de l'USTHB, et je le remercie vivement pour l'intérêt qu'il a accordé à mon travail, et pour être venu aussi de si loin pour participer à ce jury.

J'adresse mes sincères remerciements à Mr. *M. Benmohamed*, Professeur à l'université de Constantine, pour m'avoir fait l'honneur d'étudier mes travaux de thèse et de participer à ce jury en qualité d'examinateur.

Merci à tous mes collègues (anciens et présents) du département d'Informatique pour leur soutien et leurs précieux encouragements.

Je n'aurai garde d'oublier de remercier tous ceux que j'ai eu le plaisir de côtoyer durant mes stages à Paris : Le Professeur *M. Bidoit* (de ENS Cachan), Le Professeur *C. Choppy* (LIPN Université Paris 13), Professeur *G.Berthelot* (Université Evry, Val d'Essonne), et le Professeur *K.Barkaoui* (Cedric, CNAM).
Je les remercie tous de l'intérêt qu'ils ont manifesté pour mes travaux, de l'accueil qu'ils m'ont réservé au sein de leurs équipes et pour la coopération agréable et fructueuse qu'on a eu ensemble.

Enfin, ce n'est pas sans émotion que j'évoque ici le soutien moral permanent des membres de ma famille. Je ne peux m'empêcher de les citer tous : *mes parents*, *mon mari*, mes frères (*Karim* et *Hichem*) et mes soeurs (*Sihem*, *Nora* et *Chérine*).
Qu'il me pardonne de ne pas savoir traduire par les mots appropriés l'ampleur de mes sentiments à leur égard ; et qu'ils sachent que c'est avec un grand plaisir que je leur dédie à tous, cette thèse.

Table des Matières

Chapitre 4: Modules de spécification dans le langage CIRTA

Chapitre 5: Primitives du langage

Chapitre 6: Mécanisme de paramétrisation

Syntaxe formelle

Chapitre 7: Syntaxe du langage CIRTA

Sémantique formelle

Chapitre 8: Sémantique formelle des ECATNets

Chapitre 9: Sémantique formelle du langage CIRTA

Application du langage

Chapitre 10 : Etude de cas « Système de contrôle de voies ferrées »

Introduction

Introduction

"Le monde réel n'est ni plat ni séquentiel, mais au contraire multidimensionnel et hautement parallèle".
Grady Booch.

• Contexte général

La part du logiciel ne cesse de croître dans les systèmes de télécommunications, de transports terrestres et aériens, de production et de transport d'énergie, de paiement électronique,...etc. Sa complexité augmente en proportion des nouvelles fonctions dévolues au logiciel.

La sécurité des hommes et des biens d'une part et la maîtrise des coûts de maintenance d'autre part conduisent à accorder une attention plus grande aux méthodologies de développement et de vérification des logiciels.

L'acuité du problème est évidente lorsque l'on considère des systèmes qualifiés de critiques, c'est-à-dire dont les défaillances peuvent avoir des conséquences dramatiques sur les hommes qui sont dans leur environnement. A titre d'exemple citons les systèmes de contrôle de commandes de procédés industriels, les systèmes de régulation de trafic ou d'assistance au pilotage de véhicules.

Le problème concerne également des applications moins critiques comme la sécurité dans les transactions monétaires. De manière plus générale, la fiabilité des logiciels concerne également tous les logiciels embarqués non critiques dupliqués à des millions d'exemplaires et qui, de ce fait, engendrent des coûts de mise à jour prohibitifs.

Pour toutes ces raisons, le problème de développement et de validation des logiciels est aujourd'hui considéré comme un enjeu crucial sur le plan économique.

Les méthodes formelles constituent un élément de réponse à ces enjeux économiques, industriels et scientifiques. En effet, l'utilisation des méthodes formelles dans le processus de développement, améliore la compréhension des besoins, clarifie l'expression de ces besoins en levant les zones d'ombre, les contradictions et les ambiguïtés dans la spécification. Elles permettent également la vérification et la validation de la spécification et de l'implémentation correspondantes, et facilitent le passage de la spécification/conception à l'implémentation.

La première règle à suivre pour développer un logiciel de qualité est d'adopter un «Cycle de Vie ». Celui-ci détermine un ensemble de phases situées dans le temps, débutant dés la définition des besoins et s'arrêtant à la disparition du logiciel. Un cycle de vie classique [MCD-91] possède, entre autre, les phases de définition des besoins, de spécification, de conception et d'implémentation.

La phase de spécification joue un rôle important. Son but premier est d'énoncer clairement l'ensemble des fonctionnalités attendues du logiciel indépendamment des choix de mise en œuvre. Elle fournit un document de référence qui permet de prouver la correction du logiciel, de procéder à l'élaboration des prototypes de celui-ci, de guider sa réalisation, de contrôler sa maintenance et de décider s'il est réutilisable éventuellement à d'autres fins.

Il n'est pas donc étonnant que les méthodes et outils de spécification occupent aujourd'hui une place centrale dans les recherches sur le génie logiciel où les systèmes deviennent de plus en plus complexes et nécessitent la prise en considération de certains facteurs tels que la diversité des structures de données, le parallélisme, la distribution et le temps réel. Ceci a conduit à l'émergence et au développement de nombreuses approches de spécification formelles [BOW-07]. Ces formalismes permettent de prendre en compte l'un et/ou l'autre des deux aspects fondamentaux des systèmes: L'aspect relatif aux structures de données manipulées et l'aspect structure de contrôle correspondant au comportement dynamique (parallélisme, concurrence, temps réel) des systèmes.

Dans ce contexte, le formalisme des ECATNets ("*Extended Concurrent Algebraic Terms Nets*") développés au laboratoire L.I.RE de l'Université de Constantine par M.Bettaz [BMSB-93][BM-95] dans le cadre des travaux de l'équipe « Génie Logiciel et Systèmes distribués », compte parmi les formalismes de spécification les plus intéressants. Il combine à la fois la puissance des spécifications algébriques [GT-79][EFHLJ-88] et celle des réseaux de Petri [PET-79][GV-03]. L'utilisation des spécifications algébriques permet la description des structures de données du système, et les réseaux de Petri sont employés afin d'exprimer la concurrence, l'indéterminisme et d'autres propriétés relatives à l'aspect contrôle du système.

• Problématique

La spécification d'un système complexe peut rarement être décrite d'un seul tenant et la citation de G.Booch mise en exergue de cette introduction pose clairement le problème que rencontre un concepteur qui doit spécifier un système réel : les entités qu'on peut identifier sont souvent des entités actives (i.e. qui peuvent provoquer un changement de leur propre état), et dont le comportement n'est pas séquentiel.

Ces considérations nous amènent à dégager quelques qualités que doit posséder un formalisme de spécification des systèmes concurrents :

1. *Représentation satisfaisante du parallélisme*. On doit pouvoir caractériser et différencier la simultanéité, la dépendance ou l'indépendance des activités d'un système. Ce point est crucial lorsqu'il s'agit de représenter le monde réel dans lequel des évènements peuvent avoir n'importe laquelle de ces caractéristiques.

2. *Formalisme*. La représentation d'un système doit pouvoir être analysée et donc posséder les qualités formelles permettant de procéder à des preuves. On ne saurait se contenter de modèles informels dans lesquels un grand nombre des propriétés ne peut être démontré avec certitude.

3. *Représentation des communications*. Il faut être capable de mettre en valeur les communications entre les différentes parties d'un système complexe ; c'est particulièrement important dans le cas de systèmes répartis pour lesquels la communication entre les composants joue un rôle primordial.

4. *Modularité.* Pour traiter des systèmes complexes, il faut avoir la possibilité de les décomposer en sous-systèmes plus simples à modéliser, puis pouvoir recomposer les « sous-modélisations » ainsi obtenues en un tout cohérent qui forme la modélisation du système de départ.

5. *Représentation du temps*. Il faut pouvoir faire référence au temps (à des durées ou à des dates) pour représenter sa progression ou les traitements dont le comportement dépend de l'écoulement du temps.

Si nous confrontons les ECATNets aux critères énumérés plus haut, nous pouvons noter les points suivants :

- Les ECATNets présentent l'avantage de vérifier les deux premiers critères :
  - Ils permettent de modéliser le vrai parallélisme qui ne se réduit pas à un entrelacement des évènements.
  - Ils ont une sémantique formelle définie dans la logique de réécriture permettant de procéder à des preuves dans un cadre mathématique.

- Les ECATNets ne permettent pas de spécifier les contraintes temporelles des systèmes concurrents ; et manquent de concepts qui permettent d'une part, la représentation des communications, et d'autre part la définition des modules de spécification.

Pour répondre à cette problématique, et afin de mieux contrôler la phase de spécification dans le formalisme des ECATNets, il est nécessaire de disposer d'un langage de spécification muni de primitives de structuration qui offrent la possibilité de donner une spécification à l'aide d'un ensemble organisé de modules permettant l'élaboration progressive des spécifications des systèmes de taille conséquente.

• Objectifs

Cette thèse s'inscrit dans le cadre général de l'utilisation des spécifications formelles lors de la phase conceptuelle des systèmes concurrents, et concerne plus précisément l'extension des pouvoirs d'expression du modèle des ECATNets

L'objectif principal de cette thèse consiste à élaborer un langage nommé ***CIRTA*** (« ***C****onstruction* ***I****ncrémentale des* ***R****éseaux à* ***T****ermes* ***A****lgébriques* ») muni d'un ensemble de primitives expressivement conçues pour la commodité de description et de manipulation des différents composants d'un ECATNet.

L'objectif de ce langage est triple:

- Permettre la spécification des aspects statiques d'un système (structures de données, propriétés des opérations sur ces données).
- Faciliter la description adéquate des aspects dynamiques relatifs à la structure de contrôle d'un système (parallélisme, concurrence, indéterminisme, contraintes temporelles).
- Favoriser le développement modulaire des spécifications afin de pouvoir créer des spécifications à partir des modules de spécification préalablement définis.

Durant la définition du langage CIRTA deux aspects importants doivent être pris en considération :

- Le langage doit être assez expressif pour favoriser son utilisation potentielle.
- Le langage doit être formel pour permettre de procéder à des preuves.

• Synopsis de la thèse

Cette thèse est organisée selon le plan suivant:

Chapitre-1: Formalisme des ECATNets
Le premier chapitre de cette thèse constitue une introduction détaillée au formalisme des ECATNets [BMSB-93] et met en relief les aspects caractéristiques de ce formalisme.

Chapitre-2: Les ECATNets temporels
Le chapitre-2 est consacré à la présentation des *ECATNets temporels* permettant de prendre en compte l'ensemble des caractéristiques propres aux systèmes dynamiques à contraintes de temps. Ils généralisent les ECATNets en permettant la prise en compte du facteur temps comme représentant une composante explicite du système spécifié. Notre propos étant de fournir pour le langage CIRTA un formalisme de base qui généralise les ECATNets pour permettre la spécification d'une plus large gamme de systèmes.

Chapitre-3: Caractéristiques principales d'un langage de spécification
Le chapitre-3 décrit les critères de base pris en compte lors de la définition des langages de spécification et précise les principaux choix faits pour le langage CIRTA.

Chapitre-4: Modules de spécification dans le langage CIRTA
Le chapitre-4 présente l'aspect modularité dans le langage CIRTA. Nous définissons la notion de modules d'ECATNets qui permet le développement hiérarchique des spécifications. Nous précisons les différents types de modules dans le langage CIRTA et les types de communications possibles entre les modules d'une même spécification.

Chapitre-5: Primitives du langage CIRTA
Le chapitre-5 décrit les principales constructions du langage CIRTA. Il présente, en particulier, la primitive d'enrichissement ("*use*"), l'opération de composition des modules ("*combine*"), le principe de renommage ("*rename*"), et explique le mécanisme de contrôle de visibilité ("*hide/export*").

Chapitre-6: Mécanisme de paramétrisation
Le chapitre-6 introduit le concept de *paramétrisation* des ECATNets: la paramétrisation statique et la paramétrisation dynamique. Pour chacun de ces deux types de paramétrisation, nous définissons la notion de paramètre formel et la notion de paramètre effectif. Nous expliquons les différents types d'instanciations: partielle/totale et effective/ formelle.

Chapitre-7: Syntaxe du langage
Le chapitre-7 détaille la syntaxe concrète du langage CIRTA.

Chapitre-8: Sémantique formelle des ECATNets
Le chapitre-8 décrit la sémantique formelle des ECATNets (étendus avec de nouveaux concepts tels que la *capacité*, les *contraintes contextuelles* et les *contraintes temporelles)* à travers la logique de réécriture et définit la classe des modèles associée à chaque type d'ECATNets.

Chapitre-9: Sémantique formelle du langage CIRTA
Le chapitre-9 présente la sémantique du langage CIRTA. Il précise en particulier, la sémantique des différents types de modules du langage CIRTA et la sémantique formelle des différentes primitives.

Chapitre-10: Etude de cas
Afin d'illustrer les commodités de CIRTA à la spécification de systèmes informatiques de taille relativement importante, une étude de cas plus ou moins classique mais représentative de différents cas de modélisations, est développée dans le chapitre-10.

Enfin, nous concluons cette thèse en dressant un bilan qui précise les principaux résultats obtenus et les extensions possibles de notre travail.

Chapitre 1

Formalisme des ECATNets

Sommaire

VI-Comportement d'un ECATNet

1- Contexte d'une transition
   1.1 Définition
   1.2 Affectation de valeurs au contexte d'une transition
   1.3 Consistance d'une affectation

2- Franchissabilité ("enabling") d'une transition
   2-1 Franchissabilité simple d'une transition
   2-2 Franchissabilité concurrente
   2-3 Franchissabilité auto-concurrente
   2-4 Franchissabilité mixte

3- Franchissement ("firing") d'une transition
   3.1 Franchissement simple d'une transition
   3.2 Franchissemment concurrent
   3.3 Franchissement auto-concurrent
   3.4 Franchissement mixte
   3.5 Exemple

4- Marquages accessibles
   4.1 Ensemble des marquages accessibles
   4.2 Graphe des marquages accessibles
   4.3 Exemple

VI- Conclusion

I- Introduction

Parmi les méthodes de spécification formelle, deux d'entre elles, sont des plus utilisées : les *réseaux de Petri* [PET-79] [GV-03] [JEN-92] et les *spécifications algébriques* [GT-79][EF-83] [EFHLJ-88][ZEG-92].

Les réseaux de Petri expriment bien les problèmes liés à l'aspect opérationnel du contrôle des systèmes, du fait de leur simplicité et de leur aspect graphique. Le parallélisme, en particulier, comme les notions de synchronisation ou de simultanéité sont clairement exprimées. Cependant, à cause même de leur simplicité, ces réseaux sont difficilement lisibles lorsque les systèmes deviennent complexes. Cette complexité de modéliser un système réel avec les réseaux de Petri n'est pas seulement dûe au nombre de composants du système mais également à la difficulté de décrire les structures de données manipulées.

Les spécifications algébriques, quant à elles, permettent de décrire de façon formelle les données sur lesquelles opèrent les systèmes. Elles offrent de plus l'avantage de pouvoir construire des spécifications de façon modulaire et hiérarchique. On s'intéresse ici aux changements d'états des données et non à la structure de contrôle du système lui-même. On se place ainsi d'un point de vue plutôt statique que dynamique.

Ces deux méthodes de spécification procèdent donc de points de vue complémentaires. Il était donc naturel de tenter de les combiner afin de pouvoir décrire, de façon cohérente et unifiée, sans changer de modèle de spécification, à la fois les aspects statiques et les aspects dynamiques des systèmes. L'idée de combiner des formalismes de spécification pour obtenir des formalismes hybrides n'est pas nouvelle. On peut citer dans ce contexte LOTOS [BB-87] [ISO-89] qui combine le langage de spécification ACT-ONE et CCS, les réseaux OBJSA [BDM-88] [AEM-95] qui combinent les automates superposés (SA) et le langage OBJ2.

Le modèle des réseaux de Petri à termes algébriques [BET-90] combine les spécifications algébriques et les réseaux de Petri. Il offre la possibilité de définir d'abord la spécification des données selon les méthodes employées dans les types abstraits algébriques. Ensuite, une modélisation de la structure de contrôle du système peut être faite à l'aide d'un réseau inspiré des réseaux de Petri mais où les jetons circulant sur le réseau possèdent un type pour lequel une spécification algébrique existe. Ce modèle présente ainsi une certaine souplesse d'utilisation, selon que l'on veut privilégier l'aspect statique ou l'aspect dynamique du système.

Les réseaux de Petri à termes algébriques sont considérés comme une catégorie de réseaux de Petri de haut niveau [REI-91][EHH-89] généralisant les travaux de [BWW-88] [WIL-87] et ayant pour objectif la modélisation, la validation et la simulation des systèmes parallèles.

Depuis leur apparition dans [BET-90], les réseaux à termes algébriques ont connu une évolution des ATNets en CATNets [BM-91] puis en ECATNets [BMSB-93] [BM-95].

Les ATNets (***A***lgebraic ***T***erm ***Nets***) constituent la forme la plus simple de ces réseaux. Ils sont inspirés des réseaux de Petri numériques [BWW-88] où certains aspects y ont été ajoutés pour améliorer le niveau d'abstraction. En effet, les jetons sont généralisés à des termes algébriques définis sur une signature donnée au lieu qu'ils soient des tuples d'entiers. De tels termes seront considérés comme des représentations abstraites de contrôle du système et des structures de

données utilisées par ce système. Ceci permet de construire un modèle raffiné du système sans considérer les détails d'implémentation. Ainsi les ATNs contribuent à augmenter le pouvoir d'abstraction des réseaux de Petri avec l'introduction des jetons différenciés.

Les CATNets (***C***oncurrent ***A***lgebraic ***T***erm ***Nets***)[BM-91] étendent les ATNets pour les rendre des objets concurrents (les transitions sont franchies de façon concurrentes d'où un maximum de parallélisme). La différence principale entre ATNets et CATNets est que les premiers montraient plutôt l'aspect non déterministe du système et non l'aspect concurrent.

Les ECATNets (***E***xtended ***C***oncurrent ***A***lgebraic ***T***erm ***Nets***) [BMSB-93] [BM-95] ont été définis pour améliorer la puissance d'expression des CATNets en offrant, par exemple, la possibilité de fixer la capacité des places et d'associer aux transitions des conditions de franchissement.

Dans ce chapitre, on introduit les bases théoriques du formalisme des ECATNets fondé sur les réseaux de Petri et les spécifications algébriques. La présentation des ECATNets qui est donnée ici repose sur les travaux de [BET-90][BM-91][BMSB-93].

On commence par rappeler quelques notions sur les spécifications algébriques. On introduit ensuite la notion de *multi-ensemble* utilisée dans les réseaux de Petri. On montre comment étendre une spécification algébrique pour définir une telle structure de donnée. On présente ensuite les ECATNets ainsi que leur comportement dynamique.

II - Rappel sur les spécifications algébriques

1- Signature

Une signature $\Sigma = (S, Op)$ consiste en la donnée de :

- un ensemble S de noms de sortes : $s_1, s_2, ..., s_m$.
- un ensemble Op d'opérations f : $s_1\ s_2\ ...\ s_n \rightarrow s_{n+1}$.

Pour toute opération f, $(s_1, s_2, ..., s_n, s_{n+1})$ est appelé le profil de l'opération f ; $s_1, s_2, ..., s_n$ son domaine; s_{n+1} son co-domaine et n son arité. Les constantes sont des opérations d'arité nulle.

Exemple: La Figure 1.1 présente la signature des booléens avec les opérations usuelles *true, false, not, and* et *or*.

sort bool
ops true :→ bool
false :→bool
not (_): bool → bool
and : bool bool → bool
_or_ : bool bool → bool

Figure 1.1 : Signature BOOL-Sig des booléens.

2- ∑-algèbre

Soit une signature $\Sigma = (S,\ Op)$, une ∑-algèbre A est une paire $A = (S_A,\ Op_A)$ où :

(i) $S_A = (A_s)_{s \in S}$ ensembles de l'univers U.

(ii) $OP_A = (f_A)_{f \in Op}$ est l'ensemble de fonctions f_A telles que pour toute opération $f : s_1,...,s_n \rightarrow s_{n+1}$ de Op, f_A est une fonction telle que $f_A : A_{s_1}.....A_{s_n} \rightarrow A_{s_{n+1}}$.

Remarques:

- Une ∑-algèbre correspond à une interpretation de la signature Σ.
- A_s est le support de s dans A.
- f_A est l'interprétation de f dans A.
- L'interprétation d'une opération constante est une fonction constante qui détermine un élément de A.
- On note *Alg(∑)* la classe de toutes les ∑-algèbres.

3- ∑-morphisme

Soit $\Sigma = (S,\ Op)$ une signature et soient A et A' deux ∑-algèbres, un ∑-morphisme μ de A dans A' est un ensemble d'applications $(\mu_s)_{s \in S}$ telles que :

(i) $\forall\ s \in S \quad \mu_s : A_s \rightarrow A_s'$

(ii) $(\forall\ s_1,...,\ s_n,\ s_{n+1} \in S)\ (\forall\ f : s_1...s_n \rightarrow s_{n+1} \in Op)\ (\forall\ a_1 \in A_{s_1})...(\ \forall a_n \in A_{s_n})$

$$\mu_{s_{n+1}}(f_A(a_1,...,a_n)) = f_{A'}(\mu_{s_1}(a_1),...,\ \mu_{s_n}(a_n))$$

Remarques:

- Un ∑-isomorphisme est un ∑-morphisme bijectif.
- S'il existe un ∑-isomorphisme entre deux ∑-algèbres alors elles sont isomorphes.

La notion de *signature* ∑ présentée précédemment nous permet de définir les éléments de base qui vont être utilisés pour obtenir l'ensemble des expressions syntaxiques qui nous intéressent et que l'on appelle ∑-termes.

4- ∑-termes clos

Soit $\Sigma = (S,\ Op)$ une signature, l'ensemble des termes clos sur ∑ noté T_Σ est défini par $T_\Sigma = (T_{\Sigma s})_{s \in S}$ tel que :

- $(\forall s \in S)\ (\forall\ c :\rightarrow s \in Op)\ \ c \in T_{\Sigma s}$;
- $(\forall\ s_1,...,\ s_n,\ s_{n+1} \in S)\ (\forall\ f : s_1...s_n \rightarrow s_{n+1} \in Op)\ (\forall\ t_1 \in T_{\Sigma s_1})...(\forall\ t_n \in T_{\Sigma s_n})$

$$f(t_1,...,t_n) \in T_{\Sigma s_{n+1}}\ .$$

Exemple:

Si on considère la signature BOOL-Sig présentée dans la Figure1.1, les termes *true, false, not(true), true and false,* et *false or false* sont des termes clos de $T_{BOOL-Sig}$.

On note $T_{\Sigma s}$ la Σ-algèbre des Σ-termes clos de sorte *s*.

T_{Σ} la Σ-algèbre des Σ-termes clos.

5- Σ-termes avec variables

Soient $\Sigma = (S,\ Op)$ une signature et $V = (V_s)_{s \in S}$ un ensemble de variables *S*-sorté $(V \cap Op = \varnothing)$, l'ensemble des Σ-termes avec variables est noté $T_{\Sigma}(V)$ et est défini comme l'ensemble des termes clos sur la signature $\Sigma(V) = (S,\ Op \cup \{x\ :\rightarrow s\ / s \in S \wedge\ x \in V_s\})$.

On note:

- $T_{\Sigma}(V)$ la Σ-Algèbre des Σ-termes avec variables dans l'ensemble *S*-sorté *V*.
- $T_{\Sigma s}(V)$ la Σ-Algèbre des Σ-termes de sorte *s* avec variables dans l'ensemble *S*-sorté *V*.

Remarque: $T_{\Sigma s}(\varnothing) = T_{\Sigma s}$ *et* $T_{\Sigma}(\varnothing) = T_{\Sigma}$

6- Σ-équation

Soient $\Sigma = (S, Op)$ une signature et *V* un ensemble de variables, une Σ-équation est une paire $(t,\ t')$ avec *t* et *t'* deux éléments de $T_{\Sigma}(V)$ de même sorte.

Notation: Une Σ-équation $(t,\ t')$ est notée $t = t'$.

Etant donné un ensemble *E* de Σ-équations, $T_{\Sigma,E}(V)$ est la Σ-Algèbre des classes d'équivalence des Σ-termes avec variables dans l'ensemble S-sorté *V* modulo l'ensemble des équations *E*.

7- Spécification algébrique

Une spécification algèbrique est le couple $Spec = (\Sigma, E)$ où :

Σ: est une signature;

E: est un ensemble de Σ-équations.

Exemple :

Sort bool
Ops true :→ bool
false :→bool
not (_): bool → bool
and : bool bool → bool
_or_ : bool bool → bool
Var x : bool
Eqs (not(true) = false)
(not(false) = true)
(true and x = x)
(false and x = false)
(true or x = true)
(false or x = x)

Figure 1.2 : Spécification algèbrique BOOL des booléens.

Ainsi, une spécification algébrique est composée d'une signature et d'un ensemble d'équations. La signature contient les symboles utilisés pour former des expressions, et les équations (dits aussi *axiomes*) définissent certaines propriétés de ces expressions. La logique sous-jacente décrit l'ensemble des *théorèmes* que l'on peut déduire des axiomes. De plus, *la sémantique* des spécifications algébriques associe à toute spécification une classe de *modèles*, c'est à dire une classe d'algèbres multi-sortes qui satisfont les axiomes.

8- Valuation

Soient une signature $\Sigma = (S,\ Op)$, un ensemble de variables $V = (V_s)_{s\in S}$ et une $\sum$-algèbre *A*.
Une valuation de *V* dans *A* est un ensemble d'applications $v = (v_s)_{s\in S}$ telles que:

$(\forall s \in S)\ v_s : V_s \to A_s$

Une valuation consiste donc à affecter à chaque variable de sorte *s* une valeur de même sorte.

Toute valuation $v : V \to A$ est étendue de façon unique au Σ-homomorphisme $[\ ]_v : T_\Sigma(V) \to A$ appelé l'interprétation associée à *v*.

9- Interprétation des termes

Soient $\Sigma = (S,\ Op)$ une signature, $V = (V_s)_{s\in S}$ un ensemble de variables, $A = (S_A, Op_A)$ une $\sum$-algèbre et *v* une valuation de *V* dans *A*. L'interpretation d'un terme est définie par l'application :

$$[\ ]_v : T_\Sigma(V) \to A$$

$[\ ]_v$ est définie inductivement sur la structure des termes par :

- $\forall s \in S\ \ \forall x \in V_s\ \ [x]_v = v(x)$
- $\forall s \in S,\ \forall (c :\to s) \in Op\ \ [c]_v = c_A$
- $\forall s_1,\ s_2,.. s_{n+1} \in S,\ \ \forall (f : s_1\ s_2..s_n \to s_{n+1}) \in Op,\ \forall t_1 \in T_{\Sigma s_1}(V),...,\ \forall t_n \in T_{\Sigma s_n}(V)$

$$[f(t_1,...,t_n)]_v = f_A([t_1]_v,...,[t_n]_v)$$

III-Extension de la spécification algébrique pour décrire les multi-ensembles

Le concept de *terme algébrique* présenté dans les paragraphes précédents est un concept clef dans les ECATNets (Extended Concurent *Algebaic Term* Nets). Les termes permettent de décrire des entités, des conditions, des états, … etc.
Dans un système concurrent, chacune des variables d'état (correspondant dans un ECATNet à une place du réseau) peut être vue abstraitement comme une structure de donnée ayant des propriétés. Les opérations que l'on effectue sont *l'ajout* et le *retrait* de termes; et la valeur d'une variable d'état est alors définie par un multi-ensemble de termes.
Intuitivement, un multi-ensemble est un « ensemble » d'éléments (appartenant à un ensemble *S*) où chaque élément peut apparaître plusieurs fois. Il est défini par le nombre d'exemplaires de chaque élément de *S* selon la définition ci-dessous.

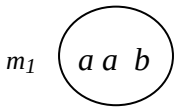
1- Notion de multi-ensemble

Un multi-ensemble *m*, sur un ensemble non vide *S* est la fonction:

$$m : S \to \mathbb{N}$$

telle que $m(s)$ est le nombre d'exemplaires de l'élément *s* dans le multi-ensemble.

Exemple: Le multi-ensemble m_1 ci-dessous contient 2 fois le terme *a* et une fois le terme *b*
(i.e. $m_1(a) = 2$, $m_1(b) = 1$)



La notion de *multi-ensemble* permet donc de décrire avec plus de précision que la notion d'ensemble, la valeur d'une variable d'état. Il est évident que les différentes opérations sur les ensembles (telles que l'union, l'intersection,...etc.) peuvent être généralisées aux multi-ensembles. Nous allons essayer de montrer comment étendre la spécification algébrique d'un ECATNet pour prendre en compte la notion de *multi-ensemble*.

Les éléments nécessaires à la description de la structure de données multi-ensemble s'obtiennent en enrichissant la spécification algébrique de façon à définir une sorte m_s pour la structure de donnée multi-ensemble de sorte *s*. Nous donnons ci-dessous une approche de cet enrichissement pour une sorte *s* quelconque.

2- Spécification algébrique des multi-ensembles

Spec MULTI-ENSEMBLE$_s$ **is**

Using BOOL + NAT + Spec-s *(Spec-s est la spécification de s)*
Sort m_s
Ops

$\emptyset_s$	: $\rightarrow m_s$	(*multi-ensemble vide*)	
$\_{}_s$	: s $\rightarrow m_s$	(*multi-ensemble à un élément*)	
$\_\oplus_s\_$	: m_s m_s $\rightarrow m_s$	(*union de multi-ensembles*)	[Assoc, comm, id= $\emptyset_s$]
$\_*_s\_$	: nat m_s $\rightarrow m_s$	(*multiplication scalaire*)	
$\lvert\_\rvert_s$	: m_s $\rightarrow$ nat	(*cardinalité de multi-ensembles*)	
$\_ ==_s \_$	: m_s m_s $\rightarrow$ bool	(*égalité de multi-ensembles*)	[comm]
$\_\in_s\_$	: s m_s $\rightarrow$ bool	(*appartenance d'un élément*)	
$\_\subseteq_s\_$	: m_s m_s $\rightarrow$ bool	(*inclusion de multi-ensembles*)	
$\_-_s\_$	: m_s m_s $\rightarrow$ m_s	(*différence de multi-ensembles*)	
$\_\cap_s\_$	: m_s m_s $\rightarrow$ m_s	(*intersection de multi-ensembles*)	[Comm]

Vars x, y : s m_1, m_2, m_3 : m_s n : nat
Eqs

(*équations pour* $\_*_s\_$)
$0 *_s m_1 = \emptyset_s$
$succ(n) *_s m_1 = m_1 \oplus_s (n *_s m_1)$

(*équations pour* $\lvert\_\rvert_s$)
$\lvert\emptyset_s\rvert_s = 0$
$\lvert x_s\rvert_s = succ(0)$
$\lvert m_1 \oplus_s m_2\rvert_s = \lvert m_1\rvert_s + \lvert m_2\rvert_s$

(*équations pour* $\_ ==_s \_$)
$m_1 ==_s m_2 = (m_1 \subseteq_s m_2)$ and $(m_2 \subseteq_s m_1)$

(*équations pour* $\_\in_s\_$)
$x \in_s \emptyset_s$ = false
$x \in_s y_s = (x = y)$
$x \in_s (y_s \oplus_s m_1) = (x = y)$ or $(x \in_s m_1)$

(*équations pour* $\_\subseteq_s\_$)
$\emptyset_s \subseteq_s m_1$ = true
$m_1 \subseteq_s \emptyset_s = m_1 = \emptyset_s$
$x_s \subseteq_s m_1 = x \in_s m_1$
$(x_s \oplus_s m_1) \subseteq_s m_2 = (x \in_s m_2)$ and $(m_1 \subseteq_s m_2 -_s x_s)$

(*équations pour* $\_-_s\_$)
$m_1 -_s \emptyset_s = m_1$
$m_1 -_s m_2 = m_1$ if $m_1 \cap_s m_2 =_s \emptyset_s$
$m_1 -_s m_2 = \emptyset_s$ if $m_1 =_s m_2$
$m_1 -_s (x_s \oplus_s m_2) = (m_1 -_s x_s) -_s m_2$
$(x_s \oplus_s m_1) -_s x_s = m_1$
$(y_s \oplus_s m_1) -_s x_s = y_s \oplus_s (m_1 -_s x_s)$ if not (x = y)

(*équations pour* $\_\cap_s\_$)
$\emptyset_s \cap_s m_1 = \emptyset_s$
$(x_s \oplus_s m_1) \cap_s m_2 = x_s \oplus_s (m_1 \cap_s (m_2 -_s x_s))$ if $x \in_s m_2$
$(x_s \oplus_s m_1) \cap_s m_2 = m_1 \cap_s m_2$ if not $(x \in_s m_2)$

end.

Dans le reste de la thèse et quand aucune ambiguité n'a lieu, le symbole *s* dans la notation des différentes opérations $\varnothing_s$ $\_{}_s\_$, $\_\oplus_s\_$, $\_*_s\_$, $\_==_s\_$, $|\_|_s$, $\_\in_s\_$, $\_\subseteq_s\_$, $\_-_s\_$, $\_\cap_s\_$ est omis pour alléger l'écriture des termes.

Notation:

- $mT_{\Sigma}(V)$ est la Σ-Algèbre des multi-ensembles de Σ-termes avec variables dans l'ensemble *S*-sorté *V*.
- $mT_{\Sigma s}(V)$ est la Σ-Algèbre des multi-ensembles de Σ-termes de sorte *s* avec variables dans l'ensemble *S*-sorté *V*.

L'extension des spécifications algébriques aux multi-ensembles permet de considérer la structure d'accueil des ressources du réseau comme une structure spécifiée algébriquement en vue d'évoluer vers un formalisme pour lequel les instances des variables d'état sont spécifiées algébriquement et l'accés à ces variables est défini par une axiomatique.

3- Principe d'extension d'une spécification pour décrire les multi-ensembles

La spécification *MULTI-ENSEMBLE-s* suppose la spécification algébrique préalable de *BOOL*, *NAT* et *Spec-s* décrivant respectivement la spécification des booléens avec les opérations classiques *true, false ,not, and* et *or* ; la spécification des entiers naturels avec les opérations *0, succ,* + , - et * ; et la spécification de la sorte *s*.
Nous avons montré comment étendre une spécification algébrique (décrivant une sorte *s*) pour définir la sorte m_S multi-ensemble de *s*. Ainsi, on peut déduire facilement la démarche à suivre dans le cas où l'ECATNet possède plusieurs variables d'état de sortes différentes.
On donne la définition suivante pour la construction de la spécification étendue permettant la description des multi-ensembles pour un ensemble donné de sortes.

Soient $\Sigma = (S, Op)$ une signature, $SPEC = (\Sigma, E)$ une spécification algébrique, et S' un ensemble de sortes tel que $S' \subseteq S$. L'extension de *SPEC* aux multi-ensembles de sortes *S'* notée $\langle SPEC \rangle_{S'}$ est définie par:

$$\langle SPEC \rangle_{S'} = SPEC + BOOL + NAT + (+_{s \in S'} MULTI\text{-}ENSEMBLE_s)$$

où + : désigne l'union de spécifications

$MULTI - ENSEMBLE_s$: la spécification qui définit la sorte multi-ensemble de sorte s ($s \in S'$) et les opérations et équations correspondantes. Cette spécification est réécrite pour chaque sorte $s \in S'$.

Les concepts nécessaires à la définition des réseaux à termes algébriques ayant été précisés, le paragraphe suivant présente la définition formelle d'un ECATNet.

IV- Définition d'un ECATNet

Un ECATNet est une paire $\mathcal{E} = (Spec, R)$ où :

$Spec = (\Sigma, E)$ est une spécification algébrique telle que :
$\Sigma = (S, Op)$ est une signature
S: est l'ensemble de sortes
Op: est l'ensemble d'opérations sur *S*
E: est l'ensemble de Σ-équations

R est un uplet tel que $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$
P: ensemble fini de places
T: ensemble fini de transitions $(P \cap T = \varnothing)$
σ: application typage des places
Cap: fonction capacité des places
IC : fonction condition d'entrée (*Input Condition*)
DT : fonction jetons détruits (*Destroyed Tokens*)
CT : fonction jetons créés (*Created Tokens*)
TC : fonction condition de transition (*Transition Condition*)

La spécification algébrique *Spec* définit les structures de données manipulées par le système. La structure de *R* décrit le comportement du système. L'ensemble des places *P* symbolise les conditions ou ressources du système (variables d'état). Les transitions *T* correspondent aux évènements qui modifient les informations de contrôle.

Dans ce qui suit on note:
$T_\Sigma(V)$: la Σ-algèbre des Σ-termes avec variables dans l'ensemble *S*-sorté *V*.
$mT_\Sigma(V)$: la Σ-algèbre des multi-ensembles de termes de $T_\Sigma(V)$
M(p): le marquage de la place *p* (i.e. le multi-ensemble de termes clos contenu dans la place *p* et qui correspond à la une valeur de la variable d'état *p*)

1- Application typage des places σ

L'application σ associe à chaque place une sorte de *Spec*.

$$\sigma : P \to S$$

Ainsi toutes les places d'un ECATNet sont typées.

2- Fonction capacité *Cap*:

C'est une fonction partielle qui associe à toute place *p* ($p \in dom(Cap)$) un entier positif spécifiant sa capacité.

$$Cap : P \to \mathbb{N} - \{0\}$$

La capacité d'une place est le nombre maximum de jetons qu'elle peut contenir (i.e c'est la cardinalité maximum du multi-ensemble de termes que peut contenir cette place).
Lorsqu'une place n'appartient pas au domaine de *Cap*, elle est considérée comme étant de capacité infinie.

3- Condition d'entrée *IC*

IC est une fonction partielle définie par:

$$IC : P \times T \rightarrow mT_{\Sigma}(V) \cup \{\sim \alpha / \alpha \in mT_{\Sigma}(V)\} \cup \{empty\}$$

$$\forall (p,t) \in dom(IC) \begin{bmatrix} IC(p,t) = \alpha \ \wedge \ \alpha \in mT_{\Sigma\sigma(p)}(V) \\ ou \\ IC(p,t) = \sim \alpha \ \wedge \ \alpha \in mT_{\Sigma\sigma(p)}(V) \\ ou \\ IC(p,t) = empty \end{bmatrix}$$

Remarque:

- $IC(p,t) = empty$ exprime la condition d'entrée $M(p) ==_{\sigma(p)} \varnothing_{\sigma(p)}$ (la place *p* est vide)
- $IC(p,t) = \alpha$ ($où \ \alpha \in mT_{\Sigma\sigma(p)}(V)$) correspond à la condition d'entrée $\alpha \subseteq_{\sigma(p)} M(p)$
  (Tous les jetons α appartiennent à la place *p*.)
- $IC(p,t) = \sim \alpha$($où \ \alpha \in mT_{\Sigma\sigma(p)}(V)$) dénote la condition d'entrée $\neg(\alpha \subseteq_{\sigma(p)} M(p))$
  (Les jetons α n'appatiennent pas à la place *p*.)

La condition d'entrée (*IC*) dans un réseau de Petri consiste en général à exiger *l'appartenance* des jetons aux places d'entrée de la transition à franchir. Dans les ECATNets on peut aussi imposer la *non-appartenance* d'un jeton à une place par la spécification (*IC(p,t)*=~α). On peut également tester si une place est vide (*IC(p,t)=empty*).

4- Jetons détruits *DT*

C'est une fonction partielle qui spécifie le multi-ensemble de jetons à détruire dans les places d'entrée pour chaque transition.

$$DT : P \times T \rightarrow mT_{\Sigma}(V) \cup \{\forall\}$$

$$\forall (p,t) \in dom(DT) \begin{bmatrix} DT(p,t) = \alpha \ \wedge \ \alpha \in mT_{\Sigma\sigma(p)}(V) \\ ou \\ DT(p,t) = \forall \end{bmatrix}$$

Remarque:

- $DT(p,t) = \alpha$ indique la destruction des termes α dans la place *p*.
  (i.e. $M(p) \leftarrow M(p) \ -_{\sigma(p)} \alpha$)
- $DT(p,t) = \forall$ exprime la destruction de *tous* les termes contenus dans la place *p*.
  (i.e. $M(p) \leftarrow \varnothing_{\sigma(p)}$)

Il faut noter que dans les ECATNets, à la différence des autres réseaux de Petri de haut niveau, on différencie entre les termes (jetons) qui satisfont la précondition (*IC*) et ceux qui sont détruits (*DT*). Ceci n'est généralement pas le cas dans les autres types de réseaux.

Cette distinction permet de modéliser le test de présence ou non d'une ressource qui ne sera pas consommée par la transition telle que par exemple les lectures parallèles d'une même donnée. De plus, les ECATNets offrent a la possibilité de détruire par une action *indivisible* (en spécifiant $DT(p,t)= \forall$) tous les jetons d'une place lors du franchissement d'une transition.

5- Jetons créés *CT*

Cette fonction précise les jetons créés dans les places de sortie de chaque transition.

$$CT : P \times T \rightarrow mT_{\Sigma}(V)$$

$$\forall (p,t) \in dom(CT) \quad CT(p,t) \in mT_{\Sigma\sigma(p)}(V)$$

6- Condition de transition *TC*

C'est une fonction partielle qui associe à une transition un terme de sorte *bool*.

$$TC : T \rightarrow T_{\Sigma\ bool}(V)$$

$$\forall t \in dom(TC) \quad TC(t) \in T_{\Sigma\ bool}(V_{ctx}(t))$$

$V_{ctx}(t)$ *appelé l''ensemble des variables du contexte de la transition t* est l'ensemble des variables apparaissant dans les arcs adjacents à *t* (i.e. dans *IC(p, t), DT(p, t)* et *CT(p, t)*).

La condition de transition est une condition supplémentaire au franchissement d'une transition, elle permet d'exprimer une contrainte fonctionnelle entre les variables composant la précondition de la transition, et/ou une synchronisation via certaines variables partagées.

7- Représentation graphique

Comme dans les autres types de réseaux de Petri (de haut niveau ou non), le réseau d'un ECATNet est représenté par un graphe orienté biparti dont l'ensemble des sommets est $P \cup T$ (les places étant représentées par des cercles ou des éllipses; et les transitions par des rectangles ou des barres). La représentation graphique de la cellule d'un réseau d'ECATNet a la forme suivante :

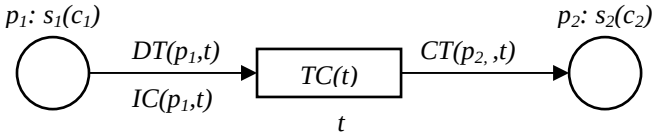
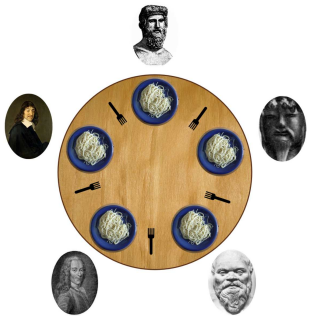


Figure1.3 : Représentation graphique de la cellule d'un réseau d'ECATNet.

Dans la Figure1.3, les places p_1 et p_2 sont respectivement de sorte s_1 et s_2 (i.e. $\sigma(p_1)= s_1$ *et* $\sigma(p_2)= s_2$) et de capacité c_1 et c_2 (i.e. $Cap\ (p_1)= c_1$ *et* $Cap\ (p_2) =c_2$). $IC(p_1,t)$ est écrite à gauche de l'arc en entrée à la transition *t* (pour un observateur situé au niveau de la transition). Les jetons à détruire $DT(p_1,t)$ sont écrits à droite de l'arc d'entrée à la transition *t*. Les jetons créés $CT(p_2,t)$ constituent la valeur de l'arc sortant de la transition *t* vers la place p_2. Dans le cas où $IC(p_1,t) = DT(p_1,t)$, une seule de ces inscriptions suffit dans le réseau.

8- Exemple

On considère l'exemple classique des philosophes avec cinq philosophes et cinq fourchettes.



Les composants de l'ECATNet spécifiant ce système sont donnés ci-dessous.

- Les structures de données sont les sortes *philo* pour les philosophes et *forks* pour les fourchettes. On établit une correspondance entre philosophes et leurs fourchettes à l'aide des opérations *left* et *right*.
- L'état du système est défini par trois variables d'état qui sont : une place *Thinking* contenant les philosophes qui pensent, une place *Eating* contenant les philosophes qui mangent, et une place *Idle* contenant les fourchettes disponibles.
- Les événements atomiques possibles dans ce système sont représentés par les transitions *Get* et *Put*. La première correspond à l'événement 'un philosophe va manger'. La seconde transition décrit l'événement 'un philosophe va penser'. Les arcs et leurs inscriptions expriment, d'une part sous quelles conditions un événement peut se produire, et d'autre part comment changent les variables d'état du système quand cet événement a lieu. Dans le cas de la transition *Get* par exemple, l'événement correspondant ne peut avoir lieu que si un philosophe *p* est dans l'état *Thinking* et ses fourchettes droite et gauche (*left(p)⊕right(p)*) sont disponibles. L'éxécution de cet événement fait passer le philosophe *p* de l'état *Thinking* à l'état *Eating* et supprime les deux fourchettes de l'ensemble des fourchettes libres.

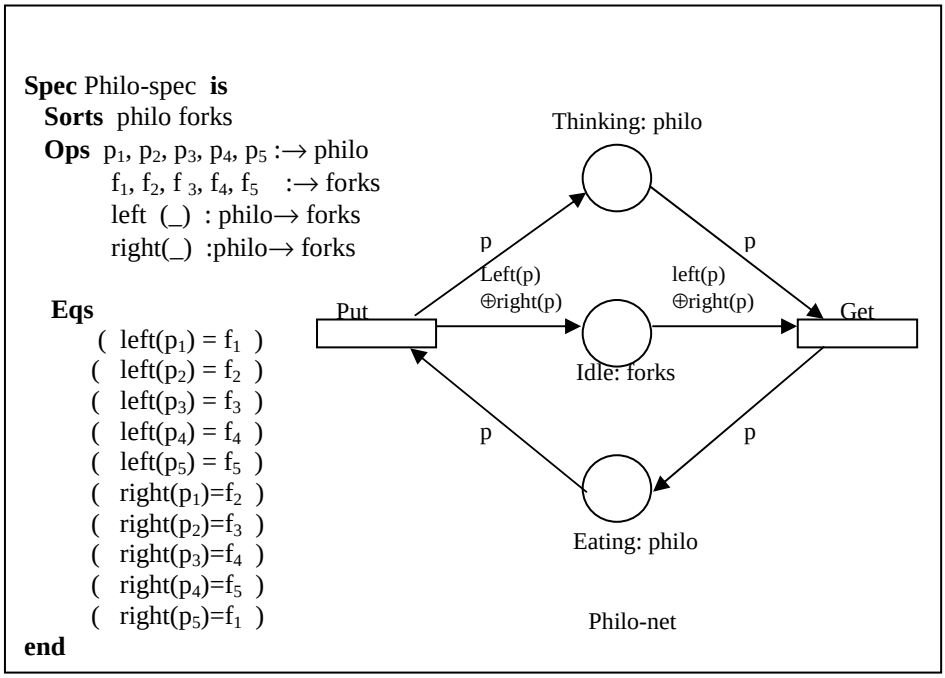


Figure1.4 : Un exemple d'ECATNet $\mathcal{E}$=(philo-spec, philo-net).

V- Marquage d'un ECATNet

L'état d'un ECATNet est défini par les multi-ensembles de valeurs associés aux différentes places du réseau.

1- Application Marquage

Soient $\mathcal{E}$: Un ECATNet
Spec : La spécification algébrique de $\mathcal{E}$
P : L'ensemble des places de $\mathcal{E}$
σ : Application de typage des places de $\mathcal{E}$
Cap : Fonction capacité de $\mathcal{E}$

Un marquage de l'ECATNet $\mathcal{E}$ est une application M qui associe à chaque place une valeur qui est un multi-ensemble de terme clos.

$M : P \rightarrow mT_{\Sigma}$

telle que: $\forall\, p \in P \;\; M(p) \in mT_{\Sigma\sigma(p)}(\varnothing) \wedge \left(p \in dom(Cap) \rightarrow | M(p) |_{\sigma(p)} \leq Cap(p) \right)$

Le marquage d'un ECATNet indique quelles marques (termes) sont présentes dans chaque place du réseau. Chaque marque symbolisant une condition réalisée ou un exemplaire de ressource disponible.

2- ECATNet marqué

Un ECATNet marqué est le couple ($\mathcal{E}$, M) tel que :
$\mathcal{E}$: est un ECATNet.
M: est un marquage de $\mathcal{E}$.

3- Restriction de marquage

Soit $\mathcal{E}$ = *(Spec, R)* un ECATNet tel que :
$Spec = (\Sigma, E)$.
P: l'ensemble des places de $\mathcal{E}$.
M: un marquage de $\mathcal{E}$.
P': un sous-ensemble de places de $\mathcal{E}$ ($P' \subseteq P$).

La restriction de M au sous-ensemble P' notée $M_{/P'}$ est définie par l'application:

$M_{/P'} : P' \rightarrow mT_{\Sigma}(\varnothing)$ telle que $\forall p \in P' \;\; M_{/P'}(p) = M(p)$

$M_{/P'}$ est appelé aussi marquage partiel de $\mathcal{E}$

4- Sous-marquage

Soit $\mathcal{E}$ = *(Spec, R)* un ECATNet tel que :
P: l'ensemble des places de $\mathcal{E}$.
M_1, M_2: deux marquages partiels de $\mathcal{E}$.

M_1 est un sous-marquage de M_2 noté $M_1 \subseteq M_2$ si et seulement si:

i) $dom(M_1) \subseteq dom(M_2)$
ii) $\forall\, p \in dom(M_1) \quad M_1(p) \subseteq_{\sigma(p)} M_2(p)$

VI- Comportement d'un ECATNet

Le comportement d'un ECATNet peut être décrit en précisant les règles de franchissement des transitions. Ces règles sont extrêmement importantes puisqu'elles décrivent le fonctionnement du schéma de contrôle du système spécifié. Trois points sont fondamentaux dans le fonctionnement des ECATNets:

- ***L'indéterminisme*:**

L'indéterminisme dans le comportement d'un ECATNet est à trois niveaux:

i)- L'indéterminisme dans le *choix de la transition à franchir*:
Si plusieurs transitions sont franchissables et elles ne peuvent être franchies en parallèles, une d'entre elles est choisie au *hasard* et est franchie. Il n'existe aucun ordre relatif de franchissement des transitions, i.e. le choix de franchissement est equiprobable et non-déterministe. Ce type d'indéterminisme est commun à tous les types de réseaux de Petri connus (de haut niveau ou non) qui ne définissent pas une relation d'ordre total de priorité entre les transitions.

ii)- L'indéterminisme dans *le choix des jetons* (multi-ensemble de termes) lors du franchissement d'une transition:
Dans le cas où une transition est franchissable et différents multi-ensembles de termes satisfont sa précondition; un multi-ensemble est choisi de façon *arbitraire* pour le franchissement. Ce type d'indéterminisme n'est pas connu dans les réseaux de Petri qui ne différencient pas les jetons d'une même place (tels que les réseaux de Petri simples, les premières versions des réseaux colorés où chaque place a une seule couleur).

iii)- L'indéterminisme dans *le franchissement d'une transition franchissable :*
Une transition franchissable peut être *franchie ou non*. Il est possible qu'une transition franchissable ne soit jamais franchie, et que dans un réseau aucune transition ne soit franchie parmi toutes celles qui sont franchissables.

- ***L'indivisibilité:***

L'évaluation de la condition de franchissabilité et l'exécution des règles de franchissement d'une transition est une action *indivisible* et *instantanée*. Le marquage des places en entrée et en sortie d'une transition est modifié en même temps que cette transition est franchie.

- ***La concurrence***:

C'est un point important qui caractérise les ECATNets en permettant le maximum de parallélisme dans les systèmes spécifiés. Cette concurrence dans le franchissement des transitions peut être à deux niveaux:

- *Concurrence entre un ensemble de transitions*: des transitions franchissables peuvent être franchies en parallèles (si ceci est possible) de façon concurrentes.

- *Auto-concurrence*: une transition peut être franchie parallèlement à elle même (degré de parallélisme pouvant être un nombre quelconque de fois) tant que sa précondition est vérifiée.

Afin de pouvoir décrire clairement le comportement d'un ECATNet et mettre l'accent sur les différents points qui le caractérisent, il est nécessaire de définir certains concepts tels que le *contexte* d'une transition, la *franchissabilité* (« enabling ») et le *franchissement* (« firing ») d'une transition, le franchissement *concurrent* et *auto-concurrent* des transitions dans un ECATNet.

1- Contexte d'une transition

Le contexte d'une transition *t* est l'ensemble *S*-sorté des variables apparaissant dans les arcs adjacents à *t* et dans la condition de transition associée à *t*. Il détermine donc le nombre et les types des variables utilisées par la transition *t*. On note *ctx* l'application qui pour chaque transition *t* définit le contexte de *t*.

1-1 Définition

Le contexte d'une transition *t* noté *ctx(t)* est défini par l'application:

$$ctx \ : \ T \to (V \times S)^*$$

$(V \times S)^*$ est l'ensemble des mots sur $(V \times S)$ où $\varnothing$ désigne le mot vide de (VxS)*

et la loi de composition (*) est supposée commutative.

V: est l'ensemble des variables,

S: est l'ensemble des sortes.

On notera :

- $(x_1 : s_1, x_2 : s_2, ..., x_n : s_n)$ le contexte $(x_1, s_1)(x_2, s_2)...(x_n, s_n)$
- $V_{ctx}(t)$: l'ensemble des variables de *ctx(t)* i.e. $V_{ctx}(t) = \{x_1, x_2, ..., x_n\}$
- $\sigma_{ctx(t)}$: l'application qui associe à chaque variable de $V_{ctx}(t)$ une sorte : $\forall i = 1..n \ \sigma_{ctx(t)}(x_i) = s_i$

Exemple:

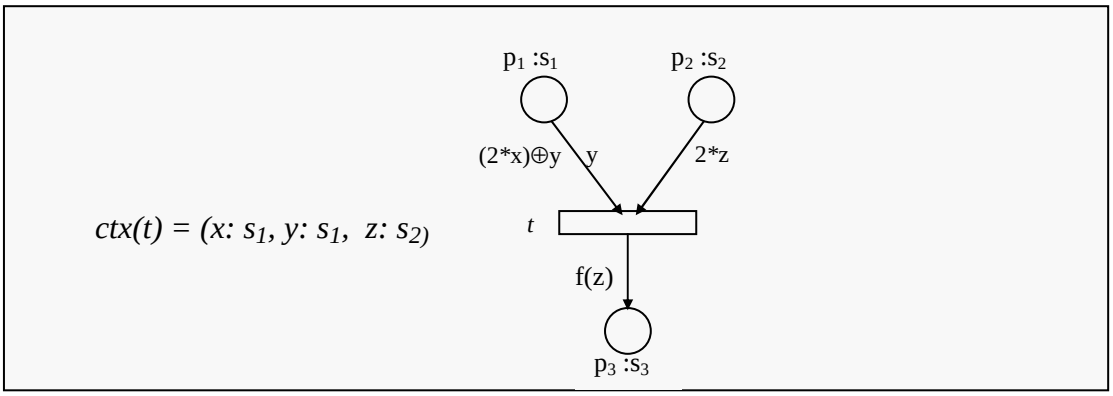


Figure 1.5 : Contexte d'une transition.

L'application *ctx* peut être déduite des différentes applications σ, *IC*, *CT* ,*DT*, *TC* de l'ECATNet considéré.

1-2 Affectation de valeurs au contexte d'une transition

C'est une application qui à toute variable de sorte *s* appartenant à *ctx(t)* associe une valeur (un terme clos) de même sorte.

$$aff_t \ : \ V_{ctx}(t) \to T_\Sigma \ \text{ telle que } \forall x_i \in V_{ctx}(t) \quad aff_t(x_i) \in T_{\Sigma_{\sigma_{ctx(t)}(x_i)}}$$

1-3 Consistance d'une affectation

Une affectation aff_t du contexte d'une transition t est consistante si et seulement si on a simultanément et pour toute place *p* adjacente à *t:*

(i) $[IC(p,t)]_{aff_t} \in mT_{\Sigma\sigma(p)}(\varnothing) \cup \{\sim \alpha / \alpha \in mT_{\Sigma\sigma(p)}(\varnothing)\} \cup \{empty\}$

(ii) $[DT(p,t)]_{aff_t} \in mT_{\Sigma\sigma(p)}(\varnothing) \cup \{\forall\}$

(iii) $[CT(p,t)]_{aff_t} \in mT_{\Sigma\sigma(p)}(\varnothing)$

(iv) $[TC(t)]_{aff_t} \in T_{\Sigma bool}(\varnothing)$

Notation:
$[Exp]_{aff_t}$: est l'expression obtenue en substituant dans *Exp* les variables du contexte de *t* par les valeurs correspondantes selon l'affectation aff_t**.**

Exemple: Dans le réseau représenté par la Figure1.6 ci-dessous, l'affectation aff_t^1 telle que $aff_t^1(x) = 2$ est consistante car 2 est de sorte *real* et 2+1 est de sorte *nat*. Par contre l'affectation aff_t^2 définie par $aff_t^2(x) = \pi$ est non consistante car π+1 n'est pas de sorte *nat*.

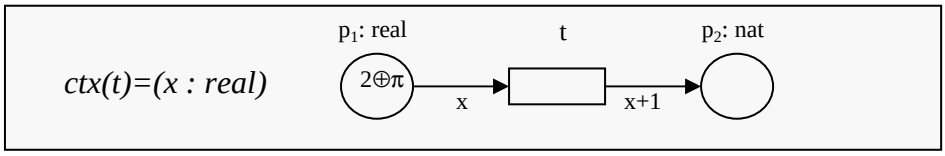


Figure 1.6 : Consistance/inconsistance d'une affectation de valeurs au contexte d'une transition.

2- Franchissabilité ("enabling") d'une transition

2-1- Franchissabilité simple

Une transition *t* de $\mathcal{E}$ est franchissable (ou simplement franchissable) pour un marquage *M* si et seulement si il existe au moins une affectation consistante aff_t de *ctx(t)* telle que les conditions suivantes soient vérifiées simultanément:

i) La condition d'entrée *IC* est satisfaite:
$\forall p \in P \ \forall (p,t) \in dom(IC)$ on a:

$M(p) ==_{\sigma(p)} \varnothing_{\sigma(p)}$ *si* $IC(p,t) = empty$

$\alpha \subseteq_{\sigma(p)} M(p)$ *si* $[IC(p,t)]_{aff_t} =_E \alpha$

$\neg(\alpha \subseteq_{\sigma(p)} M(p))$ *si* $[IC(p,t)]_{aff_t} =_E \sim \alpha$

ii) Les jetons à détruire sont disponibles
$\forall p \in P \ \forall (p,t) \in dom(DT)$ on a :

$\alpha \subseteq_{\sigma(p)} M(p)$ *si* $([DT(p,t)]_{aff_t} =_E \alpha) \wedge (DT(p,t) \neq \forall)$

iii) La condition de transition *TC* est vraie:

$$[TC(t)]_{aff_t} =_E true \quad \text{si } t \in dom(TC)$$

iv) Non dépassement des capacités des places de sortie de *t* :

$$\forall p \in dom(Cap) \qquad \left| M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p) \right|_{\sigma(p)} \leq Cap(p)$$

où

$$\begin{cases} M_{cr}(p) = [CT(p,t)]_{aff_t} & \text{si } (p,t) \in dom(CT) \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(CT) \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } (p,t) \in dom(DT) \wedge DT(p,t) = \forall \\ M_{dt}(p) = [DT(p,t)]_{aff_t} & \text{si } (p,t) \in dom(DT) \wedge DT(p,t) \neq \forall \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(DT) \end{cases}$$

Notation:

$[Exp]_{aff_t}$: est le terme obtenu à partir de *Exp* en substituant les variables par des valeurs selon une affectation consistante de aff_t .

$x =_E y$: dénote l'équivalence de *x* et *y* modulo les équations *E* de la spécification algébrique de l'ECATNet $\mathcal{E}$.

2-2 Franchissabilité concurrente

Un sous-ensemble de transitions $T' \subseteq T$ d'un ECANet $\mathcal{E}$, est franchissable de façon concurrente à un marquage *M* si et seulement si il existe au moins une famille d'affectations consistantes $(aff_t)_{t \in T'}$ telles que les conditions suivantes soient vérifiées simultanément:

i) La condition d'entrée *IC* de chaque transition de *T'* est satisfaite:

$\forall t \in T'$ on a:

$$M(p) ==_{\sigma(p)} \varnothing_{\sigma(p)} \qquad si\ IC(p,t) ==_{\sigma(p)} empty$$

$$\alpha \subseteq_{\sigma(p)} M(p) \qquad si\ [IC(p,t)]_{aff_t} =_E \alpha$$

$$\neg(\alpha \subseteq_{\sigma(p)} M(p)) \qquad si\ [IC(p,t)]_{aff_t} =_E \sim \alpha$$

ii) Les jetons à détuire par l'ensemble des transitions *T'* sont disponibles:

$$\forall p \in P \quad \alpha \subseteq_{\sigma(p)} M(p) \qquad si\ (\oplus_{\sigma(p)})_{t \in T''} [DT(p,t)]_{aff_t} =_E \alpha$$

où $T'' = \{t \in T' / (p,t) \in dom(DT) \wedge DT(p,t) \neq \forall\}$

iii) Les conditions de transitions sont vraies:

$$((and)_{t \in T''} [TC(t)]_{aff_t}) =_E true \quad \text{où} \quad T'' = \{t / \ t \in T' \cap dom(TC)\}$$

iv) Non dépassement des capacités des places :

$$\forall p \in dom(Cap) \quad \left| M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p) \right|_{\sigma(p)} \leq Cap(p)$$

où

$$\begin{cases} M_{cr}(p) = (\oplus_{\sigma(p)})_{t \in T'_{cr}} [CT(p,t)]_{aff_t} & \text{si } T'_{cr} \neq \varnothing \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } T'_{cr} = \varnothing \\ & \text{où } T'_{cr} = \{ t \in T' / \ (p,t) \in dom(CT) \} \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } T'_{dt} \neq \varnothing \wedge T'_{\forall} \neq \varnothing \\ M_{dt}(p) = (\oplus_{\sigma(p)})_{t \in T'_{dt}} [DT(p,t)]_{aff_t} & \text{si } T'_{dt} \neq \varnothing \ \wedge T'_{\forall} = \varnothing \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } T'_{dt} = \varnothing \\ & \text{où } T'_{dt} = \{ t \in T' \ (p,t) \in dom(DT) \} \\ & \quad T'_{\forall} = \{ t \in T' / (p,t) \in dom(DT) \wedge DT(p,t) = \forall \} \end{cases}$$

Proposition:

Dans un ECATNet, la franchissabilité concurrent d'un sous-ensemble de transitions $T' \subseteq T$ dans un marquage M n'implique pas la franchissabilité simple de chacune des transitions de T' dans M.

Preuve: (par contre-exemple)
Soit l'ECATNet ci-dessous, tel que la place *p* de capacité *1* est marquée par *{a}* (i.e. *M(p)=a* et *Cap(p)=1*).

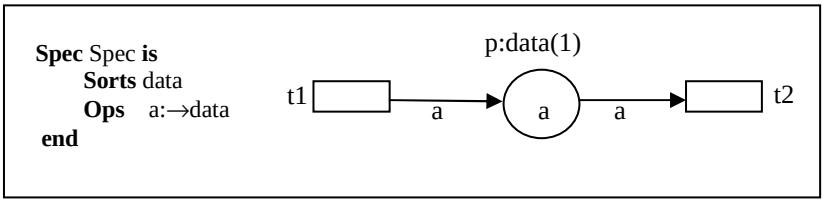


Figure 1.7 : Exemple de réseau franchissable de façon concurrente non entrelacable.

Les transitions t_1 et t_2 sont franchissables en parallèles bien que t_1 ne soit pas (simplement) franchissable car la condition (iv) du &2.1 n'est pas vérifiée (dépassement de la capacité de *p*)

2-3 Franchissabilité auto-concurrente

Une transition *t* est franchissable *n* fois de façon auto-concurrente pour un marquage *M* si et seulement si il existe au moins une famille d'affectations consistantes $(aff_t^i)_{i=1..n}$ telles que les conditions suivantes soient vérifiées simultamément:

i) La condition d'entrée *IC* est satisfaite:

$\forall\, p \in P\ \forall (p,t) \in dom(IC)\ \forall i = 1..n$ on a:

$M(p) ==_{\sigma(p)} \varnothing_{\sigma(p)}$ *si* $IC(p,t) = empty$

$\alpha \subseteq_{\sigma(p)} M(p)$ *si* $[IC(p,t)]_{aff_t^i} =_E \alpha$

$\neg(\alpha \subseteq_{\sigma(p)} M(p))$ *si* $[IC(p,t)]_{aff_t^i} =_E \sim \alpha$

ii) Les jetons à détuire sont disponibles:

$\forall\, p \in P \qquad \alpha \subseteq_{\sigma(p)} M(p) \qquad si\ (\oplus_{\sigma(p)})_{i=1..n}\ [DT(p,t)]_{aff_t^i} =_E \alpha \wedge DT(p,t) \neq \forall$

iii) La condition de transition est vraie pour toutes les affectations :

$((and)_{i=1..n}[TC(t)]_{aff_t^i}) =_E true$ si $t \in dom(TC)$

iv) Les capacités des places de sortie de *t* ne sont pas dépassées :

$\forall p \in dom(Cap) \quad \left| M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p) \right|_{\sigma(p)} \leq Cap(p)$

où

$$\begin{cases} M_{cr}(p) = (\oplus_{\sigma(p)})_{i=1..n}[CT(p,t)]_{aff_t^i} & \text{si } (p,t) \in dom(CT) \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(CT) \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } (p,t) \in dom(DT) \wedge DT(p,t) = \forall \\ M_{dt}(p) = (\oplus_{\sigma(p)})_{i=1..n}[DT(p,t)]_{aff_t^i} & \text{si } (p,t) \in dom(DT) \wedge DT(p,t) \neq \forall \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(DT) \end{cases}$$

Proposition:

Dans un ECATNet, si une transition *t* est franchissable *n* fois *(n>1)* de façon auto-concurrente pour le marquage *M*, alors *t* est franchissable *n-1* fois de façon auto-concurrente pour *M*.

Preuve: (évidente)
Les conditions i) , ii) iii) et iv) du &2.3 sont vraies pour *i=1..n*. Elles restent vérifiées pour *i=1..n-1*.

2-4 Franchissabilité mixte

C'est le type de franchissabilité qui regroupe les trois précédents et qui est défini dans les ECATNets. Les définitions ci-dessus sont généralisées aux *multi-ensembles de transitions.* A chaque transition *t* est associé un entier qui précise le degré d'auto-concurrence de *t*.

Un multi-ensemble de transition $\tau : T \to \mathbb{N}$ est franchissable au marquage *M* si et seulement si il existe au moins une famille d'affectations $((aff_t^i)_{i=1..\tau(t)})_{t \in T}$ telles que les conditions suivantes soient vérifiées simultanément:

i) La condition d'entrée IC de chaque transition est satisfaite:

$\forall t \in \{t \in T / \tau(t) \neq 0\} \forall i \in \{1,..,\tau(t)\}\ \forall (p,t) \in dom(IC)$ on a:

$M(p) ==_{\sigma(p)} \varnothing_{\sigma(p)}$ *si* $IC(p,t) = empty$

$\alpha \subseteq_{\sigma(p)} M(p)$ *si* $[IC(p,t)]_{aff_t^i} =_E \alpha$

$\neg(\alpha \subseteq_{\sigma(p)} M(p))$ *si* $[IC(p,t)]_{aff_t^i} =_E \sim \alpha$

ii) L'ensemble des jetons à détruire par toutes les transitions est disponible dans les places :

$$\forall p \in P \quad \alpha \quad \subseteq_{\sigma(p)} M(p) \quad si\ (\oplus_{\sigma(p)})_{t \in T'}((\oplus_{\sigma(p)})_{i=1..\tau(t)}\ [DT(p,t)]_{aff_t^i}) =_E \alpha$$

$$\text{où } T' = \{t \in T /\ \tau(t) \neq 0 \wedge (p,t) \in dom(DT) \wedge DT(p,t) \neq \forall\}$$

iii) Les conditions de transitions sont vraies:

$((and)_{t \in T'}((and)_{i=1..\tau(t)}[TC(t)]_{aff_t^i})) =_E true$ où $T' = \{t /(t \in T \cap dom(TC)) \wedge (\tau(t) \neq 0)\}$

iv) Non dépassement des capacités des places:

$$\forall p \in dom(Cap) \quad \left| M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p) \right|_{\sigma(p)} \leq Cap(p)$$

où

$$\begin{cases} M_{cr}(p) = (\oplus_{\sigma(p)})_{t \in T_{cr}}((\oplus_{\sigma(p)})_{i=1,..\tau(t)}[CT(p,t)]_{aff_t^i}) & \text{si } T_{cr} \neq \varnothing \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } T_{cr} = \varnothing \end{cases}$$

$$\text{où } T_{cr} = \{t \in T / \tau(t) \neq 0 \wedge\ (p,t) \in dom(CT)\}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } T_{dt} \neq \varnothing \wedge T_{\forall} \neq \varnothing \\ M_{dt}(p) = (\oplus_{\sigma(p)})_{t \in T_{dt}}((\oplus_{\sigma(p)})_{i=1,..\tau(t)}[DT(p,t)]_{aff_t^i}) & \text{si } T_{dt} \neq \varnothing\ \wedge T_{\forall} = \varnothing \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } T_{dt} = \varnothing \end{cases}$$

$$\text{où } T_{dt} = \{t \in T / (\tau(t) \neq 0 \wedge (p,t) \in dom(DT)\}$$

$$T_{\forall} = \{t \in T / \tau(t) \neq 0 \wedge (p,t) \in dom(DT) \wedge DT(p,t) = \forall\}$$

Notation :

$M[(t, aff_t)>$: t est franchissable dans le marquage M pour l'affectation aff_t.

$M\left[\|_{i=1,n}(t, aff_t^{\,i})>\right.$: t est franchissable de façon autoconcurente n fois dans le marquage M pour la famille d'affectations $(aff_t^{\,i})_{i=1,..n}$.

Cette notation est équivalente à $M\left[(t, aff_t^{\,1}) \| (t, aff_t^{\,2}) .... \| (t, aff_t^{\,n})>\right.$.

$M\left[\|_{i=1,n}(t_i, aff_{t_i})>\right.$: les transitions $t_1, t_2 ..,.t_n$ sont franchissables de façon concurrente dans le marquage M pour la famille d'affectations $(aff_{t_i})_{i=1,..n}$.

Cette notation est équivalente à $M\left[(t_1, aff_{t_1}) \| (t_2, aff_{t_2}) .... \| (t_n, aff_{t_n})>\right.$.

$M\left[\|_{i=1,n}(\|_{j=1..\tau(t_i)}(t_i, aff_{t_i}^{j}))\right.>$: les transitions $t_1, t_2 ..,.t_n$ sont franchissables de façon concurrente (chacune d'elles avec un taux d'autoconcurrence $\tau(t_i)$) dans le marquage M pour la famille d'affectations $((aff_{t_i}^{j})_{j=1..\tau(t_i)})_{i=1..n}$.

3-Franchissement d'une transition

3-1 Franchissement simple

Le franchissement d'une transition *t* consiste à oter des multi-ensembles de termes (précisés par *DT(p,t)*) de chaque place en entrée de *t* et à ajouter les multi-ensemblec *CT(p,t)* dans chaque place de sortie de *t* (ces opérations étant indivisibles).

Soit *t* une transition de $\mathcal{E}$, franchissable pour un marquage M et une affectation aff_t de *ctx (t)*. Le franchissement de *t* donne un nouveau marquage *M'* défini par:

$$\forall p \in P \qquad M'(p) = M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p)$$

où

$$\begin{cases} M_{cr}(p) = \alpha & \text{si } (p,t) \in dom(CT) \wedge [CT(p,t)]_{aff_t} =_{\mathrm{E}} \alpha \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(CT) \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } (p,t) \in dom(DT) \wedge DT(p,t) = \forall \\ M_{dt}(p) = \alpha & \text{si } (p,t) \in dom(DT) \wedge DT(p,t) \neq \forall \wedge [DT(p,t)]_{aff_t} =_{E} \alpha \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(DT) \end{cases}$$

3-2 Franchissement concurrent

Le franchissement concurrent d'un ensemble de transitions $T' \subseteq T$ consiste à oter des multi-ensembles de termes (précisés par *DT*) de chaque place en entrée de *t* ($t \in T'$) et à ajouter les multi-ensembles *CT(p,t)* dans chaque place de sortie de *t* ($t \in T'$). Toutes ces opérations sont effectuées simultanément.
Soit *T'* un ensemble de transitions de $\mathcal{E}$ franchissable de façon concurrente pour un marquage M et une famille d' affectations $(aff_t)_{t \in T'}$ de $ctx(t)_{t \in T'}$. Le franchissement concurrent de *T'* donne un nouveau marquage *M'* défini par:

$$\forall p \in P \qquad M'(p) = M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p)$$

où:

$$\begin{cases} M_{cr}(p) = \alpha & \text{si } T_{cr}^{'} \neq \varnothing \ \wedge (\oplus_{\sigma(p)})_{t \in T_{cr}^{'}} [CT(p,t)]_{aff_t} =_{\mathrm{E}} \alpha \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } T_{cr}^{'} = \varnothing \\ & \text{où } T_{cr}^{'} = \{ t \in T' / \ (p,t) \in dom(CT) \} \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } T'_{dt} \neq \varnothing \wedge T'_{\forall} \neq \varnothing \\ M_{dt}(p) = \alpha & \text{si } T'_{dt} \neq \varnothing \ \wedge T'_{\forall} = \varnothing \wedge (\oplus_{\sigma(p)})_{t \in T'_{dt}} [DT(p,t)]_{aff_t} =_E \alpha \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } T'_{dt} = \varnothing \end{cases}$$

où $T'_{dt} = \{t \in T' \ (p,t) \in dom(DT)\}$

$T'_{\forall} = \{t \in T' / (p,t) \in dom(DT) \wedge DT(p,t) = \forall\}$

Proposition:

Dans un ECATNet, le franchissement concurrent d'un ensemble de transitions T' ($T' \subseteq T$) dans un marquage *M*, ne donne pas, en général, le même marquage que le franchissement simple (non concurrent) et successif des transitions de *T'*.

Preuve: (conséquence de la proposition du & 2.2)

3-3 Franchissement auto-concurrent

Soit *t* une transition franchissable *n* fois de façon auto-concurrente pour un marquage *M* et une famille d'affectations consistantes $(aff_t^i)_{i=1,n}$. Le franchissement de *t* donne le nouveau marquage *M'* défini par:

$$\forall p \in P \quad M'(p) = M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p)$$

où

$$\begin{cases} M_{cr}(p) = \alpha & \text{si } (p,t) \in dom(CT) \wedge \ (\oplus_{\sigma(p)})_{i=1..n} [CT(p,t)]_{aff_t^i} =_E \alpha \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(CT) \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } (p,t) \in dom(DT) \ \wedge DT(p,t) = \forall \\ M_{dt}(p) = \alpha & \text{si } (p,t) \in dom(DT) \ \wedge DT(p,t) \neq \forall \wedge (\oplus_{\sigma(p)})_{i=1..n} [DT(p,t)]_{aff_t^i} =_E \alpha \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } (p,t) \notin dom(DT) \end{cases}$$

Proposition:

Si une transition *t* est franchissable *n* fois de façon auto-concurrente, le franchissement simple de *t* n'implique pas que *t* reste franchissable *n-1* fois de façon auto-concurrente.

Preuve: (par contre-exemple)
Soit l'ECATNet ci-dessous, tel que la place *p* est marquée par *M* tel que *M(p)=a⊕a⊕a*

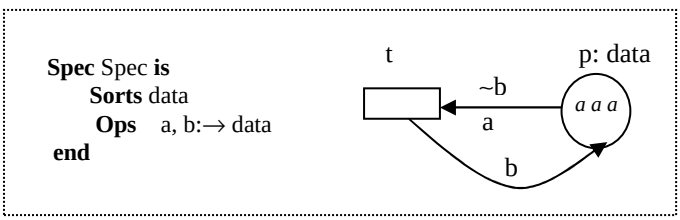


Figure 1.8 : Exemple d'ECATNet où le franchissement auto-concurrent est non entrelacable.

La transition *t* est franchissable 3 fois de façon auto-concurrente. Le frachissement simple de *t* donne le marquage $\{b \oplus a \oplus a\}$ pour la place *p* auquel cas *t* n'est plus franchissable (par conséquent *t* ne reste pas franchissable 2 fois de façon auto-concurrente). Le franchissement auto-concurrent (*3 fois*) de la transition *t* aurait donné le marquage $\{b \oplus b \oplus b\}$ pour *p*.

3-4 Franchissement mixte

Soit $\tau : T \to \mathbb{N}$ un multi-ensemble de transitions franchissables pour le marquage *M* et la famille d'affectations $((aff_t^i)_{i=1..n})_{t \in T}$. Le franchissement de ce multi-ensemble de transitions donne le marquage *M'* défini par:

$$\forall p \in P \; M'(p) = M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p)$$

où

$$\begin{cases} M_{cr}(p) = \alpha & \text{si } T_{cr} \neq \varnothing \;\wedge\; (\oplus_{\sigma(p)})_{t \in T_{cr}} ((\oplus_{\sigma(p)})_{i=1,..\tau(t)} [CT(p,t)]_{aff_t^i}) =_E \alpha \\ M_{cr}(p) = \varnothing_{\sigma(p)} & \text{si } T_{cr} = \varnothing \\ & \text{où } T_{cr} = \{t \in T / \tau(t) \neq 0 \wedge (p,t) \in dom(CT)\} \end{cases}$$

$$\begin{cases} M_{dt}(p) = M(p) & \text{si } T_{dt} \neq \varnothing \wedge T_\forall \neq \varnothing \\ M_{dt}(p) = \alpha & \text{si } T_{dt} \neq \varnothing \;\wedge T_\forall = \varnothing \wedge (\oplus_{\sigma(p)})_{t \in T_{dt}} ((\oplus_{\sigma(p)})_{i=1,..\tau(t)} [DT(p,t)]_{aff_t^i}) =_E \alpha \\ M_{dt}(p) = \varnothing_{\sigma(p)} & \text{si } T_{dt} = \varnothing \\ & \text{où } T_{dt} = \{t \in T / (\tau(t) \neq 0 \wedge (p,t) \in dom(DT)\} \\ & \quad T_\forall = \{t \in T / \tau(t) \neq 0 \wedge (p,t) \in dom(DT) \wedge DT(p,t) = \forall\} \end{cases}$$

Notation :

$M[(t, aff_t)\rangle M'$: le franchissement de *t* dans le marquage *M* pour l'affectation aff_t donne le marquage *M'*.

$M\left[\|_{i=1,n}(t, aff_t^i)\right\rangle M'$: le franchissement de *t* de façon autoconcurente *n* fois dans le marquage *M* pour la famille d'affectations $(aff_t^i)_{i=1,..n}$ donne le marquage *M'*.

Cette notation est équivalente à $M\left[(t, aff_t^1) \| (t, aff_t^2) .... \| (t, aff_t^n)\right\rangle M'$.

$M\left[\|_{i=1,n}(t_i, aff_{t_i})\right\rangle M'$: le franchissement des transitions $t_1, t_2 ..,.t_n$ de façon concurrente dans le marquage *M* pour la famille d'affectations $(aff_{t_i})_{i=1,..n}$ donne le marquage *M'*.

Cette notation est équivalente à $M\left[(t_1, aff_{t_1}) \| (t_2, aff_{t_2}) .... \| (t_n, aff_{t_n})\right\rangle M'$.

$M\left[\|_{i=1,n}(\|_{j=1,\tau(t_i)}(t_i, aff_{t_i}^j))\right\rangle M'$: le franchissement des transitions $t_1, t_2 ..,.t_n$ de façon concurrente (chacune d'elles avec un taux d'autoconcurrence $\tau(t_i)$) dans le marquage *M* pour la famille d'affectations $((aff_{t_i}^j)_{j=1..\tau(t_i)})_{i=1..n}$ donne le marquage *M'*.

3-5 Exemple

Soit l'ECATNet $\mathcal{E}_1 = (Spec_1, R_1)$ présenté par la Figure 1.9. A partir de l'état initial (défini par le marquage initial M_0) on peut effectuer au moins l'un des franchissement suivants:

-Le franchissement simple de t_1 pour l'affectation $x=a$ qui donne le marquage M_1
-Le franchissement simple de t_1 pour l'affectation $x=b$ qui donne le marquage M_2
-Le franchissement concurrent de t_1 et t_2 pour l'affectation $x=b$ qui donne le marquage M_5
-Le franchissement auto-concurrent (*2 fois*) de la transition t_1 pour $x=a$ et $x=b$ donnant le marquage M_7
-Le franchissement mixte: auto-concurrence (*2 fois*) de la transition t_1 en même temps que le franchissement de t_2 donnant le marquage M_{10}.

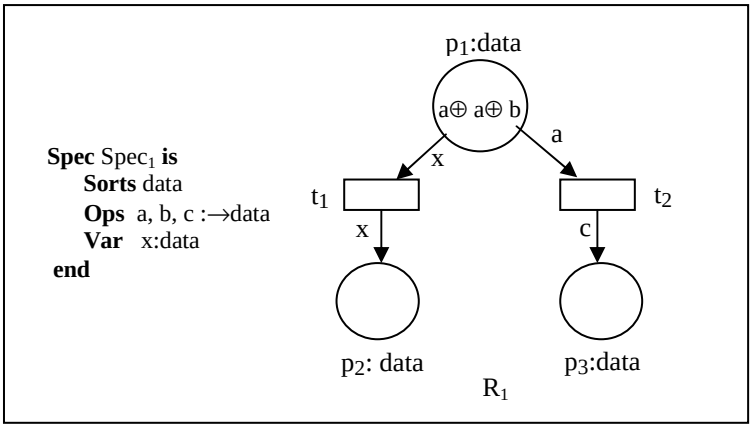


Figure 1.9 : Exemple d'ECATNet où différents types de franchissements sont possibles.

Le tableau ci-dessous donne quelques marquages accessibles à partir de M_0.

	p_1	p_2	p_3	Transitions franchies	aff_t	Type de franchissement
M_1	a⊕b	a	∅	t_1	x=a	simple
M_2	a⊕a	b	∅	t_1	x=b	simple
M_3	a⊕b	∅	c	t_2		simple
M_4	b	a	c	t_1 t_2	x=a	concurrent
M_5	a	b	c	t_1 t_2	x=b	concurrent
M_6	b	a⊕a	∅	t_1 t_1	x=a x=a	auto-concurrent
M_7	a	a⊕b	∅	t_1 t_1	x=a x=b	auto-concurrent
M_8	∅	a⊕a⊕b	∅	t_1 t_1 t_1	x=a x=a x=b	auto-concurrent
M_9	b	∅	c⊕c	t_2 t_2		auto-concurrent
M_{10}	∅	a⊕b	c	t_1 t_1 t_2	x=a x=b	mixte
M_{11}	∅	b	c⊕c	t_1 t_2 t_2	x=b	mixte

Il est clair que pour le franchissement d'une même transition à partir du même marquage, on peut obtenir des marquages différents suivant les affectations de valeurs aux variables du contexte de la transition en question. De plus, ce marquage ne dépend pas seulement du choix de la transition à franchir (quand plusieurs transitions sont franchissables) et de l'affectation de valeurs au contexte de cette transition, mais aussi du type de franchissement considéré (non concurrent, concurrent, auto-concurrent, ou mixte).

4- Marquages accessibles

4-1 Ensemble des marquages accessibles

Le franchissement des transitions fait évoluer l'ECATNet d'un marquage à un autre. L'ensemble des différents états possibles d'un système modélisé par un ECATNet est défini par l'ensemble des marquages accessibles.

L'ensemble de tous les marquages accessibles de $\mathcal{E}$ à partir du marquage M_0 est notée $\mathcal{M}(\mathcal{E},M_0)$ et est défini inductivement par :

$$\begin{cases} M_0 \in \mathcal{M}(\mathcal{E},M_0) \\ \text{Si} \quad M \in \mathcal{M}(\mathcal{E},M_0) \quad \wedge \; M\left[\left\|_{i=1,n}\left(\left\|_{j=1..\tau(t_i)}(t_i, aff_{t_i}^{\,j})\right.\right.\right. > M' \quad \text{alors} \quad M' \in \mathcal{M}(\mathcal{E},M_0) \end{cases}$$

Notation:

$M>>M'$: *M'est* accessible à partir de *M*.

$\mathcal{M}(\mathcal{E},M)$: est l'ensemble de tous les marquages accessibles de $\mathcal{E}$ à partir du marquage *M*.

$\mathcal{M}(\mathcal{E},M)_{/P'}$: l'ensemble de tous les marquages accessibles de $\mathcal{E}$ à partir de *M* , restreint à l'ensemble des places $P' \subseteq P$.

4-2 Graphe des marquages accessibles

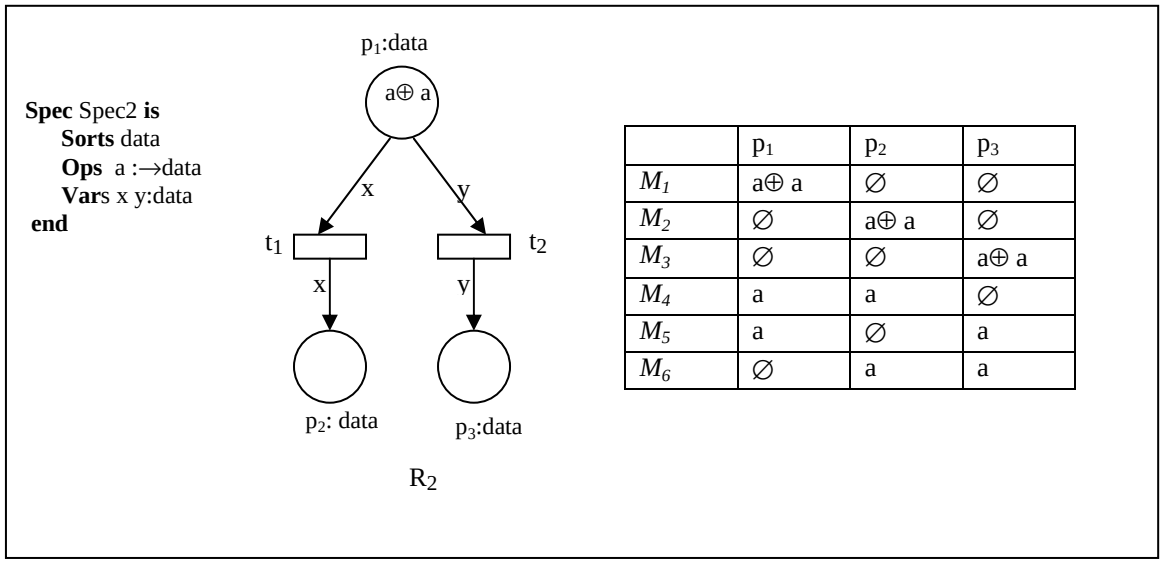
Le graphe des marquage accessibles d'un ECATNet marqué $(\mathcal{E},M)$ est le graphe ayant les éléments de $\mathcal{M}(\mathcal{E},M)$ pour ensemble de sommets et dont les arcs sont définis par la relation :

$M'(>M''$ si et seulement si $\exists \{t_1,t_2,..t_n\} \subseteq T \,/\, M'\left[\left\|_{i=1,n}\left(\left\|_{j=1..\tau(t_i)}(t_i, aff_{t_i}^{\,j})\right.\right.\right. > M''$

La construction du graphe des marquages accessibles à partir d'un marquage initial est la base de l'analyse comportementale du système modélisé par un ECATNet. Ce graphe peut être fini ou non (selon que les places sont bornées ou non).

4-3 Exemple

Si l'on considère l'ECATNet $\mathcal{E}_2 = (Spec_2, R_2)$ présenté dans la Figure 1.10, l'ensemble des marquages accessibles est défini par: $\{M_1, M_2, M_3, M_4, M_5, M_6\}$. Le graphe des marquages accessibles est donné dans la Figure 1.11.



	p_1	p_2	p_3
M_1	a⊕ a	∅	∅
M_2	∅	a⊕ a	∅
M_3	∅	∅	a⊕ a
M_4	a	a	∅
M_5	a	∅	a
M_6	∅	a	a

Figure 1.10 : Ensemble des marquages accessibles d'un ECATNet.

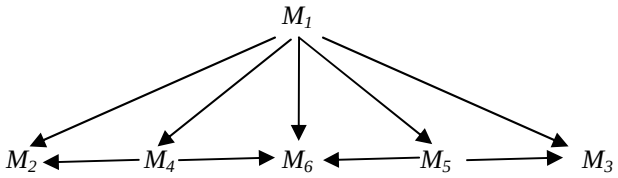


Figure 1.11 : Le graphe des marquages accessibles.

VII- Conclusion

En tant que formalisme de spécification, les ECATNets présentent un certains nombre de qualités importantes :

1- Ils disposent d'une définition *formelle* [BMSB-93][BM-95] : ce caractère formel permet de produire des spécifications exemptes d'ambiguïté; chaque construction des modèles possède une sémantique parfaitement définie.

2- Ils présentent un grand *pouvoir d'expression* :

- les ECATNets sont notamment très bien adaptés à décrire des *comportements* complexes, réactifs ou concurrents.
- Les ECATNets prennent en compte l'aspect « *structure de données* » et non seulement la description de la *structure de contrôle* d'un système et son évolution dynamique comme d'autres types de réseaux de Petri. Ils permettent en effet, de décrire la structure des données manipulées par le système, et la manière dont ces données peuvent influencer le comportement dynamique.

Ainsi, ce formalisme combinant la puissance des spécifications algébrique et celle des réseaux de Petri, permet de spécifier les aspects statiques (structures de données) et les aspects dynamiques (concurrence, indéterminisme) des systèmes.

3- Les ECATNets présentent une certaine souplesse d'utilisation selon que l'on veut privilégier l'aspect statique ou l'aspect dynamique du système. En effet, d'une part les ECATNets peuvent être utilisés comme un formalisme de spécification algébrique (cas où le réseau est vide); et d'autre part comme des réseaux de Petri simples (ECATNet où les places ont toutes le même type spécifié par la sorte *jeton* et généré par l'unique opération $\bullet : \rightarrow jeton$).

4- Ils disposent d'une représentation graphique attrayante, qui accroît la lisibilité et facilite la compréhension des modèles. Cette représentation graphique est également très utile lors de l'exécution interactive des modèles, servant alors de « débogueur » graphique.

Chapitre 2

Les ECATNets Temporels

Sommaire

I- Introduction

Dans le chapitre précédent, nous avons montré la bonne adéquation des ECATNets à la modélisation des systèmes dynamiques. Le formalisme des ECATNets permet de spécifier et d'étudier les propriétés logiques des systèmes en décrivant les relations de *causalité* entre les événements: un événement *a* est la cause de *b*, *a* précède toujours *b*, a et b peuvent s'exécuter au même moment, *a* et *b* sont ordonnés dans le temps. Ces propriétés qui dépendent de l'ordre des évènements ne font aucune référence aux instants où ces évènements arrivent. Le temps est pris en compte de manière *qualitative*.

Cependant, pour une grande classe de systèmes concurrents, le facteur *temps* est une composante primordiale. Ce dernier n'affecte pas seulement les performances du système mais aussi sa validité fonctionnelle. Un procédé peut se retrouver dans un état interdit si un résultat nécessaire à sa bonne évolution est produit trop tôt ou bien trop tard. Nous pouvons citer à titre d'exemples, les procédés agroalimentaires où les durées de préparation et de livraison des produits sont limitées par des impératifs de fraîcheur, ceux de l'industrie chimique où les réactifs utilisés apportent leur effet dans une plage de temps donnée, les systèmes d'informations où une mesure devient obsolète, voire incohérente après une durée donnée.

Afin de satisfaire les attentes nées de l'étude des systèmes où l'intervention du facteur temps doit être prise en compte de manière explicite, nous proposons une extension des ECATNets par les contraintes temporelles. Les ECATNets ainsi étendus sont appelés « *ECATNets temporels* ».

Le présent chapitre est donc consacré à la présentation des ECATNets temporels permettant de prendre en compte l'ensemble des caractéristiques propres aux systèmes dynamiques à contraintes de temps.

II- Définition des ECATNets temporels

Le modèle des ECATNets temporels est inspiré du modèle de Merlin [Mer-74], communément appelé réseau de Petri t-temporel, qui a été conçu à l'origine pour l'étude des problèmes de recouvrement pour les protocoles de communication.
Les ECATNets temporels sont obtenus depuis les ECATNets en associant à chaque transition une contrainte temporelle de type intervalle selon la définition ci-dessous.

1- Définition

Un ECATNet temporel est un couple $\mathcal{E}_t = (\mathcal{E}, IS)$ où

- $\mathcal{E}$: est un ECATNet
- $IS : T \to \mathbb{N} \times \left(\mathbb{N} \cup \{+\infty\}\right)$ est l'application *Intervalle Statique*

  telle que : $\mathbb{N}$ est l'ensemble des nombres entiers positifs ou nuls

  et

  $\forall t \in T \ \left(IS(t) = [a,b] \Rightarrow 0 \le a \le b\right)$

L'application *IS* associe à chaque transition *t* un intervalle temporel *IS(t)=[a,b]* appelé *intervalle statique* (pour le différencier de *l'intervalle dynamique* qui sera défini ultérieurement) de *t* . Cet intervalle décrit la durée de sensibilisation de *t*. Les entiers *a* (noté $\downarrow IS(t)$) et *b* (noté $\uparrow IS(t)$) sont appelés respectivement *date statique de franchissement au plus tôt* de *t* et *date de franchissement au plus tard* de *t.*
Une transition *t* doit rester sensibilisée (i.e. *franchissable* au sens des ECATNets non temporels détaillés au chapitre 1) pendant le délai minimum $\downarrow IS(t)$ avant de pouvoir être franchie, et ne peut rester sensibilisée au-delà du délai maximum $\uparrow IS(t)$ sans être franchie.
Ainsi si une transition *t* est devenue sensibilisée pour la dernière fois à la date θ, alors *t* ne peut être franchie avant la date $\theta + \downarrow IS(t)$ et doit l'être au plus tard à la date $\theta + \uparrow IS(t)$, sauf si le franchissement d'une autre transition a désensibilisé *t* avant que celle-ci ne soit franchie. Le franchissement des transitions est de durée nulle (instantané). Les ECATNets temporels expriment nativement des spécifications «en délais». En explicitant débuts et fins d'actions, ils peuvent aussi exprimer des spécifications «en durées». Leur domaine d'application est donc large.

Remarque :
Les intervalles considérés dans la définition ci-dessus sont des intervalles fermés sur les éléments de $\mathbb{N}$. Cependant il est très simple d'étendre cette classe d'ECATNets temporels en autorisant les contraintes où les intervalles sont ouverts à gauche et/ou à droite. La sémantique des ECATNets temporels s'étend de façon directe à ce type d'intervalles.

2- Représentation graphique

D'un point de vue graphique, le réseau d'un ECATNet temporel est représenté de manière identique aux ECATNets, si ce n'est que l'intervalle statique donné par l'application *IS* pour chaque transition est précisé sur cette représentation selon la Figure 2.1 ci-dessous.

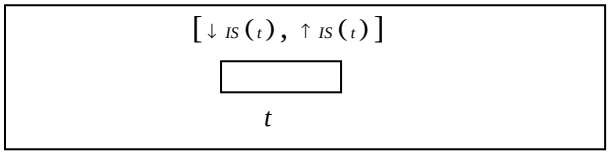


Figure 2.1 : Représentation graphique d'une transition à contraintes temporelles.

3- Exemple

La Figure 2.2 représente un ECATNet temporel dans lequel les transitions ont des contraintes temporelles différentes. La transition t_1 peut être franchie à n'importe quel instant tant qu'elle est sensibilisée (pas de contraintes temporelles proprement dites). La transition t_2 doit être franchie aussitôt qu'elle est sensibilisée (sans un délai d'attente). La transition t_3 ne peut être franchie que pendant l'intervalle de temps $[\theta+1, \theta+5]$ si elle est sensibilisée depuis l'instant θ. Par contre la transition t_4 ne peut être franchie qu'à l'instant exact $\theta+2$ si elle est sensibilisée depuis l'instant θ.

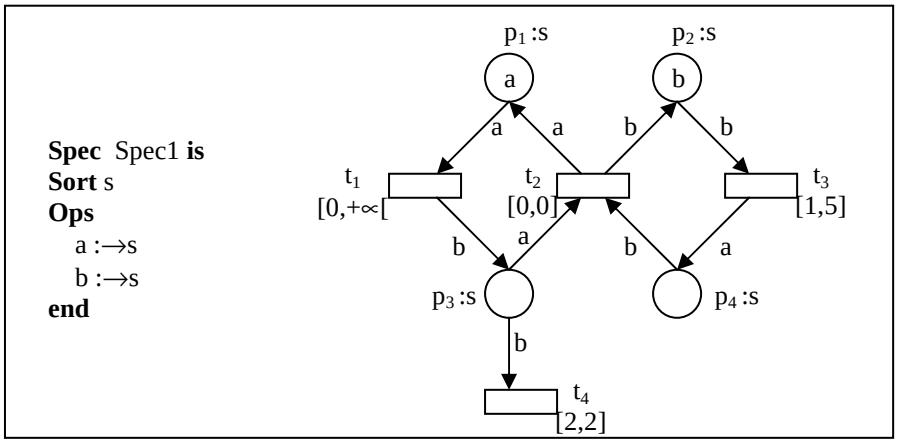


Figure 2.2 : Exemple d'un ECATNet temporel.

III- Comportement d'un ECATNet temporel

1- ECATNet temporel marqué

Un ECATNet temporel marqué est le couple ($\mathcal{E}_t$, M) où :

- $\mathcal{E}_t$ = ($\mathcal{E}$, IS) est un ECATNet temporel
- M est un marquage de $\mathcal{E}$.

Le marquage d'un ECATNet temporel $\mathcal{E}_t$ = ($\mathcal{E}$, IS) est un marquage de l'ECATNet $\mathcal{E}$.

2- Sensibilisation de transitions

Dans un ECATNet temporel on distingue entre la *sensibilisation* et la *franchissabilité* de transitions. La première notion est équivalente à la notion de franchissabilité dans les ECATNets non temporels. Une transition sensibilisée devient franchissable dans un ECATNet temporel, si ses contraintes temporelles sont vérifiées.
On définit la notion de sensibilisation de transitions dans un ECATNet temporel en utilisant la notion de franchissabilité dans les ECATNets non temporels (présentée dans le Chapitre 1), comme suit :

- Une transition t est *sensibilisée* dans un ECATNet temporel $\mathcal{E}_t$= ($\mathcal{E}$, IS) pour le marquage M si et seulement si t est *franchissable* dans l'ECATNet marqué ($\mathcal{E}$, M).

- Une transition t est *multi-sensibilisée n* fois dans un ECATNet temporel $\mathcal{E}_t$= ($\mathcal{E}$, IS) pour le marquage M si et seulement si t est *franchissable n* fois de façon *auto-concurrente* dans l'ECATNet marqué ($\mathcal{E}$, M).

- Un sous-ensemble de transitions $T' \subseteq T$ d'un ECATNet temporel $\mathcal{E}_t$= ($\mathcal{E}$, IS) est *sensibilisé de façon concurrente* pour le marquage M si et seulement si T' est *franchissable de façon concurrente* dans l'ECATNet marqué ($\mathcal{E}$, M).

- Un multi-ensemble de transitions $\tau : T \to \mathbb{N}$ est *sensibilisé* dans un ECATNet temporel $\mathcal{E}_t$= ($\mathcal{E}$, IS) pour le marquage M si et seulement si il est *franchissable* dans l'ECATNet $\mathcal{E}$ pour un marquage M.

Exemple : Dans l'ECATNet temporel de la Figure 2.2, Chacune des transitions t_1 et t_3 sont sensibilisées pour le marquage initial. De plus, le sous-ensemble de transitions $\{t_1, t_3\}$ est sensibilisé de façon concurrente.

3- Franchissabilité de transitions

Soit $\mathcal{E}_t = (\mathcal{E}, IS)$ un ECATNet temporel marqué par M. La franchissabilité des transitions de $\mathcal{E}_t$ est définie par les points suivants :

- Une transition t de $\mathcal{E}_t$ est *franchissable* pour M et une affectation aff_t de $ctx(t)$ si et seulement si :
  i) t est restée sensibilisée pour M et aff_t *depuis au moins* $\downarrow IS(t)$ unités de temps, et
  ii) t est restée sensibilisée pour M et aff_t depuis *pas plus* de $\uparrow IS(t)$ unités de temps.

- Une transition t de $\mathcal{E}_t$ est *franchissable n fois de façon auto-concurente* pour M et la famille d'affectations $\left(aff_t^i\right)_{i=1..n}$ si et seulement si :
  i) t est restée multi-sensibilisée n fois pour M et $\left(aff_t^i\right)_{i=1..n}$ *depuis au moins* $\downarrow IS(t)$ unités de temps, et
  ii) t est restée multi-sensiblisée n fois pour M et $\left(aff_t^i\right)_{i=1..n}$ depuis *pas plus* de $\uparrow IS(t)$ unités de temps.

- Un sous-ensemble de transitions T' de $\mathcal{E}_t$ ($T' \subseteq T$) est franchissable *de façon concurrente* pour le marquage M et la famille d'affectations $\left(aff_t\right)_{t\in T'}$ si et seulement si :
  i) T' est resté sensibilisé pour M et $\left(aff_t\right)_{t\in T'}$ *depuis au moins* $\boldsymbol{max}\left(\left(\downarrow IS(t)\right)_{t\in T'}\right)$ unités de temps, et
  ii) T' est resté sensibilisé pour M et $\left(aff_t\right)_{t\in T'}$ *depuis pas plus* de $\boldsymbol{min}\left(\left(\uparrow IS(t)\right)_{t\in T'}\right)$ unités de temps.

- Un multi-ensemble de transitions $\tau : T \to \mathbb{N}$ de $\mathcal{E}_t$ est franchissable *de façon concurrente* pour le marquage M et les d'affectations $\left(\left(aff_t^i\right)_{i=1..\tau(t)}\right)_{t\in T}$ si et seulement si :
  i) τ est resté sensibilisé pour M et $\left(\left(aff_t^i\right)_{i=1..\tau(t)}\right)_{t\in T}$ *depuis au moins* $\boldsymbol{max}\left(\left(\downarrow IS(t)\right)_{t\in T/\tau(t)\neq 0}\right)$ unités de temps, et
  ii) τ est resté sensibilisé pour M et $\left(\left(aff_t^i\right)_{i=1..\tau(t)}\right)_{t\in T}$ *depuis pas plus* de $\boldsymbol{min}\left(\left(\uparrow IS(t)\right)_{t\in T/\tau(t)\neq 0}\right)$ unités de temps.

Exemple : Soit $\theta = 0$ l'instant initial de l'ECATNet temporel de la Figure 2.2, les transitions t_1 et t_3 sont sensibilisées pour le marquage initial mais seule t_1 est franchissable. A l'instant $\theta = 4$, les deux transitions t_1 et t_3 sont franchissables. A l'instant $\theta = 6$ la transition t_1 est franchissable mais t_3 n'est plus franchissable. L'ensemble $\{t_1,t_3\}$ est franchissable à $\theta = 1$ mais non franchissable à l'instant $\theta = 6$.

4- Etat d'un ECATNet temporel

4.1 Notion de pas (« step »)

L'état d'un ECATNet temporel est basé sur la notion de *pas (« step »)*. Informellement un *pas* correspond à une transition entre deux états. Cette transition s'effectue généralement en franchissant un multi-ensemble de transitions.

- Un *pas élémentaire* est le couple $\widetilde{T}_i = (t, aff_t)$ où t est une transition de l'ECATNet temporel $\mathcal{E}_t = (\mathcal{E}, IS)$ et aff_t est une affectation consistante du contexte de t. Un pas élémentaire $\widetilde{T}_i = (t, aff_t)$ est sensibilisé pour M si et seulement si t est franchissable pour M et aff_t dans $\mathcal{E}$.

- Un *pas* $\tilde{T}_i = \uplus_{t \in T} \left\{ \left(t, aff_t^{\,i} \right)_{i=1..\tau_i(t)} \right\}$ est un multi-ensemble de *pas élémentaires* où $\tau_i(t)$ est le taux d'autoconcurrence de t dans le pas $\tilde{T}_i$ *et* $\uplus$ est l'opération union de multi-ensemble de pas. Le pas $\tilde{T}_i$ est sensibilisé pour M dans $\mathcal{E}_t = (\mathcal{E}, IS)$ si est seulement si le multi-ensemble τ_i est franchissable dans $\mathcal{E}$ pour M et la famille des affectations $\left(\left(t, aff_t^{\,i} \right)_{i=1..\tau_i(t)} \right)_{t \in T}$.

4.2 Etat initial d'un ECATNet temporel

L'état initial d'un ECATNet temporel $\mathcal{E}_t$ est le triplet $\left(M_0, \tilde{T}, \widetilde{IS} \right)$ où :

- M_0 est le marquage initial de $\mathcal{E}_t$
- $\tilde{T} = \left\{ \tilde{T}_1, \tilde{T}_2, ..., \tilde{T}_n \right\}$ est l'ensemble de tous les *pas* (« *steps* ») sensibilisés dans $(\mathcal{E}_t, M_0)$.
- $\widetilde{IS}$ est l'application qui associe un intervalle statique de tir à chaque *pas* de $\tilde{T}$.

$$\widetilde{IS} : \tilde{T} \to \mathbb{N} \times (\mathbb{N} \cup \{+\infty\})$$

$$\forall \tilde{T}_i \in \tilde{T} \quad \widetilde{IS}\left(\tilde{T}_i\right) = \left[\downarrow \widetilde{IS}\left(\tilde{T}_i\right), \uparrow \widetilde{IS}\left(\tilde{T}_i\right) \right] \text{ tel que:}$$

$$\downarrow \widetilde{IS}\left(\tilde{T}_i\right) = \mathbf{max}\left(\downarrow IS(t) / \tau_i(t) \neq 0 \right)$$

$$\uparrow \widetilde{IS}\left(\tilde{T}_i\right) = \mathbf{min}\left(\uparrow IS(t) / \tau_i(t) \neq 0 \right)$$

4.3 Etat dynamique

Un état d'un ECATNet temporel $\mathcal{E}_t$ est un triplet $\left(M, \tilde{T}, \tilde{I}\right)$ dans lequel :

- M est un marquage de $\mathcal{E}_t$
- $\tilde{T} = \left\{\tilde{T}_1, \tilde{T}_2, ..., \tilde{T}_n\right\}$ est l'ensemble de tous les *pas* (« *steps* ») sensibilisés dans ($\mathcal{E}_t$, *M).*
- $\tilde{I} : \tilde{T} \rightarrow \mathbb{N} \times \left(\mathbb{N} \cup \{+\infty\}\right)$ est l'application qui associe un intervalle de sensibilisation à chaque *pas* de $\tilde{T}$.

L'intervalle de sensibilisation $\tilde{I}$ considéré sera alors qualifié d'intervalle *dynamique*. En effet, ce dernier peut différer de l'intervalle statique $\widetilde{IS}$, comme l'illustre la Figure 2.3. A l'état initial, seul la transition t_1 *est* sensibilisée. L'intervalle dynamique de t_1 correspond à son intervalle statique (i.e. [1,5]). Supposons maintenant qu'après θ unités de temps ($1 \leq \theta \leq 5$) le terme *a* soit toujours contenue dans p_1, l'intervalle dynamique associé à t_1 correspondra alors à l'intervalle défini par : [max(0, 1 - θ), 5 - θ].

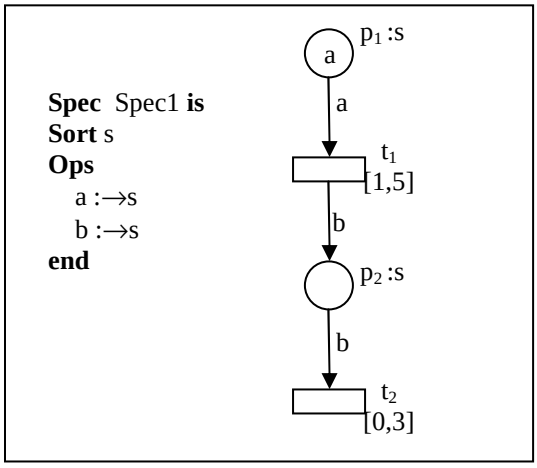


Figure 2.3 : Intervalle statique/ dynamique d'une transition.

5- Franchissement de transitions

L'état d'un ECATNet temporel est donc défini par le marquage courant, l'ensemble des pas sensibilisés, et par la valeur des intervalles temporels associées aux pas sensibilisés. Il y a deux types de changements d'états. Le premier est lié à l'écoulement du temps lorsque aucun pas n'est franchi. Si le temps θ s'est écoulé, alors sans changer le marquage courant il faut supprimer θ de chaque borne des l'intervalles temporels associés aux pas sensibilisés. Le deuxième type de changement d'état correspond au franchissement d'un pas. Comme le franchissement est instantané, le temps (en tant que variable continue n'évolue pas). A temps constant, on modifie le marquage, on construit le nouvel ensemble des pas sensibilisés. Pour les pas qui étaient sensibilisés avant le franchissement, on laisse la valeur de l'intervalle inchangé. Pour chaque pas $\tilde{T}_i$ nouvellement sensibilisé, on initialise l'intervalle temporel de $\tilde{T}_i$ de telle sorte que la date de franchissement au plus tôt soit le maximum des dates de franchissement au plus tôt des transitions de $\tilde{T}_i$, et la date de franchissement au plus tard soit le minimum des dates de franchissement au plus tard des transitions de $\tilde{T}_i$.

De manière générale, le franchissement d'un pas $\tilde{T}_i$, à une date relative θ, depuis un état $(M, \tilde{T}, \tilde{I})$ donne le nouvel état $(M', \tilde{T}', \tilde{I}')$ déterminé par :

i) Le marquage *M'* est défini par :

$\forall p \in P \quad M'(p) = M(p) \oplus_{\sigma(p)} M_{cr}(p) -_{\sigma(p)} M_{dt}(p)$

(comme dans les ECATNets non temporels ref. Chapitre.1)

ii) L'ensemble $\tilde{T}'$ est défini par : $\tilde{T}' = \{\tilde{T}_k / \tilde{T}_k$ sensibilisé dans $M'\}$

iii) L'application $\tilde{I}'$ est définie par :

Pour chaque $\tilde{T}_k \in \tilde{T}'$

- Si $\tilde{T}_k$ est sensibilisé dans M *et* $\tilde{T}_k \neq \tilde{T}_i$ alors :

$$\tilde{I}'(\tilde{T}_k) = [\downarrow \tilde{I}'(\tilde{T}_k), \uparrow \tilde{I}'(\tilde{T}_k)] \text{ tel que: } \begin{pmatrix} \downarrow \tilde{I}'(\tilde{T}_k) = \mathbf{max}(0, \downarrow \tilde{I}(\tilde{T}_k) - \theta) \\ \uparrow \tilde{I}'(\tilde{T}_k) = \uparrow \tilde{I}(\tilde{T}_k) - \theta \end{pmatrix}$$

- Si ($\tilde{T}_k$ est non sensibilisé dans *M) ou (*$\tilde{T}_k$ est sensibilisé dans M *et* $\tilde{T}_k = \tilde{T}_i$ *)* alors :

$$\tilde{I}'(\tilde{T}_k) = [\downarrow \tilde{I}'(\tilde{T}_k), \uparrow \tilde{I}'(\tilde{T}_k)] \text{ tel que: } \begin{pmatrix} \downarrow \tilde{I}'(\tilde{T}_k) = \mathbf{max}(\downarrow IS(t) / \tau_k(t) \neq 0) \\ \uparrow \tilde{I}'(\tilde{T}_k) = \mathbf{min}(\uparrow IS(t) / \tau_k(t) \neq 0) \end{pmatrix}$$

La règle de franchissement ci-dessus définit une relation d'accessibilité sur l'ensemble des états d'un ECATNet temporel. Le fonctionnement d'un ECATNet temporel peut être caractérisé par l'ensemble des états accessibles depuis son état initial. Notons que le concept d'état présenté associe un intervalle de temps à chaque *pas* franchissable (et non pas un intervalle à chaque transition franchissable). Ceci permet d'une part, de considérer le franchissement auto-concurrent d'une même transition multi-sensibilisée à des instants différents, et permet d'autre part de franchir au même instant un multi-ensemble de transitions qui ont été sensibilisées à des dates différentes.

Notation :

$(M, \tilde{T}, \tilde{I}) \xrightarrow{\tilde{T}_i @ \theta} (M', \tilde{T}', \tilde{I}')$: Le franchissement d'un pas $\tilde{T}_i$, à une date relative θ depuis un état $(M, \tilde{T}, \tilde{I})$ donne le nouvel état $(M', \tilde{T}', \tilde{I}')$.

6- Classes des états d'un ECATNet temporel

L'ensemble des états d'un ECATNet temporel est infini, même si l'ECATNet sous-jacent est borné pour le marquage initial choisi. En effet, les pas pouvant être franchis à tout instant dans leur intervalle de franchissement (et à cause de l'évolution continue du temps), les états admettent en général une infinité de successeurs. Toutefois, l'évolution du temps sans qu'un pas ne soit franchi n'a pas d'influence sur les évolutions futures (du moins lorsque l'on ne sort pas des intervalles de franchissement pour les pas franchissables). L'analyse des ECATNets temporels est donc fondée sur une notion de classe d'états qui regroupe tous les états ayant même marquage, même ensemble de pas sensibilisés, et n'étant différents que par les intervalles temporels associés aux pas sensibilisés. Tous les états regroupés dans une classe doivent être tels que l'évolution future de l'ECATNet temporel soit la même.

Exemple :
A titre d'illustration, construisons quelques classes de l'ECATNet temporel représenté par la Figure 2.2.

L'état initial est $E_0 = \left(M_0, \tilde{T}_0, \tilde{I}_0\right)$

$$\begin{cases} M_0(p_1) = a \;\; M_0(p_2) = b \;\; M_0(p_3) = \varnothing \;\; M_0(p_4) = \varnothing \\ \tilde{T}_0 = \{t_1, t_3, t_1 \uplus t_3\} \\ \tilde{I}_0(t_1) = [0, +\infty[\;\; \tilde{I}_0(t_3) = [1,5] \;\; \tilde{I}_0(t_1 \uplus t_3) = [1,5] \end{cases}$$

Le franchissement de t_1 depuis E_0 à une date $\theta_1 \in [0,1[$ mène en $E_1 = \left(M_1, \tilde{T}_1, \tilde{I}_1\right)$ tel que :

$$\begin{cases} M_1(p_1) = \varnothing \;\; M_1(p_2) = b \;\; M_1(p_3) = b \;\; M_1(p_4) = \varnothing \\ \tilde{T}_1 = \{t_3, t_4, t_3 \uplus t_4\} \\ \tilde{I}_1(t_3) = [1-\theta_1, 5-\theta_1] \;\; \tilde{I}_1(t_4) = [2,2] \;\; \tilde{I}_1(t_3 \uplus t_4) = [2,2] \end{cases}$$

Le franchissement de t_4 depuis E_1 à la date $\theta_2 = 2$ mène en $E_2 = \left(M_2, \tilde{T}_2, \tilde{I}_2\right)$ où :

$$\begin{cases} M_2(p_1) = \varnothing \;\; M_2(p_2) = b \;\; M_2(p_3) = \varnothing \;\; M_2(p_4) = \varnothing \\ \tilde{T}_2 = \{t_3\} \\ \tilde{I}_2(t_3) = [0, 3-\theta_1] \end{cases}$$

Le franchissement de t_3 depuis E_2 à l'instant θ_3 tel que $0 \le \theta_3 \le 3-\theta_1$ donne l'état $E_3 = \left(M_3, \tilde{T}_3, \tilde{I}_3\right)$

$$\begin{cases} M_3(p_1) = \varnothing \;\; M_3(p_2) = \varnothing \;\; M_3(p_3) = \varnothing \;\; M_3(p_4) = a \\ \tilde{T}_3 = \varnothing \end{cases}$$

Le franchissement de t_3 depuis E_1 à une date θ_4 tel que $1-\theta_1 \le \theta_4 \le 5-\theta_1$ donne l'état $E_4 = \left(M_4, \tilde{T}_4, \tilde{I}_4\right)$

$$\begin{cases} M_4(p_1) = \varnothing \;\; M_4(p_2) = \varnothing \;\; M_4(p_3) = b \;\; M_4(p_4) = a \\ \tilde{T}_4 = \{t_4\} \\ \tilde{I}_4(t_4) = [2-\theta_4, 2-\theta_4] \end{cases}$$

Le franchissement de t_4 depuis E_4 à l'instant θ_5 tel que $2-\theta_4 \le \theta_5 \le 2-\theta_4$ donne l'état E_3.

Les séquences de franchissement présentées ci-dessus correspondent à un sous-graphe du graphe de classes des états de l'ECATNet temporel de la Figure 2.2. Ce sous-graphe est schématisé par la Figure 2.3.

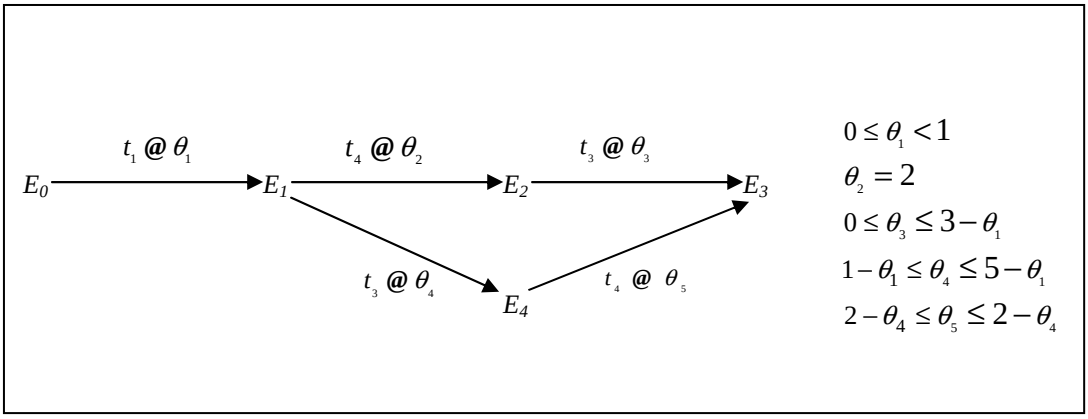


Figure2.3 : Exemple de sous-graphe d'un graphe des classes des états d'un ECATNet temporel.

IV- Conclusion

Les ECATNets temporels constituent une méthode de spécification permettant de tenir compte de manière efficace des différentes fonctionnalités associées au système, ainsi que de ses caractéristiques temporelles. Ils généralisent les ECATNets en permettant la prise en compte du facteur temps comme représentant une composante explicite du système spécifié. Les ECATNets non temporels (sans explicitation du temps) correspondent simplement au cas où tous les intervalles de sensibilisation sont égaux à $[0, +\infty[$.

Ainsi, dans ce chapitre nous avons exposé de manière succincte, les caractéristiques principales des ECATNets temporels. Notre propos étant de fournir pour le langage CIRTA un formalisme de base qui généralise les ECATNets pour permettre la spécification d'une plus large gamme de systèmes.

Chapitre 3

Caractéristiques principales d'un langage de spécification

Sommaire

I- Introduction

Dans le domaine des spécifications formelles, plusieurs langages ont été définis. Chacun d'eux ayant des caractéristiques et des objectifs propres. Une liste non exhaustive de tels langages est présentée dans [BOW-07]. L'objet de ce chapitre n'est pas d'établir une étude comparative des langages de spécification formelle existants, ni d'élaborer un état de l'art dans ce domaine, mais plutôt de définir les bases sur lesquelles une telle étude pourrait être menée.

La conception d'un langage de spécification est fondée sur le choix de certains facteurs déterminants tels que le choix du formalisme sur lequel se base le langage, le type de syntaxe, le type de sémantique, le niveau d'expression de la sémantique, la modularité, les primitives du langage et l'exécutabilité du langage. Les différents choix faits lors de cette conception caractérisent le langage de spécification et permettent de le situer ou de le comparer avec d'autres langages.

Il est donc nécessaire de présenter ces différents facteurs et de préciser les différents choix faits pour le langage CIRTA.

II- Formalisme sous-jacent du langage de spécification

Il existe trois classes de formalismes de spécification [WIN-89] : Les formalismes orientés propriétés, les formalismes orientés modèles et les formalismes hybrides. Ces classes engendrent trois classes fondamentales de langages de spécification (la classe d'un langage étant définie par celle du formalisme sous-jacent correspondant).

1- Langages de spécification orientés propriétés

Les langages de spécification orientés propriétés ont la particularité de permettre la description des systèmes à l'aide de formules logiques. On définit le comportement du système de manière indirecte en établissant un ensemble de propriétés, habituellement sous la forme d'axiomes que le système doit satisfaire. On y trouve donc tous les langages fondés sur la logique mathématique tels que les langages de spécification algébriques ou les langages basés sur la logique temporelle. Les langages de spécification algébriques sont bien adaptés à la spécification des structures de données et disposent de nombreuses références tant théoriques que pratiques [GUT-75][GTW-77][GTW-78][KL-83]. Parmi ce type de langages on peut citer CLEAR [BG-77], ASL [WIR-83] [WIR-86], ACT-ONE [EM-85], LARCH [GH-83][WIN-87], OBJ2 [GT-79][FUT-86], LOOK [ETL-84], OBSCURE [LL-87], AXIS [COL-88], LPG [BDE-87], CASL [ABK-02] [MOS-00]. Ces langages permettent de spécifier les données et en partie leurs manipulations à travers la spécification de fonctions opérant sur ces données. Les aspects comportementaux, au sens ordonnancement des manipulations, sont difficilement spécifiables à l'aide des langages de spécification algébriques. Les langages basés sur la logique temporelle permettent d'y remédier dans une certaine mesure: Ils permettent l'écriture de contraintes temporelles sur les traitements du système.

2- Langages de spécification orientés modèles

Les langages de spécification orientés modèles permettent de décrire un système à l'aide d'objets prédéfinis assemblés entre eux à l'aide d'opérateurs de composition. Ils offrent l'avantage de proposer un ensemble de briques de base possédant une sémantique implicite. C'est dans le cadre de la spécification des comportements du système que cette classe de langages est la plus utilisée. En effet, les notions d'état et d'événement constituent deux concepts auxquels il est utile d'associer des objets prédéfinis. Les langages comme VDM [BJ-78] [JON-90] [DAW-91], Z [SPI-98] [WOR-96] [DW-96], B[ABR-96a] [ABR-96b] [ABR-97a] [ABR-97b]. CCS [MIL-89], CSP[HOA-85], basés sur les systèmes de transitions, sur les algèbres de processus, les machine à états, les automates, ou sur les réseaux de Petri font partie des langages de spécifications orientés modèles les plus développés en informatique. En particulier, les réseaux de Petri offrent la possibilité de modéliser un système à l'aide de *places* (représentant l'état du système), de *transitions* (représentant les évènements du système) et de *pré* et *pos*t conditions permettant d'exprimer des contraintes sur l'état du système avant et après l'exécution d'un événement.

3- Langages de spécification hybrides

Les langages de spécification hybrides constituent la troisième classe. Ce sont des langages permettant de décrire un système à l'aide des deux méthodes précitées. On rencontre dans cette classe des langages tels que LOTOS [BB-87][ISO-89] et COOPN [BG-91a][BG-91b][BBG-97].

- **Formalisme sous-jacent du langage CIRTA**

Le langage CIRTA est un langage de spécification hybride qui fondé sur le formalisme des ECATNets combinant les spécifications algébriques et les réseaux de Petri. Il présente ainsi une grande souplesse d'utilisation selon que l'on veut privilégier l'aspect statique ou l'aspect dynamique du système à spécifier.

III- Type de syntaxe du langage de spécification

L'effort majeur dans la conception de la syntaxe d'un langage de spécification (ou autres types de langages) est de définir un ensemble minimal de primitives (ou instructions) suffisamment flexible, et de fixer quelques opérateurs simples mais fondamentaux, permettant à l'utilisateur de décrire, de transformer, et de composer différentes parties d'une spécification. La syntaxe d'un langage de spécification précise alors les règles d'écriture des spécifications qui peut être présentée sous forme d'un texte, sous forme d'un graphe ou une combinaison des deux. On parle alors de langages à syntaxe textuelle, à syntaxe graphique ou à syntaxe semi-graphique.

1- Syntaxe textuelle

Les langages de spécification orientés propriétés ont généralement une syntaxe décrite par du texte. La spécification est divisée en sections. Chacune de ces sections étant introduite par un mot clé spécifique. On trouve parmi ces langages les langages de spécification algébriques comme PLUSS [BID-89], ACT-ONE [EM-85], CASL [ABK-02] et OBJ [GT-79][FUT-86].

2- Syntaxe graphique

Ce sont des langages offrant un ensemble de briques de base possédant une sémantique implicite. Parmi ces langages on trouve les réseaux de Petri (qui sont considérés comme un langage graphique), les langages de description graphique IDEF/SADT et les diagrammes Yourdon [WM-85] [YC-78]. Bien que ces langages comportent des notations textuelles, celles-ci sont minimes. Elles servent en général à spécifier l'identificateur d'une spécification ou de l'un de ses composants (une place, une transition …etc.).

3- Syntaxe semi-graphique

Dans ce type de langage, la spécification du système est décrite par une partie textuelle et une partie graphique. Ce type de syntaxe permet de représenter avec concision la structure du système à l'aide d'un graphe, puis de décrire des propriétés à l'aide d'une partie textuelle. Ces langages présentent une syntaxe souple car selon le type du système à spécifier et les outils disponibles (éditeur graphique) et selon les tendances de l'utilisateur, la spécification comportera une partie texte plus ou moins importante par rapport à la partie graphique.
Parmi les langages ayant une syntaxe semi-graphique, on trouve les langages SEGRAS [KRA-87], LOTOS[BB-87][ISO-89], COOPN [BG-91a] [BG-91b] [BBG-97] .

- **Type de syntaxe du langage CIRTA**

La syntaxe du langage CIRTA est *semi-graphique*. Ceci permet d'une part de préserver la représentation graphique connue dans les réseaux de Petri, et d'autre part la description aisée par du texte des propriétés relatives aux structures de données. Ceci n'exclue pas le fait qu'une spécification écrite dans CIRTA, puisse être exclusivement textuelle (i.e. le réseau R d'un ECATNet peut être décrit sous forme de texte grâce à des primitives prédéfinies dans le langage CIRTA).

Il convient de noter que la syntaxe de CIRTA n'est pas une syntaxe entièrement prédéfinie. En effet, une des caractéristiques intéressantes de CIRTA (comme dans les langages de spécification algébriques) est que l'utilisateur a la possibilité de définir une syntaxe (via une signature) et de décrire les propriétés statiques du système spécifié (à l'aide des équations) en utilisant cette syntaxe. On note dans CIRTA une grande liberté dans le choix de la forme et profils des opérations. Aucune restriction n'est imposée sur la syntaxe introduite par l'utilisateur. En effet, dans CIRTA on autorise la définition d'opérations préfixées, infixées, postfixées, mixfixées (l'opérateur est distribué par rapport aux arguments) et même des opérations anonymes (opérations sans symbole visible) comme par exemple la *coersion*. Le langage CIRTA supporte également le polymorphisme ("overloading") des opérateurs.

Le langage CIRTA est conçu pour la commodité de description et de manipulation des ECATNets. Il offre des notations syntaxiques concises permettant:

- Une description statique des systèmes et de leur comportement dynamique: conditions et effets de déclenchements des transitions.
- Des transformations simples telles que l'ajout et la fusion d'éléments d'un réseau.
- La composition par fusion d'éléments de réseaux déjà définis.
- La généricité qui permet, à partir d'un réseau unique, de générer des instances potentiellement différentes.
- La spécification des interactions éventuelles entre deux modules de spécification (synchronisation).
- La création dynamique des instances de réseaux.
- Les transformations des spécifications par renommage et contrôle de visibilité.

IV- Type de sémantique du langage de spécification

Il est courant de distinguer les langages de spécification selon qu'à une spécification est associé *un modèle* particulier (*modèle initial* d'une certaine classe de modèles) ou une *classe de modèles*. On parle alors de *sémantique initiale*, ou sémantique de type *classe de modèles*.

1- Sémantique initiale

A chaque spécification est associé un modèle particulier: le modèle *initial* d'une certaine classe de modèles. Bien que cette sémantique constitue le cadre formel approprié pour prouver la *consistance* et la *complétude suffisante* d'une spécification [BER-87], et qu'elle permet généralement d'avoir des spécifications exécutables; l'existence du modèle initiale n'est pas assurée que sous certaines restrictions [BID-89].

2- Sémantique de type classe de modèles

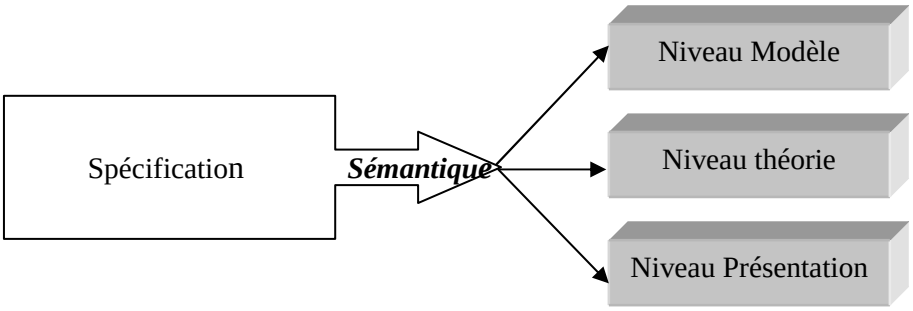
Dans ce cas, on associe à chaque spécification une classe de modèles (non nécessairement isomorphes). Cette sémantique présente l'avantage de n'imposer aucune restriction sur les spécifications et facilite l'expression de la sémantique des primitives de structuration des spécifications. Cependant, elle ne permet pas d'associer à une spécification un modèle de référence particulier et ne favorise pas ainsi l'exécutabilité des spécifications.

- **Type de sémantique pour le langage CIRTA**

Les avantages et inconvénients respectifs de chacune des approches et le choix de l'une ou l'autre ne doivent pas faire oublier qu'à l'origine le rôle principal d'un langage de spécification est de permettre une description *abstraite*. Cette description est abstraite dans le sens où elle ne précise que *les propriétés minimales* requises de toutes les réalisations effectives de la spécification. Dans le langage CIRTA, on cherche à refléter ce principe fondamental au niveau sémantique même du langage, en associant à chaque spécification une *classe de modèles* correspondants à toutes les réalisations "correctes" possibles de la spécification en question.

V- Niveau d'expression de la sémantique du langage

Définir la sémantique d'un langage de spécification consiste à préciser quel objet mathématique est associé à une spécification donnée. Compte tenu du niveau d'abstraction des langages de spécification, leur sémantique est en général décrite dans un style "dénotationnel". Dans le cas des ECATNets, la sémantique du langage de spécification CIRTA est dénotationnelle dans la mesure où elle associe à toute spécification (syntaxiquement correcte) un objet mathématique qui est *un système de réécriture*.
Le fait qu'un modèle de calcul soit associé à l'objet mathématique sémantique de la spécification (ici réécriture de termes) ne signifie pas que la sémantique est opérationnelle, mais relève plutôt, à notre avis, des aspects "exécutables" du langage de spécification. Il faut noter qu'il n'est pas toujours évident de distinguer entre sémantique opérationnelle et sémantique dénotationnelle dans les langages basés sur les spécifications algébriques [KKM-88].
Il existe trois niveaux d'expressions possibles pour la sémantique d'un langage de spécification: le *niveau présentation*, le *niveau théorie* et le *niveau modèle*.
Ces trois niveaux ne sont pas équivalents puisque le niveau modèle est le plus général (plus puissant) que le niveau théorie; lui-même plus général que le niveau présentation.



1- Niveau présentation

La sémantique d'un langage de spécification est définie comme une application du niveau textuel vers le niveau présentation.
La sémantique d'une spécification est donnée au moyen de trois applications:

- Application *Sig* : qui associe à une spécification structurée $\hat{S}$ une signature *Sig*($\hat{S}$) (qui est dans notre cas la signature de l'ECATNet plat correspondant).
- Application *Ax*: qui associe à $\hat{S}$ l'ensemble des axiomes *Ax*($\hat{S}$)
- Application *Net*: qui associe à $\hat{S}$ le réseau plat correspondant *Net*($\hat{S}$)

Parmi les langages utilisant ce niveau d'expression de la sémantique on trouve les réseaux de Petri colorés hiérarchique [Jen-92], où la sémantique des primitives est donnée en présentant le réseau plat (non hiérarchique) correspondant.

2- Niveau théorie

La sémantique d'un langage de spécification est décrite dans ce cas par une application du niveau textuel vers le niveau théorie.
La sémantique d'une spécification est définie par deux applications:

- Application *Sig* : qui associe à une spécification $\hat{S}$ une signature *Sig*($\hat{S}$) (dans notre cas il s'agit de la signature de l'ECATNet plat correspondant).
- Application *Th*: qui associe à $\hat{S}$ une théorie *Th*($\hat{S}$)

Un exemple typique de langages qui utilisent ce niveau d'expression de la sémantique est le langage CLEAR [BG-77].

3- Niveau modèle

La sémantique d'un langage de spécification est définie par une application du niveau textuel vers le niveau modèle.
La sémantique d'une spécification est définie par deux applications:

- Application *Sig* : qui associe à une spécification $\hat{S}$ une signature *Sig*($\hat{S}$) (la signature de l'ECATNet plat correspondant).
- Application *Mod*: qui associe à $\hat{S}$ la classe des modèles *Mod*($\hat{S}$).

Chacune des applications *Ax, Th, Mod* sont définies en fonction de *Sig*.

Il est évident que les trois niveaux d'expression de la sémantique ne sont pas indépendants:
- A une présentation on peut associer les théories engendrées par l'ensemble des axiomes et l'ensemble transitions
- A une théorie on associe la classe de toutes les catégories qui satisfont cette théorie.

- **Niveau d'expression de la sémantique du langage CIRTA**

La sémantique de CIRTA est exprimée au niveau modèle. L'intérêt d'une sémantique de type classe de modèles est de permettre d'associer à une spécification une classe de modèles non nécessairement isomorphes à la spécification.

VI- Propriétés explicites et implicites de la sémantique d'un langage

Les propriétés vérifiées par les modèles d'une spécification peuvent être des propriétés explicites ou des propriétés implicites. Les premières sont des conséquences directes des axiomes (comportementaux ou non) de la spécification. Le reste des propriétés dites "implicites" sont des propriétés supplémentaires vérifiées par les modèles de la spécification (telles que par exemple la définition de la sorte multi-ensemble).
Il est évident que le niveau d'expression de la sémantique influe sur la possibilité plus au moins grande de combiner dans la sémantique d'une spécification les propriétés explicites et implicites.
En effet, au niveau présentation, on ne peut pas proprement parler de propriétés implicites dans la sémantique (par exemple la sorte multi-ensemble doit être explicitement spécifiée); alors qu'au niveau modèle les possibilités d'inclure des propriétés implicites sont assez variées.

- **Quelques propriétés implicites de la sémantique du langage CIRTA**

Dans le langage CIRTA on essaye de libérer l'utilisateur d'un maximum de contraintes. Des aspects implicites sont inclus dans la sémantique même du langage tels que:

- La définition de spécification algébrique multi-ensemble d'une sorte donnée: la définition d'une place *p* de sorte *s* implique la définition implicite de la sorte m_s multi-ensemble de la sorte *s* et de toutes les opérations et équations correspondantes.
- La spécification BOOL est implicitement définie dans chaque spécification pour permettre la définition des conditions des transitions *TC* et l'écriture des équations conditionnelles.
- La spécification des entiers naturels NAT est implicitement définie dans chaque spécification pour permettre la définition des capacités des places *Cap* et des intervalles temporels *IS*.

VII- Exécutablité d'un langage de spécification

En général, les spécifications ne sont jamais exécutables par elles même, mais elles peuvent être rendues exécutables en effectuant un certain nombre de transformations sur les axiomes (comportementaux ou non). La version exécutable ainsi obtenue doit pouvoir être mise en correspondance avec un des modèles associés à la spécification afin d'assurer la "correction" des transformations effectuées. Ainsi, il est évident de chercher à exécuter un ECATNet en utilisant le système de réécriture qui lui est associé. Un tel système de réécriture est formé d'une part des axiomes de la spécification algébrique considérés comme équations orientées de façon arbitraire (en général de gauche à droite); et d'autre part d'un ensemble de règles de réécriture correspondant aux différentes transitions de l'ECATNet.
Bien que cette approche semble intuitive, il convient de noter un certain nombre de points importants relatifs à l'utilisation d'une telle méthode pour l'exécution des spécifications dans le cadre des ECATNets:

a) Il n'est pas toujours possible de considérer les axiomes d'une spécification algébrique comme des règles de réécriture car certains axiomes *ne peuvent pas être orientés* tels que les axiomes relatifs aux propriétés de commutativité et d'associativité.
b) Il faut souligner le fait que *ce n'est pas la spécification en question qui est proprement exécutée* mais plutôt le système de réécriture supposé "équivalent".
c) La traduction d'un ECATNet en un système de réécriture "équivalent" est faite au niveau présentation uniquement. La structuration de la spécification en modules n'est pas considérée.

Dans le langage CIRTA, on n'impose pas aux spécifications d'être exécutables. Ce qui signifie d'une part qu'*aucune restriction sur l'écriture des axiomes* n'est imposée; et d'autre part qu'une spécification ne pourra pas être directement considérée comme fournissant un prototype du système à réaliser car il semble judicieux de séparer *spécification* et *prototypage*.

VIII- Modularité d'un langage de spécification

1- Modularité des spécifications

Tout langage de spécification raisonnable doit être modulaire. La spécification d'un système de taille réaliste sera constituée d'une collection de modules. Un module de spécification à l'aide des ECATNets contiendra des déclarations de *sortes*, des déclarations d'*opérations*, des *axiomes* et d'un *réseau*. Ces quatre types d'entités constituent le noyau du module de spécification. Le module contient en plus des *directives* spécifiant comment ce module doit être assemblé avec d'autres modules pour former la spécification globale du système.
Les primitives de construction permettant d'assembler les modules entre eux varient d'un langage de spécification à un autre. Cependant la plupart d'entre eux recouvrent au moins la notion d'*enrichissement* ou de *composition*.
Ainsi la structuration d'une spécification en une collection de modules correspond à une structure de graphe (où les nœuds représentent les modules et les arcs correspondent aux directives d'assemblage). Cette structuration induit une structure hiérarchique sur les modules: un module de niveau *i* ne fait référence qu'à des modules de niveau inférieur ou égal à *i-1* et à au moins un module de niveau *i-1*. On parle dans ce cas de spécification hiérarchique.
Dans le langage CIRTA, les primitives de structuration des spécifications sont inspirées des primitives de structuration connues dans le domaine des spécifications algébriques et dans celui des réseaux de Petri.

2- Choix des primitives de structuration des spécifications

Le développement modulaire des spécifications est assisté dans le langage CIRTA par un ensemble de primitives permettant de structurer de façon adéquate les grandes spécifications. Ces primitives sont déduites des primitives connues d'une part dans le domaine des spécifications algébriques et d'autre part dans le domaine des réseaux de Petri.

Les principales primitives qu'on peut noter sont:

1- l'enrichissement de spécification
2- la somme (ou composition) de spécifications
3- le renommage d'une spécification
4- le contrôle de visibilité
5- la paramétrisation d'une spécification par d'autres spécifications
6- l'instanciation d'une spécification

3- Modularité de la sémantique du langage de spécification

La sémantique d'un langage est qualifiée de *modulaire* si on peut refléter la structure hiérarchique des spécifications au niveau sémantique. Dans ce cas la sémantique *globale* d'une spécification serait obtenue comme une certaine *combinaison* des sémantiques *élémentaires* de chacun des modules qui la compose. Cette *combinaison* est fonction de la structure de la spécification et des primitives d'assemblage des modules. Dans le cas des ECATNets, il semble difficile et complexe de définir une sémantique réellement modulaire i.e. associer un objet

mathématique à un module de spécification et d'obtenir la sémantique globale de la spécification comme combinaison de ces objets. Cette difficulté est due essentiellement au fait qu'on ne peut pas associer directement une signature à un module de spécification, car l'ensemble des déclarations des sortes et des opérations d'une même signature peuvent ne pas être définis dans le même module. En effet un module d'ECATNet peut par exemple utiliser des termes (dans les équations ou dans le réseau) définis sur la signature d'un autre module qu'il enrichit.
On essayera de munir le langage CIRTA d'une sémantique aussi modulaire que possible. Cette sémantique sera modulaire dans le sens où une spécification $\tilde{S}$ sera décomposée en un module $\hat{S}$ et un ensemble de sous-sépcifications $\widetilde{S_1}$, $\widetilde{S_2}$,..,$\widetilde{S_n}$ enrichi par le module $\hat{S}$. La sémantique de la spécification $\tilde{S}$ sera définie en combinant la sémantique du module $\hat{S}$ et les sémantiques des spécifications $\widetilde{S_1}$, $\widetilde{S_2}$,...,$\widetilde{S_n}$. En fait, il est presque toujours possible de définir un *foncteur d'oubli* $\breve{f}_i$ de la classe des modèles de $\tilde{S}$ à la classe des modèles de $\widetilde{S_i}$ quand $\hat{S}$ est un enrichissement de $\widetilde{S_i}$. Ce foncteur d'oubli $\breve{f}_i$ est également utilisé pour définir la sémantique des spécifications dans d'autres langages comme ASL et PLUSS.

IX- Conclusion

Il largement connu dans le domaine des méthodes formelles qu'il n'existe pas de langage de spécification *universel*, et que selon les aspects prépondérants d'un système (système de gestion de données, système de contrôle, interface utilisateur,...etc.) certaines techniques de spécification seront plus adéquates que d'autres. Le langage CIRTA est destiné à la spécification des systèmes concurrents, mais son domaine d'application couvre également la spécification algébrique des propriétés statiques des systèmes.

Nous avons précisé de façon générale, les caractéristiques principales du langage CIRTA qui permettront de le situer ou de le comparer avec d'autres langages de spécification formelle.

Dans les deux chapitres précédents nous avons présenté le formalisme de base du langage CIRTA. Les prochains chapitres détailleront chacun des autres aspects du langage tels que la modularité, la syntaxe et sémantique.

Chapitre 4

Modules de spécification dans le langage CIRTA

Sommaire

I- Introduction

Face à un problème complexe, en l'occurrence la spécification d'un système de taille réaliste, l'idée de décomposition en sous-problèmes plus petits ou modules s'impose à l'esprit. Elle n'est pas nouvelle: Descartes l'a prônée il y a bien des années, comme un moyen de réduire la complexité. Quand cette décomposition est répétée à plusieurs niveaux, on obtient une démarche de conception descendante (« *top down* »). De manière symétrique, mais aucunement exclusive, s'impose également l'idée de composition de la solution d'un problème ou d'un sous-problème, à partir des éléments partiels, précédemment réalisés. On parle alors de la démarche de conception ascendante (« *bottom up* »).

La recherche de décomposition de systèmes en éléments plus simples est donc, classiquement, une voie suivie pour surmonter les problèmes liés à l'établissement et au traitement de modèles de grandes dimensions. La méthode consiste à choisir une partition du système en divers sous-systèmes convenablement choisis selon la logique de fonctionnement du système en question. Les traits de modularité doivent être perçus comme un caractère découlant d'une aptitude du système à être considéré par parties. Sous cette forme, les spécifications sont désignées par modulaires. Leurs traits directeurs sont liés:

- au principe de représentation qui exprime les dépendances "hiérarchiques" de fonctionnement du système; ces relations de dépendances étant mises en évidence entre les sous-modules.
- aux modes pratiques d'analyse qui conduisent à une limitation des traitements à effectuer par exploitation des liens de dépendance entre les différentes parties du système.

Il s'avère donc utile de fournir dans le langage CIRTA un moyen de décomposer une spécification en différentes parties. La définition des composants d'une spécification et leur agencement constitue la structure de la spécification. La structuration d'une spécification permet un abord plus méthodique du processus de spécification. Elle peut être utilisée aussi pour représenter la structure même du système à spécifier. Elle facilite la tâche de construction de la spécification en la décomposant en sous-tâches, et simplifie son expression.

Dans les chapitres 1 et 2 nous avons présenté les éléments nécessaires pour la définition des ECATNets *non structurés*. Il est nécessaire d'enrichir le formalisme des ECATNets avec des primitives de structuration si l'on veut offrir un formalisme adapté à la spécification de systèmes complexes. Le but de ce chapitre est de définir la notion de *module d'ECATNet* permettant d'élaborer des spécifications structurées. Pour ce faire nous utilisons la notion de spécifications modulaires qui permet de composer des modules à l'aide de deux opérateurs principaux qui sont l'union et l'enrichissement. Des travaux effectués dans cette direction sur d'autres types de réseaux sont ceux entrepris par [BCM-88] présentant le langage OBJ-SA et les travaux de [BG-90] introduisant le formalisme de spécification CO-OPN.

Nous commençons ce chapitre par une présentation des qualités attendues du concept de *modularité* dans un langage de spécification. Nous définissons la notion d'*interface* d'un module d'ECATNet. Nous introduisons la notion de *module de spécification* CIRTA et nous citons quelques relations possibles entre les différents modules, et les principaux types de modules.

N.B : Il convient de noter que tout au long de ce chapitre le terme *ECATNets* est utilisé pour désigner les *ECATNets temporels* qui offrent un cadre de spécification plus général que celui des ECATNets proprement dits.

II- Qualités attendues et caractéristiques

Indiquons quelques qualités fondamentales que l'on peut attacher au concept de modularité et que l'on est en droit d'attendre, par ricochet, des constructions offertes par les langages de spécification:

- assurer une bonne *lisibilité.*
- faciliter *l'évolutivité.*
- faciliter la *vérification* des propriétés des modules et des assemblages de modules.
- faciliter la *réutilisation* de composants préexistants.
- permettre le *prototypage séparé* (indépendant) des composants.

Il découle de ces qualités attendues, un ensemble assez hétéroclite de caractéristiques, souvent associées au concept de modularité ; et qui concernent les langages de spécification et les méthodes de conception:

1) Les langages doivent offrir des *constructions syntaxiques* délimitant explicitement les composants modulaires.

2) Chaque module doit constituer 'un tout' par lui-même, relativement *autonome* (« *self contained* »); le texte du module doit être compréhensible en soi, sans lire autre chose, à l'exception d'un petit nombre d'interactions avec d'autres modules apparaissant explicitement au niveau de la partie *interface* de chaque module.

3) Les modules doivent permettre de *dissimuler* à la vue de l'extérieur des informations (« *information hiding* ») et en particulier, de *cacher les détails de leur réalisation* (notion d'*abstraction*); toute information non déclarée explicitement *publique* au niveau de l'interface, devrait être *privée* et donc inaccessible du dehors (notion de *protection*).

4) L'interface d'un module doit constituer une description suffisante pour permettre *l'écriture indépendante* d'autres modules interagissant avec lui.

5) Le *confinement des erreurs* au seul module où elles se déclarent, ou tout au moins le contrôle de leur propagation, doit être recherché.

6) Il est souvent intéressant de *ne pas dissocier spécification des structures de données et spécification du système de contrôle d'un même composant du système,* et au contraire, de les associer systématiquement au sein des composants modulaires .

7) La *taille* des modules doit être raisonnable.

8) Les modules doivent présenter une certaine *homogénéité logique* (cohérence interne).

9) Les interactions entre les modules, ou couplages doivent être en nombre réduit.

10) Toute modification de portée limitée doit n'avoir de conséquence que localisée dans un seul module.

III- Interface d'un module

Dès que l'on veut spécifier un grand système, il est indispensable de savoir découper la spécification en unités plus petites appelées modules de spécification; et une spécification sera constituée d'une collection de modules. Chacun de ces modules communique avec les autres via une interface bien définie. L'interface d'un module précise à la fois l'ensemble des sortes, des opérations, des places et des transitions définies dans ce module et accessibles aux autres modules.

Dans le langage CIRTA on suppose que les sortes et les opérations d'un module sont *implicitement accessibles* (visibles) aux autres modules et ceci pour les raisons suivantes :

- Les identificateurs de sortes et d'opérations d'un module *M*, ne peuvent que très rarement être locaux à ce module, dans la mesure où il leur sera le plus souvent fait référence dans les modules utilisant *M* (par enrichissement, composition, ou paramétrisation).
- Choisir pour le langage CIRTA un mécanisme de contrôle de visibilité (comme par exemple dans le cas du langage ADA) où lors de déclaration des sortes et des opérations, on stipule que ceux-ci sont locaux (non visibles), s'avère être une solution trop restrictive et donc mal adaptée dans le cas des langages de spécification. Il semble préférable de considérer (au contraire) que les identificateurs des sortes et des opérations ont une portée indéfinie, et d'offrir des mécanismes de contrôle de visibilité qui permettent (éventuellement lors de processus de spécification) de limiter la portée de certains identificateurs en les rendant inaccessibles pour les utilisations ultérieures.

Compte tenu de ces considérations, l'interface d'un module de spécification CIRTA est définie alors, par un ensemble de noms de places avec leurs sortes et leur capacité, et par un ensemble de transitions accompagnées de leurs caractéristiques (intervalle temporel, contexte indiquant une liste de variables et leurs types respectifs).
Les places interfaces d'un module sont des places accessibles aux autres modules. Les informations représentées par leurs marquages sont des informations (non cachées) publiques. Les transitions interfaces d'un module ont pour but d'être « appelées » par d'autres modules, les contextes de celles-ci servent aux transferts de valeurs typées.
Les places et transitions interfaces sont représentées respectivement par des cercles et rectangles en pointillé.

1- Définition

Soit $\mathcal{E} = ((Spec, R), IS)$ un ECATNet où $Spec = ((S, Op), E)$ et $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$.
Une interface de $\mathcal{E}$ est le couple $I = (P_I, T_I)$ telle que :

P_I : est l'ensemble des places interfaces ($P_I \subseteq P$)

T_I : est l'ensembles des transitions interfaces ($T_I \subseteq T$)

- Pour une interface donnée I , on note $Cap_I, \sigma_I, IC_I, DT_I, CT_I, TC_I, IS_I$ les restrictions respectives des fonctions $Cap, \sigma, IC, DT, CT, TC, IS$ à P_I et T_I .
- On définit également la fonction ctx_I qui associe à chaque transition de T_I la liste des variables *S*-sortées de son contexte : $ctx_I : T_I \to (V \times S)^*$
- Si le réseau est marqué par M alors M_I désigne le marquage des places de l'interface *I*.

Exemple : La Figure 4.1 représente le schéma d'un exemple d'interface où :
$I = (P_I, T_I)$ $P_I = \{p_1, p_2\}$ $T_I = \{t\}$ $Cap_I(p_1) = c_1$ $Cap_I(p_2) = c_2$ $\sigma_I(p_1) = s_1$ $\sigma p_I(p_2) = s_2$
$ctx_I(t) = (x : s_2, y : s_3)$ $IS_I(t) = [\min,\max]$

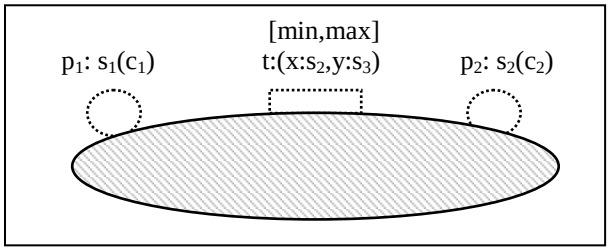


Figure 4.1 : Exemple d'interface d'un module.

2- Types d'interfaces

Les interfaces des modules peuvent être classées suivant le type de communications qu'elles offrent pour l'environnement de ces modules. Quand cette interface n'est pas vide(i.e. $P_I \cup T_I \neq \varnothing$), on distingue trois types d'interfaces: synchrones, asynchrones, et mixtes.

a- Interface synchrone

Une interface $I = (P_I, T_I)$ est synchrone si et seulement si $P_I = \varnothing \wedge T_I \neq \varnothing$. La communication du module avec son environnement consiste dans ce cas, en une synchronisation d'actions (transitions). Une transition interface décrit dans ce cas une partie d'une action complexe. Le franchissement d'une transition interface *t* n'est possible que si elle franchissable dans tous les modules ayant *t* en interface. Le franchissement de *t* s'effectue alors simultanément dans tous ces modules. Les différentes occurrences de *t* dans les différents modules correspondent à une même transition même si celle-ci est représentée par plusieurs exemplaires.

b- Interface asynchrone

Une interface $I = (P_I, T_I)$ est asynchrone si et seulement si $T_I = \varnothing \wedge P_I \neq \varnothing$. Dans ce cas, le module communique avec son environnement via des places symbolisant un partage de ressources ou d'informations. Une place interface *p* décrit une ressource commune à plusieurs modules. Une même place interface a la même sorte, la même capacité et le même marquage dans tous les modules dans lesquels elle est définie. Si un jeton est supprimé ou ajouté à *p* dans l'un des modules (suite au franchissement d'une transition de ce module), il est également supprimé ou ajouté dans toutes les occurrences de *p* dans les autres modules. Les différentes occurrences de *p* dans les différents modules correspondent à une même place même si celle-ci est représentée par plusieurs exemplaires

c- Interface mixte

Une interface $I = (P_I, T_I)$ est dite mixte si et seulement si $P_I \neq \varnothing$ *et* $T_I \neq \varnothing$. Ce type d'interface permet à un module d'établir à la fois des communications synchrones et asynchrones avec son environnement. C'est le type de communication le plus général qui permet une plus grande liberté durant la phase de spécification. En effet l'utilisateur n'est pas limité à un seul type de communication (synchrone ou asynchrone), mais peut spécifier des modules qui communiquent avec leur environnement à la fois de façon synchrone et asynchrone.

Quelque soit le type d'interfaces considéré, la communication entre deux systèmes ne peut s'établir que sous certaines conditions. Une première contrainte est le type d'interface de chacun d'eux. En effet un système à interface synchrone, par exemple, ne peut communiquer avec un système à communication asynchrone. Ils doivent au moins avoir une place interface ou une transition interface commune. En plus de cette contrainte, d'autres conditions s'imposent pour permettre une communication et une coopération cohérente entre les différents modules. Ces conditions définissent la notion de *compatibilité des interfaces*.

3- Compatibilité d'interfaces

La définition de la compatibilité de deux interfaces repose sur la définition des concepts de *sortes interfaces* et *signature interface.*

a- Sortes interfaces d'un module

Soit $I = (P_I, T_I)$ l'interface d'un ECATNet $\mathcal{E} = ((Spec, R), IS)$ (où: $Spec = ((S, Op), E)$ et $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$)

L'ensemble des sortes de l'interface *I,* noté S_I ($S_I \subseteq S$) est défini par:

(i) $\forall p \in P_I \quad \sigma_I(p) \in S_I$

(ii) $\forall t \in T_I \quad \left(ctx_I(t) = (x_1 : s_1, x_2 : s_2, ..., x_n : s_n) \Rightarrow \{s_1, s_2, ..., s_n\} \subseteq S_I\right)$

Exemple:

Soit $I = (P_I, T_I)$ l'interface schématisée par la Figure 4.2. L'ensemble des sortes interfaces est $S_I = \{s_1, s_4\}$.

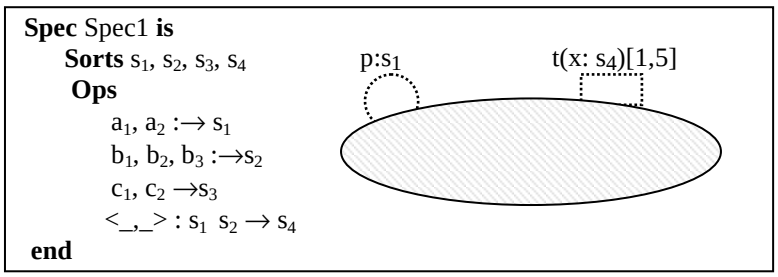


Figure 4.2 : Exemple de sortes interfaces d'un module.

Les nœuds (places et transitions) interfaces ayant pour but d'être « connectés » à d'autres modules, les sortes interfaces servent aux transferts de valeurs typées entre les modules.

Etant donnée la signature Σ d'un ECATNet, les sortes interfaces engendrent une sous-signature $\Sigma_I \subseteq \Sigma$ appelée *signature d'interface*.

b- Signature interface

Soit $Spec = ((S, Op), E)$ la spécification algébrique d'un ECATNet $\mathcal{E} = ((Spec, R), IS)$ et $S_I \subseteq S$ l'ensemble de sortes interfaces de $\mathcal{E}$. La signature interface Σ_I est définie par $\Sigma_I = (S_I', Op_I')$ telle que :

- S_I' est l'ensemble des sortes défini par :

$$\begin{cases} \forall s \ (s \in S_I \Rightarrow s \in S_I') \\ \forall s \in S_I' \ \forall (f : s_1 \ s_2 \ ... s_n \rightarrow s) \in Op \quad on \ a \quad \{s_1, s_2, ..., s_n\} \subseteq S_I' \end{cases}$$

- Op' est l'ensemble des opérations définies sur S_I' déterminé par :

$$\begin{cases} \forall \ s \in S_I' \ \forall (c :\rightarrow s) \in Op \quad on \ a \quad (c :\rightarrow s) \in Op_I' \\ \forall \ s_1, s_2, .., s_{n+1} \in S_I' \ \forall \ (f : s_1 \ s_2 \ ... s_n \rightarrow s_{n+1}) \in Op \quad on \ a \ (f : s_1 \ s_2 \ .. s_n \rightarrow s_{n+1}) \in Op_I' \end{cases}$$

Exemple:
La signature interface correspondant à l'exemple de la Figure 4.2 précédent est définie dans la Figure 4.3 ci-dessous.

Sorts s_1, s_2, s_4
Ops
a_1, $a_2 :\rightarrow s_1$
b_1, b_2, $b_3 :\rightarrow s_2$
$<\_,\_> : s_1 \ s_2 \rightarrow s_4$

Figure 4.3 : Exemple de signature interface Σ_I d'un module.

La définition de la signature interface d'un module est nécessaire pour procéder à la vérification de la compatibilité entre les modules lors d'un transfert de données (jetons) ou d'une synchronisation d'actions.

c- Compatibilité de deux interfaces

Soient $I_1 = (P_{I_1}, T_{I_1})$ et $I_2 = (P_{I_2}, T_{I_2})$ les interfaces de deux ECATNets $\mathcal{E}_1$ et $\mathcal{E}_2$ où $\mathcal{E}_i = (Spec_i, R_i), IS_i)$ $i = 1, 2$. I_1 et I_2 sont compatibles si et seulement si les conditions suivantes sont vérifiées:

i- Les spécifications algébriques $Spec_1$ et $Spec_2$ sont compatibles pour la signature commune $\Sigma_0 = \Sigma_{I_1} \cap \Sigma_{I_2}$ i.e. :

$$\begin{cases} \forall M_1 \in Mod(Spec_1) \ \exists M_2 \in Mod(Spec_2) \ / \ M_{1|\Sigma_0} = M_{2|\Sigma_0} \\ \forall M_2 \in Mod(Spec_2) \ \exists M_1 \in Mod(Spec_1) \ / \ M_{1|\Sigma_0} = M_{2|\Sigma_0} \end{cases}$$

où $Mod(Spec_i)$: est l'ensemble des modèles de la spécification algébrique $Spec_i$

$M_{|\Sigma_0}$: est la restriction d'un modèle M à la signature Σ_0. Cette restriction est définie à travers le concept de foncteur d'oubli.

ii- $\forall p \in P_{I_1} \cap P_{I_2}$ $\begin{cases} Cap_{I_1}(p) = Cap_{I_2}(p) \\ \sigma_{I_1}(p) = \sigma_{I_2}(p) \\ M_{I_1}(p) = M_{I_2}(p) \quad \textit{si les réseaux sont marqués} \end{cases}$

iii- $\forall t \in T_{I_1} \cap T_{I_2}$ $\begin{cases} IS_{I_1}(t) = IS_{I_2}(t) \\ \forall x \in V \ \left(\left((x:s) \in ctx_{I_1}(t) \wedge (x:s') \in ctx_{I_2}(t) \right) \Rightarrow s=s' \right) \end{cases}$

iv- $\forall (p,t) \in (P_{I_1} \times T_{I_1}) \cap (P_{I_2} \times T_{I_2})$ $\begin{cases} IC_{I_1}(p,t) = IC_{I_2}(p,t) & si \ (p,t) \in dom(IC_{I_1}) \cap dom(IC_{I_2}) \\ DT_{I_1}(p,t) = DT_{I_2}(p,t) & si \ (p,t) \in dom(DT_{I_1}) \cap dom(DT_{I_2}) \\ CT_{I_1}(p,t) = CT_{I_2}(p,t) & si \ (p,t) \in dom(CT_{I_1}) \cap dom(CT_{I_2}) \end{cases}$

(Un arc commun a les mêmes inscriptions dans les deux interfaces)

Exemple :

Soient I_1, I_2, I_3 les interfaces schématisées dans la Figure 4.4. I_1 et I_2 sont compatibles si elles vérifient la condition i) du paragraphe précédent ; alors que I_2 et I_3 ne sont pas compatibles (car p_1 n'a pas la même sorte dans I_2 et I_3).

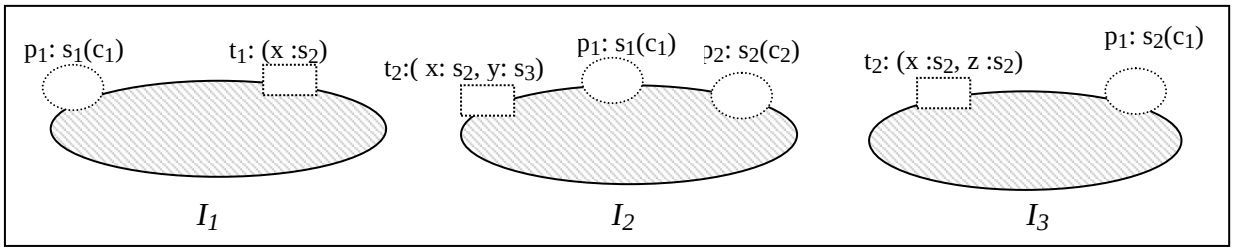


Figure 4.4 : Exemple d'interfaces compatibles.

Nous avons présentés dans les sections précédentes le concept d'*interface*, qui nous permet d'établir la définition *d'un module de spécification* nécessaire pour le langage de spécification CIRTA.

IV- Module de spécification

Un module de spécification CIRTA désigne une entité de spécification qui fournit la spécification d'une (petite) partie bien définie d'un système plus complexe. Un module peut faire appel à d'autres modules, leur transmettre des données et recevoir des données en retour. L'ensemble des modules ainsi reliés doit alors être capable de spécifier le système global.

Un module représentant ainsi une unité élémentaire de spécification, est constitué de déclarations de *sortes*, relations *de sous-sortes*, *opérations*, *variables*, *équations* et d'un *réseau*; ainsi qu'éventuellement des *directives (une interface)* précisant comment ce module doit être assemblé avec d'autres modules. Les modules de spécification seront notés $\hat{E}, \hat{E}_1, \hat{E}_2 \ldots \hat{E}_n$.

Les constructions permettant d'assembler entre eux un ensemble de modules pour former une spécification varient suivant les langages de spécification considérés; mais utilisent souvent deux notions communes qui sont l'*union* et l'*enrichissement*.

En général, la structuration d'une spécification complexe en une collection de modules a une structure de graphe orienté où les sommets correspondent aux modules et les arcs aux directives d'assemblage (Figure 4.5). Ce graphe possède un seul module racine qui est associé à la spécification globale du système. Un même module peut participer à plusieurs spécifications, qui peuvent être à leur tour des sous-spécifications d'une même spécification. De plus, chaque module du graphe détermine lui-même un sous-graphe dont il est racine et qui correspond à une sous-spécification. La spécification associée à un module $\hat{\mathcal{E}}$ est notée $\overleftrightarrow{\mathcal{E}}$.

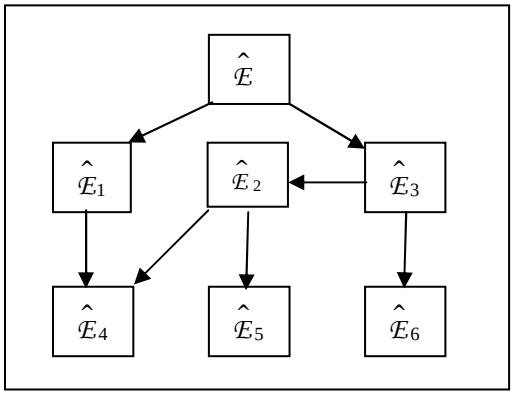


Figure 4.5: Exemple de graphe des modules associés à une spécification $\overleftrightarrow{\mathcal{E}}$.

1- Module de base

Dans le langage CIRTA, un module de base est une paire $\hat{\mathcal{E}} = (\mathcal{E}, I)$ où

- $\mathcal{E}=((Spec, R), IS)$ est un ECATNet

  $Spec = ((S, Op), E)$: est la spécification algébrique de $\mathcal{E}$

  $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$: est le réseau de $\mathcal{E}$
- $I=(P_I, T_I)$ est l'interface de $\hat{\mathcal{E}}$.

Exemple:
Le module de spécification de base ci-dessous, spécifie le comportement d'un philosophe dans le problème classique des philosophes.

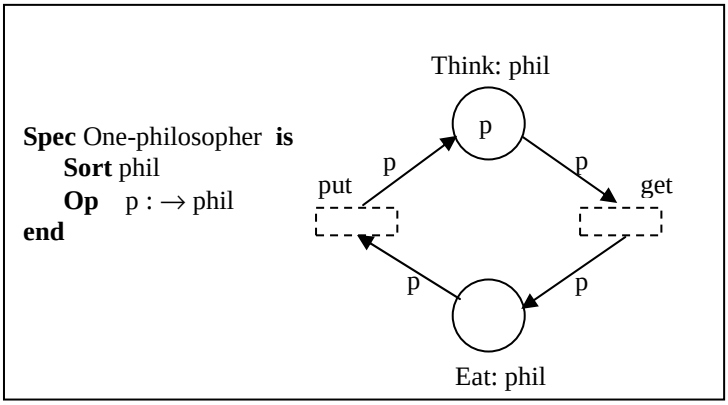


Figure 4.6: Exemple de module de base.

Les transitions *get et put* sont des transitions interfaces (i.e. $P_I = \varnothing$ *et* $T_I = \{put, get\}$)

2- Module structuré

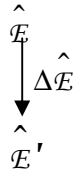
Un module structuré $\hat{\mathcal{E}}$ est :

- Soit un module de *base* $\hat{\mathcal{E}} = (\mathcal{E}, I)$.
- Soit un *enrichissement* d'un module $\hat{\mathcal{E}}'$ par un module de présentation $\Delta\hat{\mathcal{E}}$ noté $\hat{\mathcal{E}} = \hat{\mathcal{E}}' + \Delta\hat{\mathcal{E}}$.
- Soit une *union* de modules $\hat{\mathcal{E}}_1, \hat{\mathcal{E}}_2, ..., \hat{\mathcal{E}}_n$ notée $\hat{\mathcal{E}} = \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2 \cup ... \cup \hat{\mathcal{E}}_n$

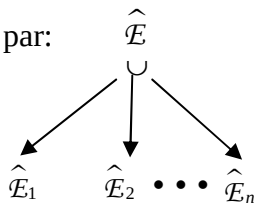
$\Delta\hat{\mathcal{E}}$ dénote l'ensemble des déclarations de sortes, des relations de sous-sortes, des opérations, des variables, des équations, et du réseau ajoutés à un module lors d'un enrichissement. Il faut noter qu'en général $\Delta\hat{\mathcal{E}}$ n'est pas un module et ne constitue pas une spécification car par exemple une opération déclarée dans $\Delta\hat{\mathcal{E}}$ peut avoir dans son profil (domaine et codomaine) des sortes non déclarées dans $\Delta\hat{\mathcal{E}}$ et par conséquent l'ensemble des sortes et opérations déclarées dans $\Delta\hat{\mathcal{E}}$ ne constituent pas une signature.

La structure d'un module de spécification sera représentée graphiquement comme suit:

- un module de base $\hat{\mathcal{E}}$ sera représenté par un sommet noté $\hat{\mathcal{E}}$ (sans sommet successeur)
- un module $\hat{\mathcal{E}} = \hat{\mathcal{E}}' + \Delta\hat{\mathcal{E}}$ sera représenté par:



- un module $\hat{\mathcal{E}} = \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2 \cup ... \cup \hat{\mathcal{E}}_n$ sera schématisé par:



D'après la définition précédente, chaque module de spécification est construit par assemblage de modules. L'ensemble de ces modules est appelé ensemble de modules associés à un module et est construit par la définition ci-après.

3- Ensemble de modules associés à un module

L'ensemble des modules associés à un module $\hat{\mathcal{E}}$ est l'ensemble noté $Modules\left(\hat{\mathcal{E}}\right)$ défini par:

- $Modules\left(\hat{\mathcal{E}}\right)=\left\{\hat{\mathcal{E}}\right\}$ si $\hat{\mathcal{E}}$ est un module de base
- $Modules\left(\hat{\mathcal{E}}\right)=Modules\left(\hat{\mathcal{E}'}\right)\cup\left\{\hat{\mathcal{E}}\right\}$ si $\hat{\mathcal{E}}=\hat{\mathcal{E}}'+\Delta\hat{\mathcal{E}}$
- $Modules\left(\hat{\mathcal{E}}\right)=\cup_{i=1..n}Modules\left(\hat{\mathcal{E}}_i\right)\cup\left\{\hat{\mathcal{E}}\right\}$ si $\hat{\mathcal{E}}=\hat{\mathcal{E}}_1\cup\hat{\mathcal{E}}_2\cup\ldots\cup\hat{\mathcal{E}}_n$

L'ensemble des modules associés à un module $\hat{\mathcal{E}}$ ne suffit pas pour déterminer la spécification algébrique définie par le module. En effet, cette spécification algébrique dépend également de la structure de $\hat{\mathcal{E}}$.

4- Spécification algébrique associée à un module

La spécification algébrique associée à un module $\hat{\mathcal{E}}$ est notée $Spec\left(\hat{\mathcal{E}}\right)$ et est définie par:

- $Spec\left(\hat{\mathcal{E}}\right)=Spec$ si $\hat{\mathcal{E}}=\left(\left(\left(Spec,R\right),IS\right),I\right)$ est un module de base
- $Spec\left(\hat{\mathcal{E}}\right)=Spec\left(\hat{\mathcal{E}}'\right)\cup Spec\left(\Delta\hat{\mathcal{E}}\right)$ si $\hat{\mathcal{E}}=\hat{\mathcal{E}}'+\Delta\hat{\mathcal{E}}$
- $Spec\left(\hat{\mathcal{E}}\right)=\cup_{i=1..n}Spec\left(\hat{\mathcal{E}}_i\right)$ si $\hat{\mathcal{E}}=\hat{\mathcal{E}}_1\cup\hat{\mathcal{E}}_2\cup\ldots\cup\hat{\mathcal{E}}_n$

La spécification algébrique associée à un module a la même structure hiérarchique que le module considéré.

Dans le paragraphe III de ce chapitre, nous avons présenté la notion d'interface indépendamment de la structure du module. Dans un module structuré l'ensemble des places et des transitions interfaces n'est pas connu à priori à cause de la structure hiérarchique du module. Ainsi, étant donné un module, il est nécessaire de connaître l'interface globale du module, qui n'est autre que l'union des interfaces de tous les modules qui le composent.

5- Interface associée à un module

L'interface associée à un module $\hat{\mathcal{E}}$ est notée $I\left(\hat{\mathcal{E}}\right)$ et est définie inductivement sur la structures de $\hat{\mathcal{E}}$ par:

- $I\left(\hat{\mathcal{E}}\right)=I$ si $\hat{\mathcal{E}}=\left(\mathcal{E},I\right)$ est un module de base
- $I\left(\hat{\mathcal{E}}\right)=I\left(\hat{\mathcal{E}}'\right)\uplus I\left(\Delta\hat{\mathcal{E}}\right)$ si $\hat{\mathcal{E}}=\hat{\mathcal{E}}'+\Delta\hat{\mathcal{E}}$
- $I\left(\hat{\mathcal{E}}\right)=\ \uplus_{i=1..,n}I\left(\hat{\mathcal{E}}_i\right)$ si $\hat{\mathcal{E}}=\hat{\mathcal{E}}_1\cup\hat{\mathcal{E}}_2\cup\ldots\cup\hat{\mathcal{E}}_n$

Notation:

$I\left(\hat{\mathcal{E}}_1\right) \uplus I\left(\hat{\mathcal{E}}_2\right) = \left(P_{I_1} \cup P_{I_2}, T_{I_1} \cup T_{I_2}\right)$ est l'union des interfaces des modules $\hat{\mathcal{E}}_1$ *et* $\hat{\mathcal{E}}_2$

où $I\left(\hat{\mathcal{E}}_1\right) = \left(P_{I_1}, T_{I_1}\right)$ $I\left(\hat{\mathcal{E}}_2\right) = \left(P_{I_2}, T_{I_2}\right)$.

6- Module de base associé à un module structuré

Un module est vu par définition comme une composition d'un ensemble de modules à l'aide des opérateurs d'enrichissement et d'union. Un module est donc une spécification structurée d'un système. La spécification non structurée du même système est dite "*spécification plate*" du système. Par conséquent, à chaque module $\hat{\mathcal{E}}$ on peut associer un module de base unique dit le "*module plat*" de $\hat{\mathcal{E}}$ noté $\overleftrightarrow{\mathcal{E}}$ correspondant à la spécification plate (non structurée) du système spécifié par $\hat{\mathcal{E}}$. Nous définissons le module plat associé à un module donné selon la structure de celui-ci.

6.1 Module plat associé à un module de base

Il est évident que le module plat associé à un module de base $\hat{\mathcal{E}}$ est $\overleftrightarrow{\mathcal{E}} = \hat{\mathcal{E}}$

6.2 Module plat associé à l'enrichissement d'un module

Soient $\hat{\mathcal{E}}'$: un module tel que $\overleftrightarrow{\mathcal{E}}' = \left(\mathcal{E}', I'\right)$

$\hat{\mathcal{E}} = \hat{\mathcal{E}}' + \Delta\hat{\mathcal{E}}$: un module enrichissant $\hat{\mathcal{E}}'$ par $\Delta\hat{\mathcal{E}}$ tels que :

$\mathcal{E}' = \left(\left(Spec', R'\right), IS'\right)$

$Spec' = \left(\left(S', Op'\right), E'\right) = \left(\Sigma', E'\right)$

$R' = \left(P', T', \sigma', Cap', IC', DT', CT', TC'\right)$

$I' = \left(P_I', T_I'\right)$

$\Delta\hat{\mathcal{E}} = \left(\left(\left(\Delta Spec, \Delta R\right), \Delta IS\right), \Delta I\right)$

$\Delta Spec = \left(\left(\Delta S, \Delta Op\right), \Delta E\right) = \left(\Delta\Sigma, \Delta E\right)$,

$\Delta R = \left(\Delta P, \Delta T, \Delta\sigma, \Delta Cap, \Delta IC, \Delta DT, \Delta CT, \Delta TC\right)$

$\Delta I = \left(\Delta P_I, \Delta T_I\right)$

Si $\overleftrightarrow{\mathcal{E}}'$ et $\Delta\hat{\mathcal{E}}$ communiquent via une interface effective compatible définie par l'ensemble des places $P_0 = P_I' \cap \Delta P_I$ et l'ensemble des transitions $T_0 = T_I' \cap \Delta T_I$, le module plat associé à $\hat{\mathcal{E}}$ noté $\overleftrightarrow{\mathcal{E}}$ défini par $\overleftrightarrow{\mathcal{E}} = \left(\mathcal{E}, I\right)$ (où $\mathcal{E} = \left(\left(Spec, R\right), IS\right)$, $R = \left(P, T, \sigma, Cap, IC, DT, CT, TC\right)$) tel que :

- $Spec = Spec' \cup \Delta\ Spec = ((S,Op),E)$ où $S=S' \cup \Delta S$ $Op=Op' \cup \Delta Op$, $E=E' \cup \Delta E$
- $P=P' \cup \Delta P$ $T=T' \cup \Delta T$
- $\sigma : P' \cup \Delta P \rightarrow S' \cup \Delta S$

$$\sigma(p)= \begin{cases} \sigma'(p) & si\ p \in P'\backslash P_0 \\ \Delta\sigma(p) & si\ p \in \Delta P\backslash P_0 \\ \sigma'(p)=\Delta\sigma(p) & si\ p \in P_0 \end{cases}$$

- $Cap : P' \cup \Delta P \rightarrow \mathbb{N}$

$$Cap(p)= \begin{cases} Cap'(p) & si\ p \in dom(Cap') \wedge p \in P'\backslash P_0 \\ \Delta Cap(p) & si\ p \in dom(\Delta Cap) \wedge p \in \Delta P\backslash P_0 \\ Cap'(p)=\Delta Cap(p) & si\ p \in dom(Cap') \cap dom(\Delta Cap) \wedge p \in P_0 \end{cases}$$

- $IC : (P' \cup \Delta P) \times (T' \cup \Delta T) \rightarrow \{\alpha / \alpha \in mT_{\Sigma' \cup \Delta\Sigma}(V)\} \cup \{\sim\alpha / \alpha \in mT_{\Sigma' \cup \Delta\Sigma}(V)\} \cup \{empty\}$

$$IC(p,t)= \begin{cases} IC'(p,t) & si\ (p,t) \in dom(IC')\backslash dom(\Delta IC) \\ \Delta IC(p,t) & si\ (p,t) \in dom(\Delta IC)\backslash dom(IC') \\ IC'(p,t)=\Delta IC(p,t) & si\ (p,t) \in dom(IC') \cap dom(\Delta IC) \end{cases}$$

- $DT : (P' \cup \Delta P) \times (T' \cup \Delta T) \rightarrow mT_{\Sigma' \cup \Delta\Sigma}(V) \cup \{\forall\}$

$$DT(p,t)= \begin{cases} DT'(p,t) & si\ (p,t) \in dom(DT')\backslash dom(\Delta DT) \\ \Delta DT(p,t) & si\ (p,t) \in dom(\Delta DT)\backslash dom(DT') \\ DT'(p,t)=\Delta DT(p,t) & si\ (p,t) \in dom(DT') \cap dom(\Delta DT) \end{cases}$$

- $CT : (P' \cup \Delta P) \times (T' \cup \Delta T) \rightarrow mT_{\Sigma' \cup \Delta\Sigma}(V)$

$$CT(p,t)= \begin{cases} CT'(p,t) & si\ (p,t) \in dom(CT')\backslash dom(\Delta CT) \\ \Delta CT(p,t) & si\ (p,t) \in dom(\Delta CT)\backslash dom(CT') \\ CT'(p,t)=\Delta CT(p,t) & si\ (p,t) \in dom(CT') \cap dom(\Delta CT) \end{cases}$$

- $TC : T' \cup \Delta T \rightarrow mT_{(\Sigma' \cup \Delta\Sigma),bool}(V)$

$$TC(t)= \begin{cases} TC'(t) & si\ t \in dom(TC') \setminus dom(\Delta TC) \\ \Delta TC(t) & si\ t \in dom(\Delta TC) \setminus dom(TC') \\ TC'(t)\ and\ \Delta TC(t) & si\ t \in dom(TC') \cap dom(\Delta TC) \end{cases}$$

- $IS : T' \cup \Delta T \rightarrow \mathbb{N} \times (\mathbb{N} \cup \{+\infty\})$

$$IS(t)= \begin{cases} IS'(t) & si\ t \in dom(IS') \setminus dom(\Delta IS) \\ \Delta IS(t) & si\ t \in dom(\Delta IS) \setminus dom(IS') \\ IS'(t)= \Delta IS(t) & si\ t \in dom(IS') \cap dom(\Delta IS) \end{cases}$$

- $I = I' \uplus \Delta I$ l'interface I de $\widehat{\mathcal{E}}$ est l'union des interfaces I' et ΔI
- Si $\widehat{\mathcal{E}}'$ et $\Delta\widehat{\mathcal{E}}$ sont marqués et ont respectivement les marquages M' et ΔM, le module $\widehat{\mathcal{E}}$ est alors marqué par le marquage M tel que :

$M : P' \cup \Delta P \rightarrow mT_{\Sigma' \cup \Delta\Sigma}(\varnothing)$

$$M(p)= \begin{cases} M'(p) & si\ p \in P'\backslash P_0 \\ \Delta M(p) & si\ p \in \Delta P\backslash P_0 \\ M'(p)=\Delta M(p) & si\ p \in P_0 \end{cases}$$

6.3 Module plat associé à l'union de modules

Soient $\hat{\mathcal{E}}_1$ *et* $\hat{\mathcal{E}}_2$: deux modules tels que $\overleftrightarrow{\mathcal{E}_i} = (\mathcal{E}_i, I_i)$ pour i=1,2.

$$\mathcal{E}_i = ((Spec_i, R_i), IS_i) \qquad I_i = (P_{I_i}, T_{I_i})$$
$$Spec_i = ((S_i, Op_i), E_i) = (\Sigma_i, E_i)$$
$$R_i = (P_i, T_i, \sigma_i, Cap_i, IC_i, DT_i, CT_i, TC_i)$$

Si $\overleftrightarrow{\mathcal{E}_1}$ et $\overleftrightarrow{\mathcal{E}_2}$ communiquent via une interface effective compatible définie par l'ensemble des places $P_0 = P_{I_1} \cap P_{I_2}$ et l'ensemble des transitions $T_0 = T_{I_1} \cap T_{I_2}$, le module plat associé à la composition de $\hat{\mathcal{E}}_1$ *et* $\hat{\mathcal{E}}_2$ notée $\hat{\mathcal{E}} = \hat{\mathcal{E}}_1 \cup \hat{\mathcal{E}}_2$ est le module $\overleftrightarrow{\mathcal{E}}$ défini par $\overleftrightarrow{\mathcal{E}} = (\mathcal{E}, I)$ (où $\mathcal{E} = ((Spec, R), IS)$, $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$) tel que :

- $Spec = Spec_1 \cup Spec_2 = ((S,Op),E)$ où $S=S_1 \cup S_2$ $Op=Op_1 \cup Op_2$, $E=E_1 \cup E_2$
- $P=P_1 \cup P_2$ $T=T_1 \cup T_2$
- $\sigma : P_1 \cup P_2 \to S_1 \cup S_2$

$$\sigma(p) = \begin{cases} \sigma_1(p) & si\ p \in P_1 \backslash P_0 \\ \sigma_2(p) & si\ p \in P_2 \backslash P_0 \\ \sigma_1(p) = \sigma_2(p) & si\ p \in P_0 \end{cases}$$

- $Cap : P_1 \cup P_2 \to \mathbb{N}$

$$Cap(p) = \begin{cases} Cap_1(p) & si\ p \in dom(Cap_1) \wedge p \in P_1 \backslash P_0 \\ Cap_2(p) & si\ p \in dom(Cap_2) \wedge p \in P_2 \backslash P_0 \\ Cap_1(p) = Cap_2(p) & si\ p \in dom(Cap_1) \cap dom(Cap_2) \wedge p \in P_0 \end{cases}$$

- $IC : (P_1 \cup P_2) \times (T_1 \cup T_2) \to \{\alpha / \alpha \in mT_{\Sigma_1 \cup \Sigma_2}(V)\} \cup \{\sim\alpha / \alpha \in mT_{\Sigma_1 \cup \Sigma_2}(V)\} \cup \{empty\}$

$$IC(p,t) = \begin{cases} IC_1(p,t) & si\ (p,t) \in dom(IC_1) \backslash dom(IC_2) \\ IC_2(p,t) & si\ (p,t) \in dom(IC_2) \backslash dom(IC_1) \\ IC_1(p,t) = IC_2(p,t) & si\ (p,t) \in dom(IC_1) \cap dom(IC_2) \end{cases}$$

- $DT : (P_1 \cup P_2) \times (T_1 \cup T_2) \to mT_{\Sigma_1 \cup \Sigma_2}(V) \cup \{\forall\}$

$$DT(p,t) = \begin{cases} DT_1(p,t) & si\ (p,t) \in dom(DT_1) \backslash dom(DT_2) \\ DT_2(p,t) & si\ (p,t) \in dom(DT_2) \backslash dom(DT_1) \\ DT_1(p,t) = DT_2(p,t) & si\ (p,t) \in dom(DT_1) \cap dom(DT_2) \end{cases}$$

- $CT : (P_1 \cup P_2) \times (T_1 \cup T_2) \to mT_{\Sigma_1 \cup \Sigma_2}(V)$

$$CT(p,t) = \begin{cases} CT_1(p,t) & si\ (p,t) \in dom(CT_1) \backslash dom(CT_2) \\ CT_2(p,t) & si\ (p,t) \in dom(CT_2) \backslash dom(CT_1) \\ CT_1(p,t) = CT_2(p,t) & si\ (p,t) \in dom(CT_1) \cap dom(CT_2) \end{cases}$$

- $TC : T_1 \cup T_2 \rightarrow mT_{(\Sigma_1 \cup \Sigma_2),bool}(V)$

$$TC(t) = \begin{cases} TC_1(t) & si\ t \in dom(TC_1) \setminus dom(TC_2) \\ TC_2(t) & si\ t \in dom(TC_2) \setminus dom(TC_1) \\ TC_1(t)\ and\ TC_2(t) & si\ t \in dom(TC_1) \cap dom(TC_2) \end{cases}$$

- $IS : T_1 \cup T_2 \rightarrow \mathbb{N} \times \left(\mathbb{N} \cup \{+\infty\}\right)$

$$IS(t) = \begin{cases} IS_1(t) & si\ t \in dom(IS_1) \setminus dom(IS_2) \\ IS_2(t) & si\ t \in dom(IS_2) \setminus dom(IS_1) \\ IS_1(t) = IS_2(t) & si\ t \in dom(IS_1) \cap dom(IS_2) \end{cases}$$

- $I = I_1 \uplus I_2$: L'interface I de $\widehat{E}$ est l'union des interfaces de I_1 et I_2
- Si $\widehat{E}_1$ et $\widehat{E}_2$ sont marqués respectivement par M_1 et M_2, le module $\widehat{E}$ est alors marqué par le marquage M tel que :

$M : P_1 \cup P_2 \rightarrow mT_{\Sigma_1 \cup \Sigma_2}(\varnothing)$

$$M(p) = \begin{cases} M_1(p) & si\ p \in P_1 \setminus P_0 \\ M_2(p) & si\ p \in P_2 \setminus P_0 \\ M_1(p) = M_2(p) & si\ p \in P_0 \end{cases}$$

Remarque :

La définition du module plat associé à $\widehat{E}_1 \cup \widehat{E}_2$ suppose que les identificateurs des places (resp. des transitions) non interfaces $\widehat{E}_1$ sont différents de ceux de $\widehat{E}_2$. Il en est de même pour $\widehat{E}' + \Delta\widehat{E}$

6.4 Exemple

Soient le module *Two-forks* spécifiant le comportement de deux fourchettes d'un philosophe.

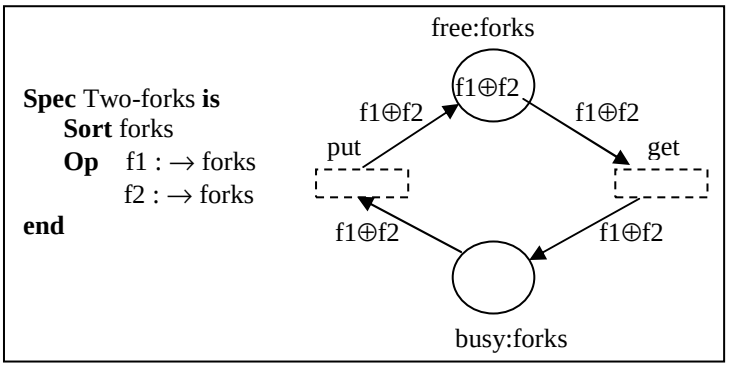


Figure 4.7: Exemple de module de base.

Les modules *Two-forks* (Figure 4.7) et *One-philosopher* (Figure 4.6) sont des modules de base ayant chacun comme interface (représentée en pointillé) les deux transitions *get* et *put*.
La spécification du comportement d'un philosophe ainsi que celui de ses deux fourchettes est donnée par le module *Two-forks-and-one-philosopher* spécifié dans la Figure 4.8.

Spec Two-forks-and-one-philosopher **is**
Combine Two-forks One-philosopher
end

Figure 4.8: Exemple de module structuré.

Le module *Two-forks-and-one-philosopher* est un module combinant les deux modules de base précédents qui se synchronisent via leur interface commune constituée des deux transitions *get* et *put*. Le module *Two-forks-and-one-philosopher* est équivalent au module plat qui lui est associé et qui est représenté par la Figure 4.9.

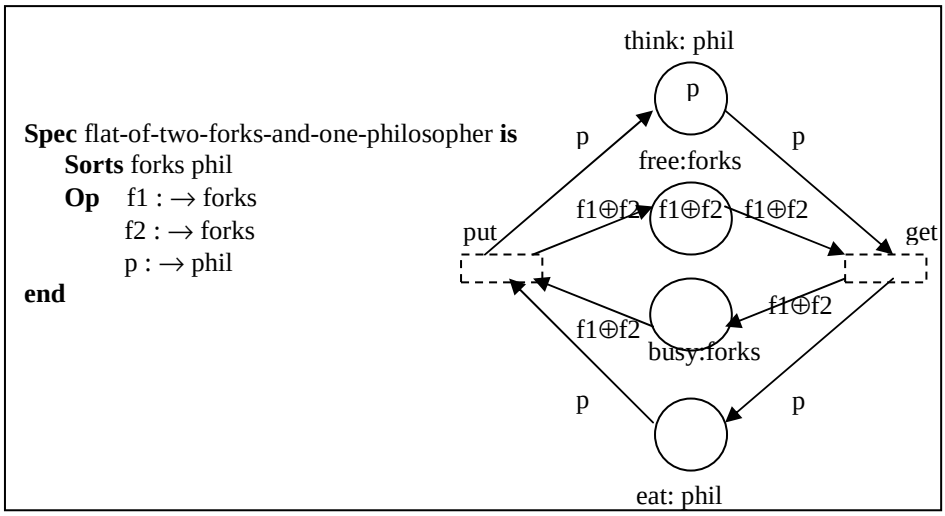


Figure 4.9: Exemple de module plat associé à un module structuré.

7- Marquage d'un module d'ECATNet:

L'état d'un module est caractérisé par la distribution des états des modules qui le composent. Ainsi, le marquage d'un module est réparti sur les places de tous ses sous-modules.

Soient $\hat{\mathcal{E}}$: un module de spécification

$Spec\left(\hat{\mathcal{E}}\right)$: la spécification algébrique associée au module $\hat{\mathcal{E}}$

$\Sigma\left(\hat{\mathcal{E}}\right)$: est la signature de $Spec\left(\hat{\mathcal{E}}\right)$

$P\left(\hat{\mathcal{E}}\right)$: est l'ensemble des places de $\hat{\mathcal{E}}$ défini par:

$$\begin{cases} \bullet\ P\left(\hat{\mathcal{E}}\right)=\{p_1, p_2, .., p_k\} & si\ \ \hat{\mathcal{E}}\ \textit{est un module de base et}\ \{p_1, p_2, .., p_k\}\ \textit{est l'ensemble de ses places} \\ \bullet\ P\left(\hat{\mathcal{E}}\right)=P\left(\hat{\mathcal{E}}'\right)\cup P\left(\Delta\hat{\mathcal{E}}\right) & si\ \ \hat{\mathcal{E}}=\hat{\mathcal{E}}'+\Delta\hat{\mathcal{E}} \\ \bullet\ P\left(\hat{\mathcal{E}}\right)=\cup_{i=1..n} P\left(\hat{\mathcal{E}}_i\right) & si\ \ \hat{\mathcal{E}}=\hat{\mathcal{E}}_1\cup\hat{\mathcal{E}}_2\cup ...\cup\hat{\mathcal{E}}_n \end{cases}$$

Un marquage du module $\widehat{\mathcal{E}}$ est une application :

$$\widehat{M} : P\left(\widehat{\mathcal{E}}\right) \rightarrow mT_{\Sigma\left(\widehat{\mathcal{E}}\right)}$$

telle que : $\forall \widehat{\mathcal{E}}_i \in Modules\left(\widehat{\mathcal{E}}\right)$ $\widehat{M}_{|P\left(\widehat{\mathcal{E}}_i\right)}$ *est un marquage de* $\widehat{\mathcal{E}}_i$

Un module marqué est alors défini par le couple $\left(\widehat{\mathcal{E}}, \widehat{M}\right)$.

V- Comportement d'un module

Le comportement d'un module de spécification CIRTA est défini par le comportement de ses sous-modules. Ces sous-modules spécifient des sous-systèmes qui n'évoluent pas indépendamment les uns des autres: des synchronisation au niveau des transitions interfaces, et des partages de ressources communes modélisées par les places interfaces se produisent. Il est donc important de préciser certains concepts tels que le contexte d'une transition interface, la franchissabilité et la sensibilisation d'une transition interface et l'effet de son franchissement sur l'état (marquage) global du module. Il est également nécessaire de définir la façon suivant laquelle change le marquage d'une place interface.

1- Contexte d'une transition interface

Soit t une transition interface dans un module de spécification $\widehat{\mathcal{E}}$ (i.e. $t \in T_{I\left(\widehat{\mathcal{E}}\right)}$)

Le contexte de *t* dans $\widehat{\mathcal{E}}$ noté $ctx_{\widehat{\mathcal{E}}}(t)$ est défini par:

- $ctx_{\widehat{\mathcal{E}}}(t) = ctx_I(t)$ si $\widehat{\mathcal{E}} = (\mathcal{E}, I)$ est un module de base.
- $ctx_{\widehat{\mathcal{E}}}(t) = ctx_{\widehat{\mathcal{E}}'}(t) * ctx_{\Delta\widehat{\mathcal{E}}}(t)$ si $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}' + \Delta\widehat{\mathcal{E}}$
- $ctx_{\widehat{\mathcal{E}}}(t) = *_{i=1..n} ctx_{\widehat{\mathcal{E}}_i}(t)$ si $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}_1 \cup \widehat{\mathcal{E}}_2 \cup ... \cup \widehat{\mathcal{E}}_n$

2- Sensibilisation et franchissabilité d'une transition interface

- Une transition interface *t* d'un module $\widehat{\mathcal{E}}$ est *sensibilité* pour un marquage $\widehat{M}$ si et seulement si il existe au moins une affectation consistante aff_t de $ctx_{\widehat{\mathcal{E}}}(t)$ telle que:

$$\forall \widehat{\mathcal{E}}_i \in Modules\left(\widehat{\mathcal{E}}\right) \quad t \in T_{I\left(\widehat{\mathcal{E}}_i\right)} \Rightarrow t \text{ sensibilisée dans } \widehat{\mathcal{E}}_i \text{ pour le marquage } M_{|P\left(\widehat{E}_i\right)}$$
$$\text{et l' affectation } aff_{t|ctx_i(t)}$$

où $ctx_i(t)$ est la restriction de $ctx_{\widehat{\mathcal{E}}}(t)$ à $\widehat{\mathcal{E}}_i$ (i.e. $ctx_i(t) \subseteq ctx_{\widehat{\mathcal{E}}}(t)$)

- Une transition interface *t* d'un module $\widehat{\mathcal{E}}$ est *franchissable* pour un marquage $\widehat{M}$ et une affectation consistante aff_t de $ctx_{\widehat{\mathcal{E}}}(t)$ si et seulement si elle est restée sensibilisée depuis au moins $\downarrow IS_I(t)$ unités de temps et pas plus de $\uparrow IS_I(t)$.

3- Franchissement d'une transition interface

Le franchissement d'une transition interface *t* dans un module $\widehat{\mathcal{E}}$ pour un marquage $\widehat{M}$ et une affectation consistante aff_t de $ctx_{\widehat{\mathcal{E}}}(t)$ donne le nouveau marquage $\widehat{M}'$ défini par:

$$\forall \widehat{\mathcal{E}}_i \in Modules\left(\widehat{E}\right) \quad \widehat{M}_{|\widehat{\mathcal{E}}_i}\left[\left(t, aff_{t|ctx_i(t)}\right)>\widehat{M}'_{|\widehat{\mathcal{E}}_i}\right.$$

4- Evolution du marquage d'une place interface

Soient *p*: une place interface dans $\widehat{\mathcal{E}}$ (i.e. $p \in P_{I(\widehat{\mathcal{E}})}$)

$\widehat{M}_0$: le marquage initial de $\widehat{\mathcal{E}}$

A tout instant, le marquage de *p* vérifie les propriétés suivantes :

- $\forall \hat{\mathcal{E}}_i, \hat{\mathcal{E}}_j \in Modules(\hat{\mathcal{E}}) \qquad p \in P_{I(\hat{\mathcal{E}}_i)} \cap P_{I(\hat{\mathcal{E}}_j)} \Rightarrow \widehat{M}_{0|P(\hat{\mathcal{E}}_i)}(p) = \widehat{M}_{0|P(\hat{\mathcal{E}}_j)}(p)$
- $\forall \hat{\mathcal{E}}_i, \hat{\mathcal{E}}_j \in Modules(\hat{\mathcal{E}}) \ \forall \widehat{M} \qquad \left(\widehat{M}_0\left[>\widehat{M} \ \wedge \ p \in P_{I(\hat{\mathcal{E}}_i)} \cap P_{I(\hat{\mathcal{E}}_j)}\right) \Rightarrow \widehat{M}_{|P(\hat{\mathcal{E}}_i)}(p) = \widehat{M}_{|P(\hat{\mathcal{E}}_j)}(p)\right.$

Cette règle d'évolution du marquage d'une place interface signifie qu'une place interface *p* a le même marquage dans tous les modules où elle est définie (que ce marquage soit le marquage initial ou un marquage accessible à partir du marquage initial).
Par définition, les différentes occurrences d'une place interface *p* représente *une même place:* tout jeton ajouté (resp. enlevé) à une occurrence de cette place dans un module, est *implicitement* ajouté (resp. enlevé) à toutes les autres occurrences de *p* dans tous les autres modules (ayant *p* en interface). Une place interface est donc vue comme une place commune aux différents modules où elle est définie.

VI- Quelques relations entre les modules

1- Modules à spécifications algébriques équivalentes

Soient $\hat{\mathcal{E}}_1$ et $\hat{\mathcal{E}}_2$ deux modules, ayant respectivement les spécifications algébriques associées $Spec\left(\hat{\mathcal{E}}_1\right)$ et $Spec\left(\hat{\mathcal{E}}_2\right)$. Les spécifications algébriques $Spec\left(\hat{\mathcal{E}}_1\right)$ et $Spec\left(\hat{\mathcal{E}}_2\right)$ sont équivalentes et on note $Spec\left(\hat{\mathcal{E}}_1\right) \equiv Spec\left(\hat{\mathcal{E}}_2\right)$ si et seulement si: $Mod\left(Spec\left(\overleftrightarrow{\mathcal{E}}_1\right)\right) = Mod\left(Spec\left(\overleftrightarrow{\mathcal{E}}_2\right)\right)$

Notation:

$Mod\left(Spec\left(\overleftrightarrow{\mathcal{E}}\right)\right)$ dénote la classe des modèles de la spécification algébrique du module plat associé à $\hat{\mathcal{E}}$.

Les modules ayant des spécifications algébriques équivalentes sont des modules qui ont les mêmes propriétés *statiques*. Ils manipulent les mêmes types abstraits de données.

2- Modules à comportement local équivalent

Soient $\hat{\mathcal{E}}_1$ et $\hat{\mathcal{E}}_2$ deux modules de spécification CIRTA et $\forall i = 1,2\ \ \overleftrightarrow{\mathcal{E}}_i = (\mathcal{E}_i, I_i)$ est le module plat associé à $\hat{\mathcal{E}}_i$. $\hat{\mathcal{E}}_1$ et $\hat{\mathcal{E}}_2$ ont un comportement local équivalent si et seulement si: $\mathcal{E}_1 = \mathcal{E}_2$

Deux modules CIRTA à comportement local équivalent ont le même ECATNet plat (pas nécessairement le même module plat) mais peuvent coopérer via des interfaces différentes avec leur environnement.
Quand ils ne communiquent pas avec leur environnement (pas de synchronisation, ni de partage de ressources ou transfert de données avec d'autres modules), les modules à comportement local équivalent ont des comportements identiques.

3- Modules à comportement observationnel équivalent

Deux modules de spécification $\hat{\mathcal{E}}_1$ et $\hat{\mathcal{E}}_2$ ont un comportement observationnel (visible) équivalent si et seulement si:

- $I(\hat{\mathcal{E}}_1) = I(\hat{\mathcal{E}}_2)$ ($\hat{\mathcal{E}}_1$ et $\hat{\mathcal{E}}_2$ ont même interface)
- $I(\hat{\mathcal{E}}_1)$ et $I(\hat{\mathcal{E}}_2)$ ont le même comportement

Le comportement interface d'un module est le comportement visible (observationnel) de ce module. Il est caractérisé par le marquage des places interfaces et les séquences de franchissement des transitions interfaces.
L'état interface est un sous-état de l'état global du module. Une séquence de franchissement interface est une sous-séquence d'une séquence de franchissement du module.
Deux modules à comportement observationnel équivalents peuvent avoir des modules plats (associés) différent mais ont le même comportement visible dans tout environnement donné. Ceci signifie qu'ils coopèrent de la même manière avec l'extérieur mais n'ont pas nécessairement le même comportement local (non visible).
Différentes notions sur l'équivalence observationnel des comportements des réseaux de Petri (de haut niveau ou non) existent telles que la notion de "*bissimulation*" [NMV-90][BDKP-91][NW-95][FG-06] et "*d'équivalence*" [POM-85][BGV-90][RES-99]. Elles font l'objet de plusieurs travaux dans le domaine de l'analyse des propriétés des réseaux de Petri. La vérification d'une telle propriété sort du cadre de notre travail et concerne plutôt le domaine de l'analyse modulaire des ECATNets .

4- Equivalence de modules

Deux modules de spécification $\hat{\mathcal{E}}_1$ et $\hat{\mathcal{E}}_2$ sont équivalents et on note $\hat{\mathcal{E}}_1 \equiv \hat{\mathcal{E}}_2$ si et seulement si: Les *modules* plats qui leur sont associés sont égaux i.e. $\overleftrightarrow{\mathcal{E}}_1 = \overleftrightarrow{\mathcal{E}}_2$.

Cette notion d'équivalence entre deux modules recouvre à la fois l'équivalence de spécifications algébriques, l'équivalence d'interfaces, l'équivalence de comportement local et observationnel.

VII- Typologies des modules CIRTA

Les modules de spécification CIRTA peuvent être classés selon différents critères. Un premier exemple de critère consiste à associer à un module le type de son interface: on parlera alors de modules à communication synchrones, de modules à communication asynchrone et de modules à communication mixte. Un autre critère de différenciation entre les modules est leur structure: on distinguera alors les modules de base (qui ne font référence à aucun autre module) et les modules structurés (qui sont construits à partir d'autres modules). On peut également classer les modules CIRTA selon le type de systèmes ou de problèmes qu'ils spécifient: on parlera alors de module de structure de données (quand le réseau est vide), ou de modules de structure de contrôle (quand le module a un réseau non vide).
Ces classifications dépendent souvent des manipulations destinées à être effectuées sur les modules (exemple: analyse des systèmes synchrones, implémentation des modules de base, …etc.). Dans le cadre de la définition du langage CIRTA, nous classons les modules selon leur sémantique. Les différents types de modules définis dans CIRTA permettent une grande souplesse d'utilisation et favorise le développement modulaire des spécifications. On distingue dans ce langage trois types de modules: les modules ordinaires, les modules paramètre formels, et les modules génériques

1- Module ordinaire

Un module de spécification ordinaire dans CIRTA est introduit par le mot clé **Spec** et se termine par le mot réservé **end**.

Il peut être soit :
- Un module de base
- L'enrichissement d'autres modules ordinaires
- La composition d'un ensemble de modules ordinaires
- Le renommage d'un module ordinaire
- le contrôle de visibilité d'un module ordinaire
- L'instanciation totale d'un module générique

Il est constitué de :
- Un en-tête précisant le nom du module
- Directives d'enrichissement, de composition, de renommage, de visibilité ou d'instanciation d'autres modules
- Corps du module proprement dit (c'est à dire un ensemble de sortes, de relations de sous-sortes, d'opérations, de variables, d'axiomes et d'un réseau).

Les règles relatives à la syntaxe des modules ordinaires sont:

Module-ordinaire ::= **Spec** Mod-id **is** corps-de-module **end**
Spec Mod-id **is use** Module-requis corps-du-module **end**
Spec Mod-id **is combine** Modules-à-combiner **end**
Spec Mod-id **is rename** Module-à-renommer morphisme **end** /
Spec Mod-id **is** Module-visibilité **end /**
Spec Mod-id **is** Module-instanciation **end**

L'enrichissement est réalisé au moyen de la primitive **use** qui permet de spécifier de façon incrémentale de nouvelles sortes, opérations, axiomes, et parties de réseaux.

2- Module paramètre formel

L'objectif des modules paramètres formels est de préciser les propriétés *minimales* requises pour que les spécifications paramètres effectifs soient des paramètres admissibles (ou valides). Ces propriétés minimales peuvent concerner la définition des sortes, la définition des opérations, les propriétés devant être vérifiées par les opérations ou bien des propriétés sur le comportement dynamique décrit par le réseau du module paramètre formel.

Un module paramètre formel peut être soit :

- Un module paramètre formel de base
- L'enrichissement d'autres modules (ordinaires,ou modules paramètres formels)
- La composition d'un ensemble de modules (ordinaires, ou paramètres formels)
- Le renommage d'un autre module (ordinaire, ou module paramètre formel)
- Le contrôle de visibilité d'un autre module (ordinaire, ou module paramètre formel)
- L'instanciation totale d'un module générique

Il est constitué de :

- Un en-tête précisant le nom du module
- Directives d'enrichissement, de composition ou d'instanciation d'autres modules
- Corps du module proprement dit (c'est à dire un ensemble de sortes, d'opérations, de variables, d'axiomes et d'un réseau).

Le corps du module de spécification paramètre formel est en général assez réduit. Les règles syntaxiques des modules paramètre formels sont les suivantes:

Module-paramètre::= **Formal** Mod-id **is** corps-de-module **end /**
Formal Mod-id **is use** Module-requis corps-du-module **end /**
Formal Mod-id **is combine** Modules-à-combiner **end /**
Formal Mod-id **is rename** Module-à-renommer morphisme **end** /
Formal Mod-id **is** Module-visibilité **end /**
Formal Mod-id **is** Module-instanciation **end**

La structure d'un module paramètre formel est pratiquement semblable à celle d'un module ordinaire. La différence réside dans leur sémantique. Nous expliquerons dans le chapitre 9 qu'un module paramètre formel et un module ordinaire ont généralement des classes de modèles différentes.

3- Module générique

Les modules génériques définissent chacun une classe de modules. Ils permettent d'éviter l'écriture d'autant de versions d'un même "modèle" de spécification que d'instance de ce modèle sont nécessaires. La paramétrisation s'avère ainsi être un moyen de réutiliser aisément des modules de spécification et compte de ce fait parmi les mécanisme indispensables à tout langage de spécification modulaire. La paramétrisation d'un module concerne la spécification entière définie par ce module.
Un module de spécification paramétré est introduit par le mot clé **Gen** et se termine par le mot réservé **end**.

Un module générique est soit:

- L'enrichissement d'autres modules ordinaires, formel ou génériques
- La composition d'un ensemble de modules ordinaires, formel ou génériques
- Le renommage d'un module générique
- Le contrôle de visibilité d'un module générique
- L'instanciation partielle d'un module générique
- L'instanciation formelle d'un module générique

Il est constitué de

- Un en-tête précisant le nom du module et la liste des modules paramètres formels
- Directives d'enrichissement, de composition ou d'instanciation d'autres modules
- Corps du module proprement dit (c'est à dire un ensemble de sortes, d'opérations, de variables, d'axiomes et d'un réseau).

Un module générique est construit de la même manière qu'un module ordinaire. La différence réside dans la liste des identificateurs des paramètres formels. Cette liste fait implicitement partie de l'identificateur du module paramétré.

Les règles syntaxiques pour les modules paramétrés sont comme suit:

Module-générique::=

Gen Mod-id[paramètre-formel]$^{+}$ **is** corps-de-module **end /**
Gen Mod-id [paramètre-formel]$^{+}$ **is use** Module-requis corps-du-module **end /**
Gen Mod-id[paramètre-formel]$^{+}$ **is combine** Modules-à-combiner **end /**
Gen Mod-id[paramètre-formel]$^{+}$ **is rename** Module-à-renommer morphisme **end** /
Gen Mod-id [paramètre-formel]$^{+}$ **is** Module-visibilité **end /**
Gen Mod-id [paramètre-formel]$^{+}$ **is** Module-instanciation **end**

VIII- Conclusion

Dans ce chapitre nous avons enrichi le formalisme des ECATNets par le concept de *module*. L'avantage de cette approche est que l'on peut spécifier tous les composants d'un système de façon indépendante et exprimer leurs interactions. Plusieurs opérations de compositions peuvent être ensuite appliquées pour permettre l'obtention d'un module décrivant le système dans son ensemble; et le module ainsi obtenu pourra être réutilisé comme composant d'un autre système.

Chaque module de spécification CIRTA a une structure hiérarchique et communique avec les autres modules via une *interface*. Cette communication peut être synchrone, asynchrone ou les deux à la fois. Ce qui offre une certaine liberté lors du processus de spécification où l'utilisateur n'est pas limité dans la définition des modules à un seul type d'interface comme dans [BBG-97].

Une fois construits, les modules peuvent être combinés et transformés pour définir les spécifications des systèmes plus complexes. L'objectif du langage CIRTA est de fournir les outils théoriques et pratiques pour faciliter une telle tâche. Nous exposons dans le chapitre suivant les principales primitives du langage CIRTA qui permettent d'enrichir, de composer, de renommer et de générer des modules de spécification. La sémantique formelle des modules de spécification est décrite dans le chapitre 9.

Chapitre 5

Primitives du langage CIRTA

Sommaire

I- Introduction

Les modules d'ECATNets définis dans le chapitre précédent, permettent de spécifier des composants d'un système. La spécification globale d'un système est obtenue par une certaine combinaison de ces modules. Afin de favoriser le développement modulaire des spécifications, nous présentons dans ce chapitre un ensemble de primitives permettant de créer de nouvelles spécifications à partir de modules de spécification existants. Ces primitives de combinaisons des modules sont inspirées du domaine des spécifications algébriques et de celui des réseaux de Petri. Elles se basent sur l'enrichissement, la composition, le renommage et le contrôle de visibilité. Pour chacune de ces primitives, nous présentons son principe, éventuellement ses différentes variantes, un exemple d'utilisation et sa syntaxe. La sémantique formelle de ces primitives est détaillée en chapitre 9.

II- Enrichissement

Le développement de la spécification d'un système en un seul niveau peut causer l'omission ou l'oubli de certains aspects en raison du nombre souvent important de détails considérés en même temps. De plus un tel type de développement ne permet pas une présentation adéquate de la structure du système spécifié. C'est pourquoi une approche progressive de spécification basée sur l'*enrichissement* incrémental des spécifications est plus adoptée puisqu'elle permet, entre autre, de faire abstraction (temporaire) de certains détails et de séparer le système en composants bien définis.

1- Principe de l'enrichissement

L'enrichissement est l'opération de structuration fondamentale dans le langage CIRTA. Elle a pour but de permettre *l'ajout* (sans redondance) de nouveaux éléments (sortes, opérations, variables, axiomes, réseau) à un module de spécification préalablement défini. Considéré depuis le module où se fait l'enrichissement, le module enrichi est appelé module *requis* ou *importé*.

La construction d'un module $\hat{\mathcal{E}}$ par enrichissement un module $\hat{\mathcal{E}}'$ est abstraitement notée par: $\hat{\mathcal{E}} = \hat{\mathcal{E}}' + \Delta\mathcal{E}$. La partie de spécification $\Delta\mathcal{E}$ précise l'ensemble des éléments (sortes, opérations,…etc.) ajoutés à $\hat{\mathcal{E}}'$, et ne constitue pas en général, par elle-même, un module de spécification. En effet l'ensemble des sortes et opérations peuvent ne pas former une signature (les opérations de $\Delta\mathcal{E}$ peuvent être définies sur des sortes non spécifiées par $\Delta\mathcal{E}$, les axiomes peuvent utiliser des opérations autres que celles de $\Delta\mathcal{E}$, et la partie de réseau ajoutée peut avoir comme jetons des termes non définis sur les sortes et opérations de $\Delta\mathcal{E}$).

2- Types d'enrichissement

L'enrichissement, dans le langage CIRTA, est une construction inspirée d'une part, du domaine des spécifications algébriques (où elle est largement utilisée sous différentes formes : *enrich, use, extend, protect* [ABK-02] [BID-89] [GT-79]) ; et d'autre part du domaine des réseaux de Petri (où on se base sur la notion de *fusion* de réseaux via les places et/ou les transitions communes [SEN-96], ainsi que la notion de *substitution* de places ou de transitions [JEN-92]).
L'enrichissement dans le langage CIRTA supporte à la fois le développement ascendant et descendant des spécifications, et peut être de deux types : un enrichissement par *duplication* ou un enrichissement par *substitution*.

2-1 Enrichissement par duplication

L'enrichissement par duplication consiste à ajouter des éléments $\Delta\mathcal{E}$ à un module $\widehat{\mathcal{E}}'$ tels que $\widehat{\mathcal{E}}'$ et $\Delta\mathcal{E}$ partagent *éventuellement* un ensemble d'éléments (places interfaces, ou transitions interfaces). Un élément commun (ou partagé) appartient à l'interface de chaque module ($\widehat{\mathcal{E}}$ et $\widehat{\mathcal{E}}'$), et sa définition est, par conséquent, "*dupliquée*" dans chacun de ces modules.
La définition d'un module $\widehat{\mathcal{E}}$ enrichissant par duplication le module $\widehat{\mathcal{E}}'$ par $\Delta\mathcal{E}$, est réalisée au moyen de la primitive ***use*** selon la Figure5.1 où $\widehat{\mathcal{E}}'$ et $\widehat{\mathcal{E}}$ ont éventuellement des éléments communs parmi les places interfaces $\{p_1,\ldots p_n\}$ et les transitions interfaces $\{t_1,\ldots t_m\}$.

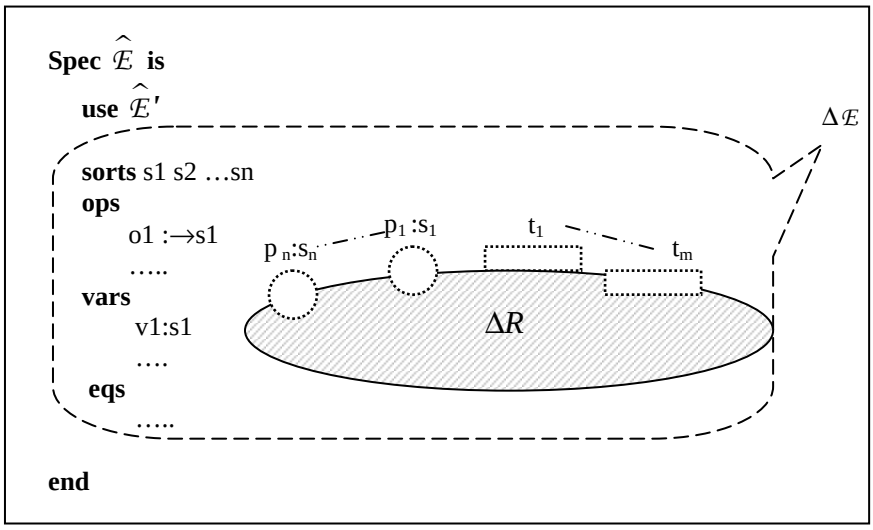


Figure 5.1: Principe de l'enrichissement par duplication.

La Figure 5.2 présente un exemple d'application de la primitive *use* où le module $\widehat{\mathcal{E}}$ enrichit $\widehat{\mathcal{E}}'$ qui à son tour enrichit le module $\widehat{\mathcal{E}}''$.

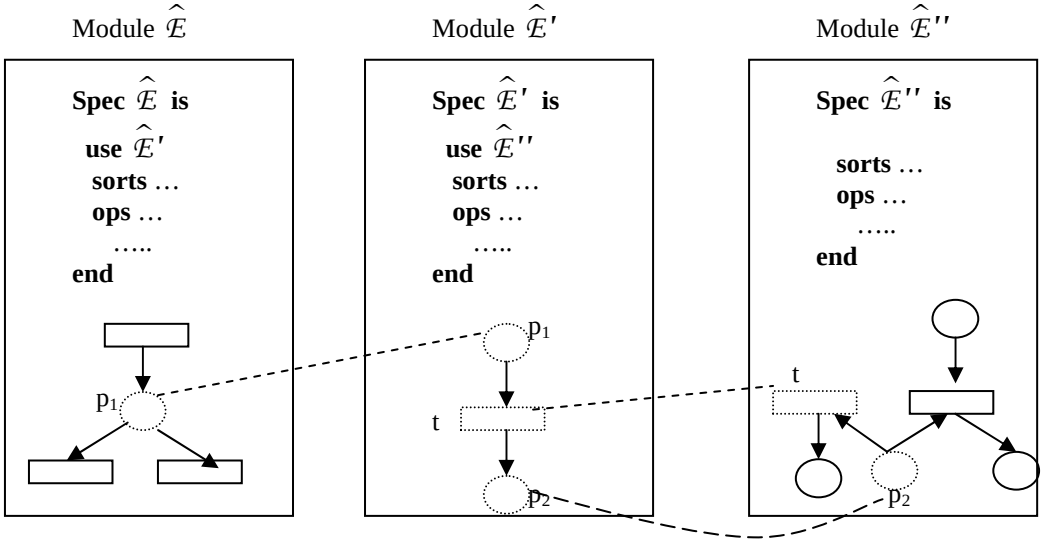


Figure 5.2 : Exemple d'enrichissements par duplication.

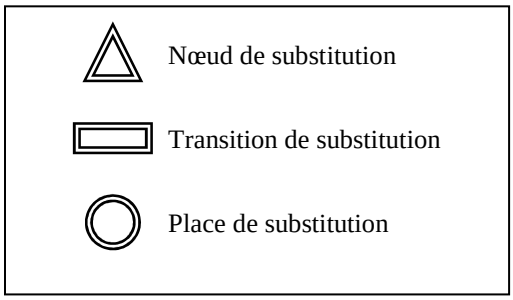
2-2 Enrichissement par substitution

Pendant la spécification d'un système complexe, il peut être convenable de pouvoir ignorer certaines parties du système en les *remplaçant* par des composants plus simples. Sous certaines conditions structurelles simples mais restrictives, un module $\widehat{\mathcal{E}}'$ peut être représenté, sous forme condensée, par un *nœud* particulier N dans le réseau du module $\widehat{\mathcal{E}}$ qui enrichit $\widehat{\mathcal{E}}'$.

Le rôle du noeud N (appelé *nœud de substitution*) est simplement de décrire comment le réseau du sous-module $\widehat{\mathcal{E}}'$ (correspondant à N) est inséré dans le module $\widehat{\mathcal{E}}$ (de même manière qu'en programmation, l'appel à une subroutine indique l'emplacement où son code sera inséré).

Un nœud de substitution représente donc un composant du système. A chaque nœud de substitution correspond un module qui contient la spécification détaillée du composant. La représentation du noeud de substitution dans le réseau de $\widehat{\mathcal{E}}$ permet de préciser la façon suivant laquelle sera intégré le réseau du module importé (correspondant au noeud de substitution) dans la spécification $\widehat{\mathcal{E}}$.
Les places (resp. les transitions) adjacentes à un noeud de substitution doivent être de type *interface* et correspondent à des places interfaces (resp. transitions interfaces) dans le module importé. Cette correspondance est établie via un morphisme appelé *morphisme de substitution*.

Dans le cas général, un nœud de substitution est représenté graphiquement par un triangle tel qu'il est schématisé par la Figure 5.3. Cette représentation peut être remplacée par celle d'une transition de substitution (resp. place de substitution) si tous les nœuds adjacents au nœud de substitution sont des places (resp. des transitions).



Figue 5.3 : Représentation graphique d'un nœud de substitution.

La notion de *nœud de substitution* définie dans CIRTA peut être vue comme une généralisation de la notion de *transition de substitution* et de *place de substitution* (définies dans le domaine des réseaux de Petri [JEN-92]) dans la mesure où un nœud de substitution peut être *adjacent à la fois* à des places et à des transitions.

La définition d'un module $\widehat{\mathcal{E}}$ enrichissant (par substitution) le module $\widehat{\mathcal{E}}'$ par $\Delta\mathcal{E}$, est réalisée au moyen de nœud de substitution *N* selon la Figure5.4.

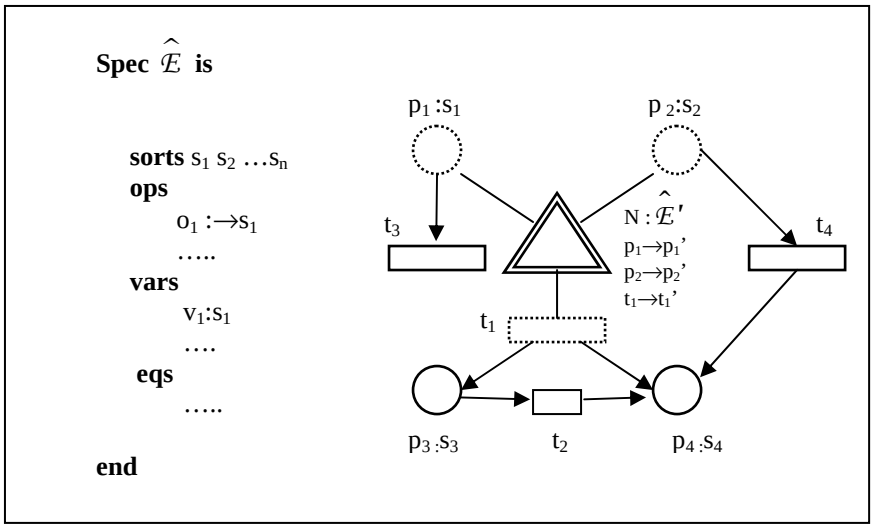


Figure 5.4: Principe de l'enrichissement par substitution.

Les inscriptions jointes au nœud de substitution précisent :

- L'identificateur du nœud (*N*)
- Le nom du module qui correspond à *N* (i.e. $\widehat{\mathcal{E}}'$).
- Le morphisme de substitution qui décrit la manière suivant laquelle le réseau du module enrichi sera inséré ($p_1 \rightarrow p_1'$,$p_2 \rightarrow p_2'$,$t_1 \rightarrow t_1'$). Les places {p_1', p_2'} et la transition t_1' sont définies dans $\widehat{\mathcal{E}}'$ et sont de type interface. Par défaut le morphisme de substitution est le morphisme identité.

La Figure 5.5 présente un exemple d'application des nœuds de substitution où la transition de substitution *t* dans $\widehat{\mathcal{E}}$ désigne le module $\widehat{\mathcal{E}}'$ et la place de substitution *p* dans $\widehat{\mathcal{E}}'$ correspond au module $\widehat{\mathcal{E}}''$.

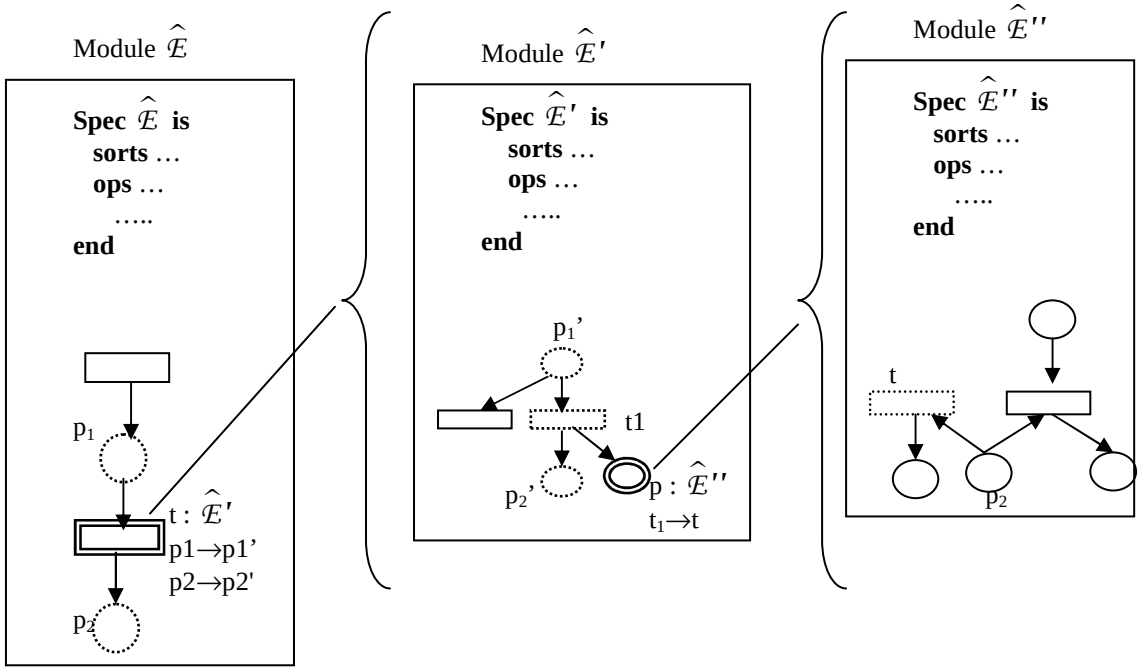


Figure 5.5 : Exemple d'enrichissements par substitution.

Il est évident qu'on peut transformer un module utilisant des nœuds de substitution en un module utilisant uniquement la primitive *use* (et vice versa). L'idée clef consiste (comme le précise la Figure 5.6) à identifier $\Delta\mathcal{E}$ comme étant la spécification déduite à partir du module $\widehat{\mathcal{E}}$ par :

- Renommage des places et des transitions adjacentes au nœud de substitution selon le morphisme précisé par le nœud de substitution, et
- Suppression du nœud de substitution.

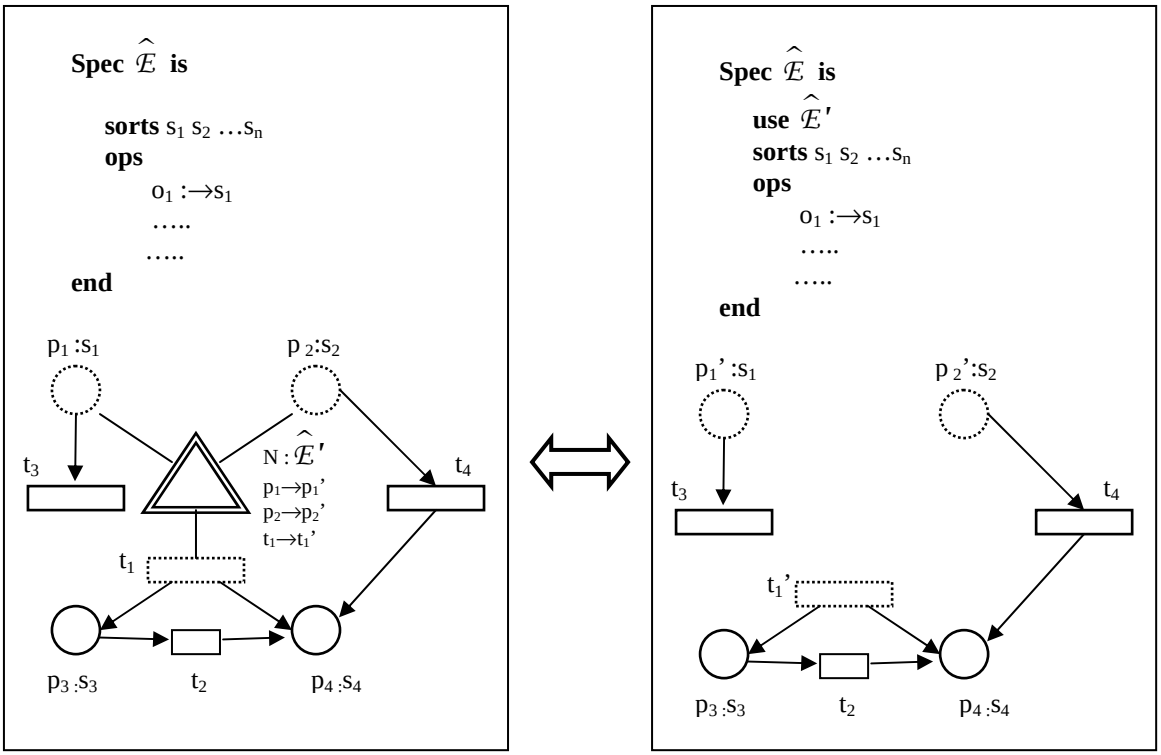


Figure 5.6 : Equivalence entre enrichissement par substitution
et enrichissement par duplication.

3- Exemple

Pour illustrer l'utilisation de la primitive *use* et des *nœuds de substitution*, considérons une chaîne de montage formée de 2 machines *M1* et *M2*. *M1* procède à la production de pièces finales (de sorte *p-final*) à partir de pièces initiales (de sortes *p-initial*). Pour chaque couple de pièces initiale (x,y) la machine *M1* produit la pièce finale *f(x,y).* La machine *M2* effectue l'emballage des pièces finales. Les deux machines communiquent via un buffer intermédiaire *B.* Les pièces initiales sont en stock représenté par la place *In* et les pièces finies et emballées sont stockées dans *Out*. Les produits d'emballage sont représentés par la place *E*. On décrit par le module Chaine-de-montage de la Figure 5.7, le système global sans détailler le fonctionnement interne de la machine *M1 et* en faisant abstraction de la spécification du type de pièces utilisées (*p-initial* et *p-final*). Le module de spécification *M1* décrit le fonctionnement de la machine *M1*. La spécification des types de pièces utilisées dans la chaîne de montage est donnée par le module *Pièce.*

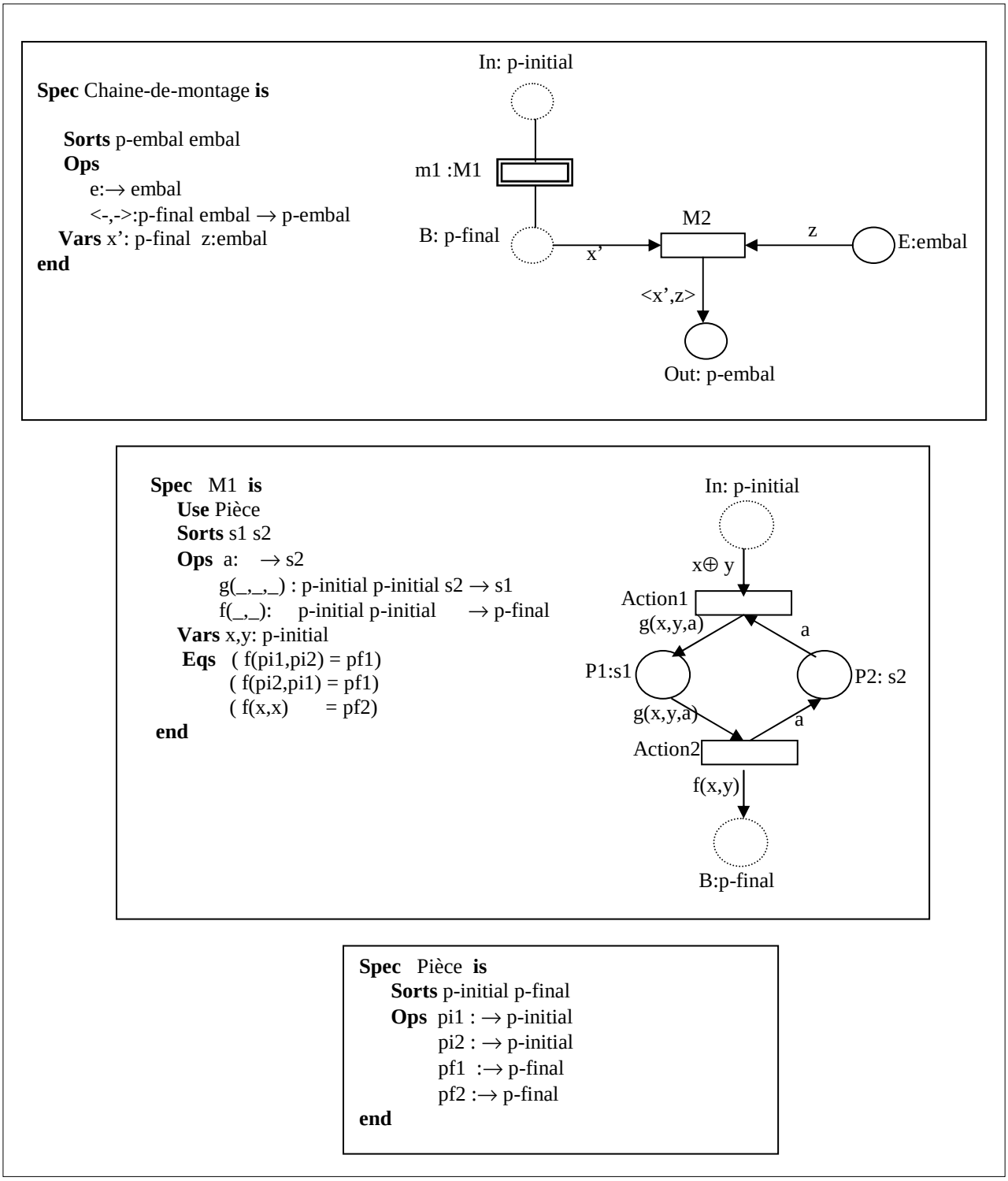


Figure 5.7 : Exemple d'enrichissement de modules.

La spécification Chaine-de-montage est alors équivalente à la spécification non structurée Flat-of-Chaine-de-montage décrite dans la Figure 5.8.

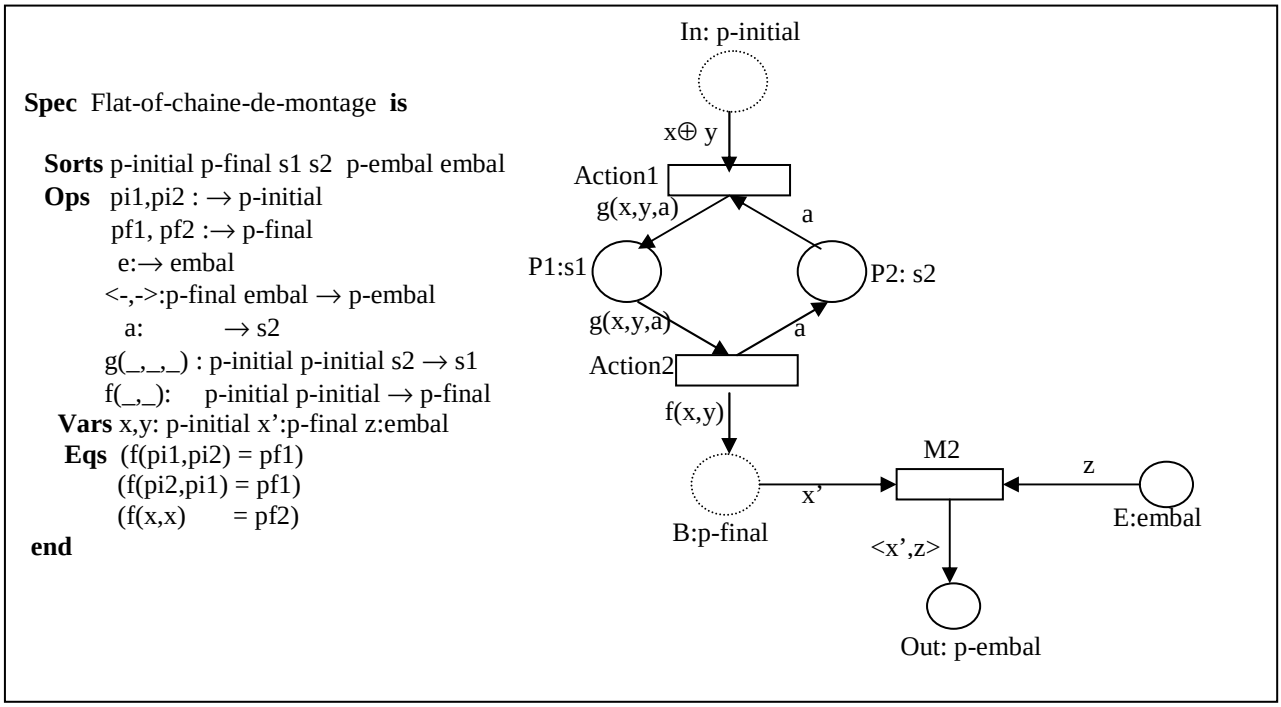


Figure 5.8 : Module Flat-of-Chaine-de-montage.

4- Syntaxe de l'enrichissement

4.1 Syntaxe abstraite

Il faut noter que quelque soit le type d'enrichissement (par substitution ou par duplication), enrichir un module consiste à construire un module par ajout d'éléments à un module préalablement défini.

En dépit de l'approche utilisée quelle soit descendante ou ascendante, l'enrichissement peut être abstraitement vu comme une opération invoquant trois arguments:

- L'identificateur de module construit par enrichissement: *id-module1*
- L'identificateur du module à enrichir: *id-module2*
- Une partie d'un module d'ECATNet $\Delta E = (((\Delta Spec, \Delta R), \Delta IS), \Delta I)$

En faisant abstraction des règles qui définissent la syntaxe abstraite de ΔE , la syntaxe abstraite de l'enrichissement est définie par la règle:

Enrichissement = (*id-module1, id-module2,* ΔE *)*

4.2 Syntaxe concrète

De manière générale, dans le langage CIRTA, un module peut enrichir non seulement un module mais une certaine combinaison de modules désignée par "une expression de module". L'expression de module désigne une spécification créée par une ou plusieurs compositions, renommages, instanciations des modules existants. La spécification ainsi obtenue n'est pas identifiée par le nom d'un module. Ces expressions de modules peuvent être complexes et invoquer une série de transformations des spécifications comparables aux expressions arithmétiques. Ainsi, on peut par exemple, enrichir un module renommé ou enrichir une instanciation de module paramétrée ou encore enrichir la composition de l'instanciation de deux modules paramétrés.

L'enrichissement dans CIRTA est défini par la syntaxe concrète suivante:

- **Syntaxe textuelle :**

<enrichissement> : := {**use, using**} <expression de module> <corps du module>

<expression de module> : := <expression élémentaire> ⟦ **+** <expression élémentaire> ⟧*

<expression élémentaire> : := <module simple> ⟦ ***(**<renommage>**)** ⟧

<module simple> : := <identificateur de module> ⟦ **[**<liste des paramètres réels>**]** ⟧

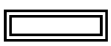
<liste de paramètres réels> : := <paramère réel> ⟦ , <paramètre réel> ⟧ *

<corps de module > : := ⟦ <déclaration de sortes> ⟧ *
⟦ <déclaration des sous-sortes> ⟧ *
⟦ <déclaration des opérations > ⟧ *
⟦ <déclaration des variables> ⟧ *
⟦ <déclaration des équations > ⟧ *
⟦ <déclaration de réseau> ⟧ *

- **Syntaxe graphique :**

<enrichissement> : :=  <inscription du nœud de substitution>|

 <inscription de place de substitution>|

 <inscription de transition de substitution >

<inscription du nœud de substitution> ::= <identificateur du noeud> **:**<expression de module>
⟦< morphisme de substitution >⟧*

<morphisme de substitution> ::= ⟦< morphisme élémentaire >⟧+

< morphisme élémentaire > ::= <morphisme de substitution de place>|
<morphisme de substitution de transition>

<morphisme de substitution de place> ::= **pl** <identificateur de place> → <identificateur> |
<identificateur de place> → <identificateur>

<morphisme de substitution de transition>::=
tr <identificateur de transition>→<identificateur> |
<identificateur de transition> → <identificateur>

<inscription de place de substitution> ::=
<identificateur de place de substitution> **:** <expression de module>
⟦<morphisme de substitution de transition >⟧*

<inscription de transition de substitution> ::=
<identificateur de transition de substitution> **:** <expression de module>
⟦<morphisme de substitution de place >⟧*

Notation :
<A> : symbole non terminal A.
A : symbole terminal (en caractères gras).
| : opérateur indiquant une alternative.
⟦ A ⟧ : zéro ou une occurrence de A.
⟦ A ⟧* : un nombre quelconque (nul ou non) d'occurrence de A.
⟦ A ⟧+ : au moins une occurrence de A.

III- Composition

La primitive d'enrichissement permet d'ajouter de nouvelles sortes, opérations équations et portions de réseau à un module préalablement défini. Les éléments ajoutés ne constituent pas en général *un module* CIRTA. La primitive de composition que nous allons présenter consiste à ajouter un *module* à un autre *module*. Elle peut être considérée comme l'enrichissement de chacun des deux modules par l'autre. Cette opération est commutative et elle est associative si elle est appliquée à plusieurs modules.

1- Principe de composition

La composition de deux modules construit un nouveau module par un ajout *sans redondance* des sortes, opérations, axiomes, places et transitions du premier module à ceux du deuxième. Cette opération entraîne une fusion de leurs objets visibles portant le même nom.

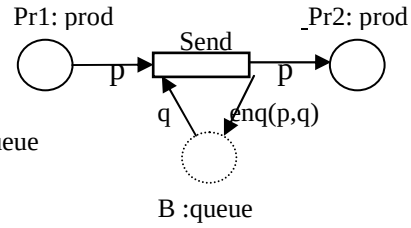
La construction d'un module $\widehat{\mathcal{E}}$ par composition des modules $\widehat{\mathcal{E}}_1$ et $\widehat{\mathcal{E}}_2$ est réalisée au moyen de la primitive ***combine*** qui est abstraitement notée par: $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}_1 \cup \widehat{\mathcal{E}}_2$.

2- Exemple

Considérons les modules $\widehat{\mathcal{E}}_1$ et $\widehat{\mathcal{E}}_2$ représentés dans la Figure5.9 et qui spécifient indépendamment et respectivement les producteurs et les consommateurs. La composition de $\widehat{\mathcal{E}}_1$ et $\widehat{\mathcal{E}}_2$ est spécifiée par le module $\widehat{\mathcal{E}}$ dont le module plat équivalent est Flat-of-$\widehat{\mathcal{E}}$ décrit dans la Figure 5.10.

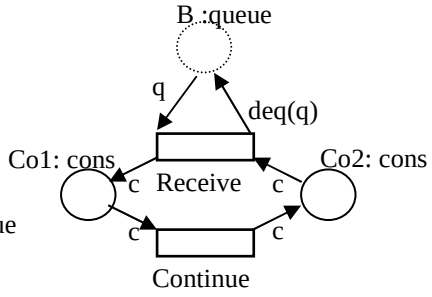
Spec $\widehat{\mathcal{E}}_1$

Sorts prod queue
Ops
$p_1 : \rightarrow$prod
$p_2 : \rightarrow$prod
nil : → queue
enq (_,_) : prod queue → queue
Vars p :prod q :queue
end



Spec $\widehat{\mathcal{E}}_2$ **is**

Sorts prod queue cons
Ops
$p_1 : \rightarrow$prod
$p_2 : \rightarrow$prod
c_1 : → cons
c_2 : → cons
nil : → queue
enq (_,_) : prod queue → queue
deq(_):queue : → queue
Vars x : prod q:queue c :cons
Eqs (deq(enq(x,q)) = q)
(deq(nil) = nil)
end



Spec $\widehat{\mathcal{E}}$ **is**
Combine $\widehat{\mathcal{E}}_1$ $\widehat{\mathcal{E}}_2$
end

Figure 5.9 : Exemple de composition de modules.

Spec Flat-of-$\widehat{\mathcal{E}}$
Sorts prod queue cons
Ops
$p_1 : \rightarrow$prod
$p_2 : \rightarrow$prod
c_1 : → cons
c_2 : → cons
nil : → queue
enq (_,_) : prod queue → queue
deq(_):queue : → queue
Vars x, p: prod q:queue c :cons
Eqs (deq(enq(x,q)) = q)
(deq(nil) = nil)
end

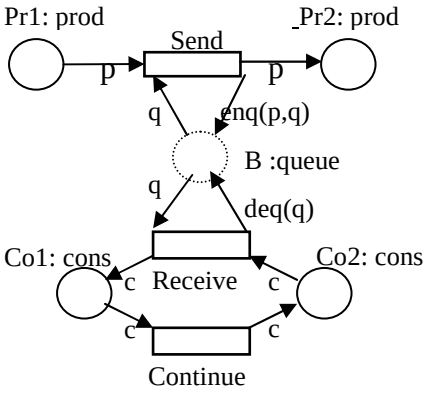


Figure 5.10: Module Flat-of-$\widehat{\mathcal{E}}$.

3- Syntaxe de la composition

3.1 Syntaxe abstraite

Deux facteurs interviennent lors de la composition des modules:
- L'identificateur du module qui résulte de la composition: *id-mod*
- La liste des identificateurs des modules à composer [*id-mod1, id-mod2,…*]

La syntaxe abstraite est donnée par la règle:

Composition = (*id-mod*, [*id-mod1, id-mod1,..*])

3.2 Syntaxe concrète

<Composition> ::= **combine** <expression de module> ⟦ <expression de module> ⟧$^{+}$ |
<expression de module> ⟦ **+** <expression de module> ⟧$^{+}$

IV- Renommage

Lorsque plusieurs modules de spécification participent à la spécification d'un même système, il est possible que des conflits d'identificateurs de sortes, d'opérations, de variables, de places ou de transitions se produisent. Ce problème surgit en particulier lorsque les modules ont été développés de façon relativement indépendante (proviennent par exemple d'une bibliothèque de modules préalablement définie).
Le renommage est une opération qui permet d'éviter les conflits d'identificateurs. Elle consiste à appliquer un *morphisme* (partiel ou total) sur l'ensemble des identificateurs des sortes, des opérations, des places, des transitions et des variables d'un module.

1- Principe du renommage

Dans le langage CIRTA, le renommage d'un module $\widehat{\mathcal{E}}$ consiste à *créer* un nouveau module $\widehat{\mathcal{E}}'$ à partir de $\widehat{\mathcal{E}}$ moyennant un changement de noms des sortes, opérations, variables, places ou transitions.

Le renommage ne concerne que les éléments visibles (donc accessibles) aux autres modules. Les éléments visibles d'un module sont : les sortes, les opérations, les places interfaces, les transitions interfaces et les variables(définies dans les contexte des transitions interfaces) ; qui n'ont pas été explicitement ou implicitement déclarées cachées par les primitives ***hide*** ou **export** (décrites dans le paragraphe V de ce chapitre)**.**

Le renommage est réalisé grâce à un morphisme dit *morphisme de renommage* ρ. Le principe de renommage d'un module $\widehat{\mathcal{E}}$ par un morphisme (partiel ou total) ρ (*où* $\rho(e_i) = e_i'$ pour chaque élément visible e_i de $\widehat{\mathcal{E}}$) est décrit par la Figure 5.11 .

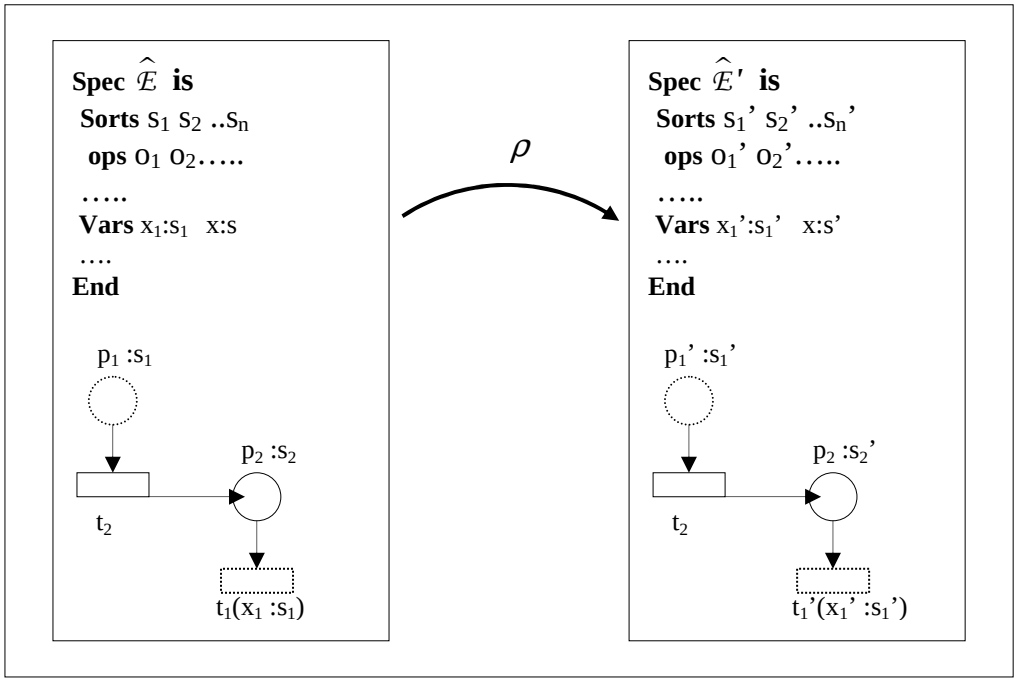


Figure 5.11 : Principe du renommage d'un module.

2- Types de renommages

2-1 Renommage des sortes

Quand une sorte est renommée, on renomme implicitement toutes ses apparitions dans le module à renommer : les profils des opérations, les sortes des places, les types de variables et les contextes des transitions.

2-2 Renommage des opérations

Le renommage d'une opération signifie que tous les termes qui l'utilisent sont renommés qu'ils soient des termes dans une équation de E, des termes intervenant dans les inscriptions du réseau (*IC, DT, CT, TC*) ou dans son marquage. Etant donné que CIRTA permet la surcharge des opérateurs (« overloading »), il est nécessaire de définir dans le morphisme le profil de l'opération à renommer. Si aucune confusion n'a lieu d'être, la forme de l'opération suffit dans le morphisme de renommage.

2-3- Renommage des places

Le renommage d'une place revient à changer son identificateur. Ceci peut être indispensable lors de la composition des modules car les places à fusionner doivent avoir le même identificateur dans les deux modules à composer.

2-4- Renommage des transitions

Le renommage d'une transition consiste à substituer d'une part l'identificateur de cette transition par un autre identificateur; et d'autre part, à changer éventuellement les identificateurs des variables de son contexte. Comme dans le cas de renommage d'une place, ceci peut s'avérer indispensable, lors de la composition des modules ou les transitions à fusionner doivent avoir le même identificateur dans les deux modules à composer, et des contextes compatibles. Généralement le renommage d'une transition ne concerne pas seulement son identificateur mais également les identificateurs des variables de son contexte.

3- Exemple

Le module Two-states-machine présenté dans la Figure 5.12 définit une machine à deux états par renommage du module One-philosophe (qui spécifie les états d'un philosophe).

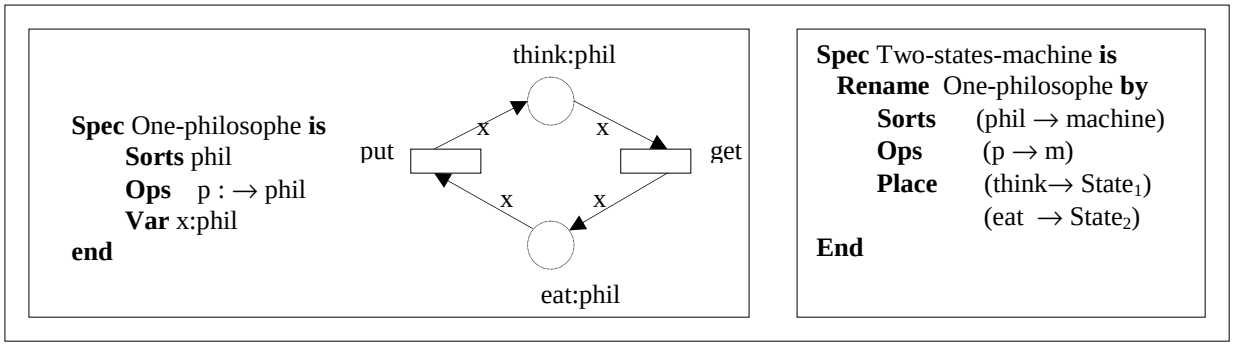


Figure 5.12 : Exemple de renommage d'un module.

La spécification plate de Two-state-machine est donnée par le module Flat-of-two-states-machine suivant.

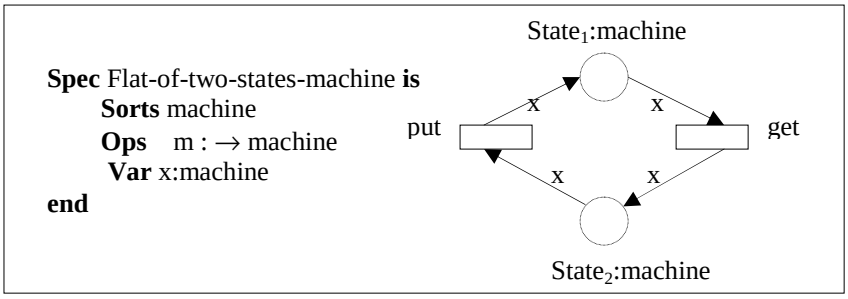


Figure 5.13 : Module Flat-of-two-states-machine.

Les transitions *put* et *get* sont invisibles de l'extérieur et correspondent aux actions de changement d'état. La machine à deux états spécifiée par Two-states-machine n'est visible que par ses états $State_1$ et $State_2$.

4- Syntaxe du renommage

4.1 Syntaxe abstraite

Un module construit par renommage d'un autre module est défini par le triplet:
- Identificateur du module construit par renommage: *id-module*$_1$.
- Identificateur du module à renommer: *id-module*$_2$.
- Le morphisme de renommage: *morphisme-de-renommage*.

Le morphisme de renommage est un ensemble de pair d'éléments de même classe (sortes, opérations, places, ou transitions.)

La syntaxe abstraite du renommage est donnée par les règles suivantes:

Module-renommage = (*id-module*$_1$, *id-module*$_2$, *morphisme-de-renommage*)
morphisme-de-renommage = [*couples-de-sortes* | *couples-de-opérations* | *couples-de-places* | *couples-de-transitions-contexte*]$^+$

4.2 Syntaxe concrète

Pour éviter toute confusion, il s'avère nécessaire de préciser le type de l'objet renommé (sorte, opération, place, transition) car le langage CIRTA permet par exemple qu'une sorte et une constante, ou une place et une transition aient le même identificateur. C'est pourquoi chaque objet renommé est précédé de son type (sort, ops, pl, tr) dans la définition du morphisme de renommage.

La syntaxe concrète de la primitive de renommage est définie par les règles suivantes:

<renommage> : := **Rename** <expression de module> **by**
<morphisme>
<morphisme> : := ⟦ <morphisme de renommage des sortes> ⟧ *
⟦ <morphisme de renommage des opérations> ⟧ *
⟦ <morphisme de renommage des places> ⟧ *
⟦ <morphisme de renommage des transitions> ⟧*

<morphisme de renommage des sortes> : := {**so, sort, sorts**}
⟦ **(**<identificateur de sorte> → <identificateur de sorte>**)** ⟧ $^+$

<morphisme de renommage des opérations> : :={**ops, opns**}
⟦ **(** <opérateur>→ <forme d'opération>**)** ⟧ $^+$

<morphisme de renommage des places> : := {**place, places**}
⟦ **(**<identificateur de place> → <identificateur de place>**)** ⟧ $^{+}$

<morphisme de renommage des transitions> : := {**transition, transitions**}
⟦ **(**<identificateur de transition et contexte>→
<identificateur de transition et contexte>**)** ⟧ $^{+}$

<identificateur de transition et contexte> : := <identificateur de transition> ⟦ **(**<contexte>**)** ⟧
<contexte> : := <identificateur de variable>**:** <identificateur de sorte>
⟦ **,** <identificateur de variable>**:** <identificateur de sorte> ⟧$^{+}$
<opérateur> : := <forme d'opération> ⟦ **:** <domaine> → <codomaine> ⟧

V- Contrôle de visibilité

Renommer un sous-ensemble des éléments (sortes, opérations, places, transitions) d'un module ne constitue pas toujours la solution adéquate pour résoudre un « conflit » d'identificateurs. En effet, le renommage ne diminue pas le nombre global des éléments visibles, et si la spécification est de taille importante (et fait intervenir un grand nombre de modules), le choix des nouveaux identificateurs doit être fait en dehors de tous les éléments courants. Par conséquent, il s'avère utile dans le cas des spécifications complexes, d'utiliser des identificateurs *locaux* dont la visibilité est limitée.

Dans CIRTA, on offre deux primitives de contrôle de visibilité : la primitive *Hide* et la primitive *Export* :

- La primitive *Hide* permet de définir les éléments qui ne seront plus visibles.
- La primitive *Export* permet de préciser les éléments qui seront visibles.

Les deux primitives opèrent de façon symétrique et le choix entre *Export* ou *Hide* dépend uniquement de la proportion d'identificateurs qui doivent rester visibles.

1-Principe du contrôle de visibilité

La définition d'un module $\widehat{\mathcal{E}}'$ par l'application d'une primitive de contrôle de visibilité sur un module $\widehat{\mathcal{E}}$ revient à appliquer sur $\widehat{\mathcal{E}}$ le morphisme h selon la Figure 5.14 où la notation $\not{e}$ signifie que l'élément e n'est plus visible.

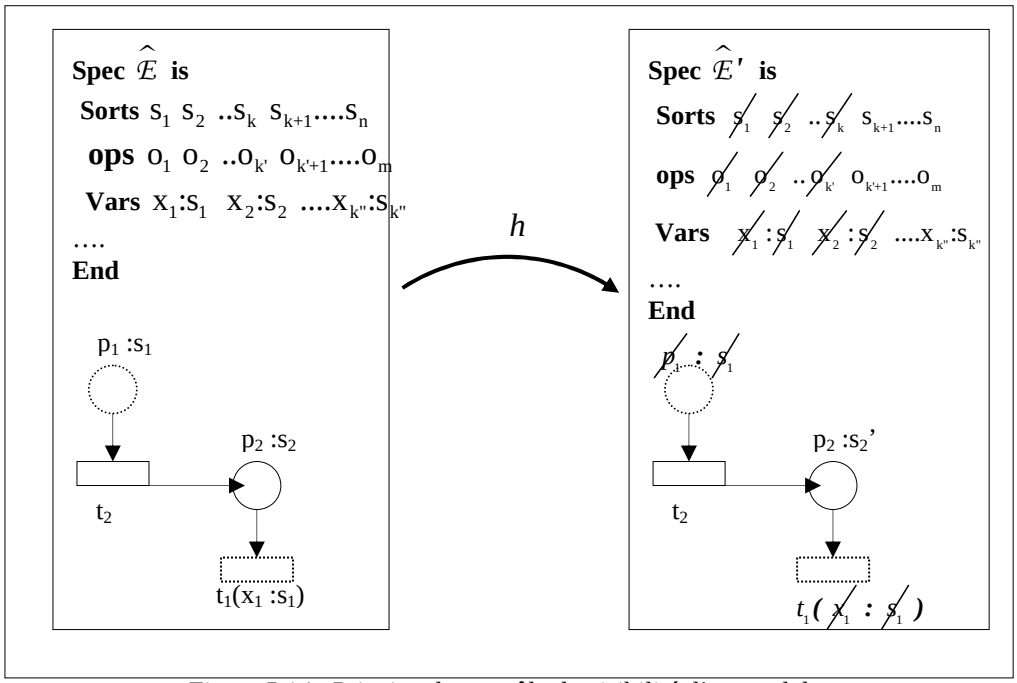


Figure 5.14 : Principe de contrôle de visibilité d'un module.

2- Types de contrôle de visibilité

2-1 Contrôle de visibilité des sortes

Une sorte *s* rendue non visible est inaccessible pour l'extérieur. Il en est implicitement de même pour toutes les opérations dans le profil desquelles figure cette sorte. Les places interfaces de sorte *s* deviennent non visibles ainsi que les variables de sorte s dans les contextes des transitions interfaces.

2-2 Contrôle de visibilité des opérations

Quand une opération devient non visible, les termes qui l'utilisent deviennent implicitement non visibles dans les cas suivants :
- Ces termes appartiennent au marquage d'une place interface visible.
-Ces termes apparaissent dans les inscriptions des arcs (*IC,DT,CT)* entre deux nœuds visibles .
- Ces termes sont utilisés pour définir la condition *TC* d'une transition interface visible.

Etant donné que CIRTA permet la surcharge des opérateurs, il est nécessaire de définir le profil de l'opération qui sera non visible. Si aucune confusion n'a lieu d'être, la forme de l'opération est suffisante.

2-3 Contrôle de visibilité des places

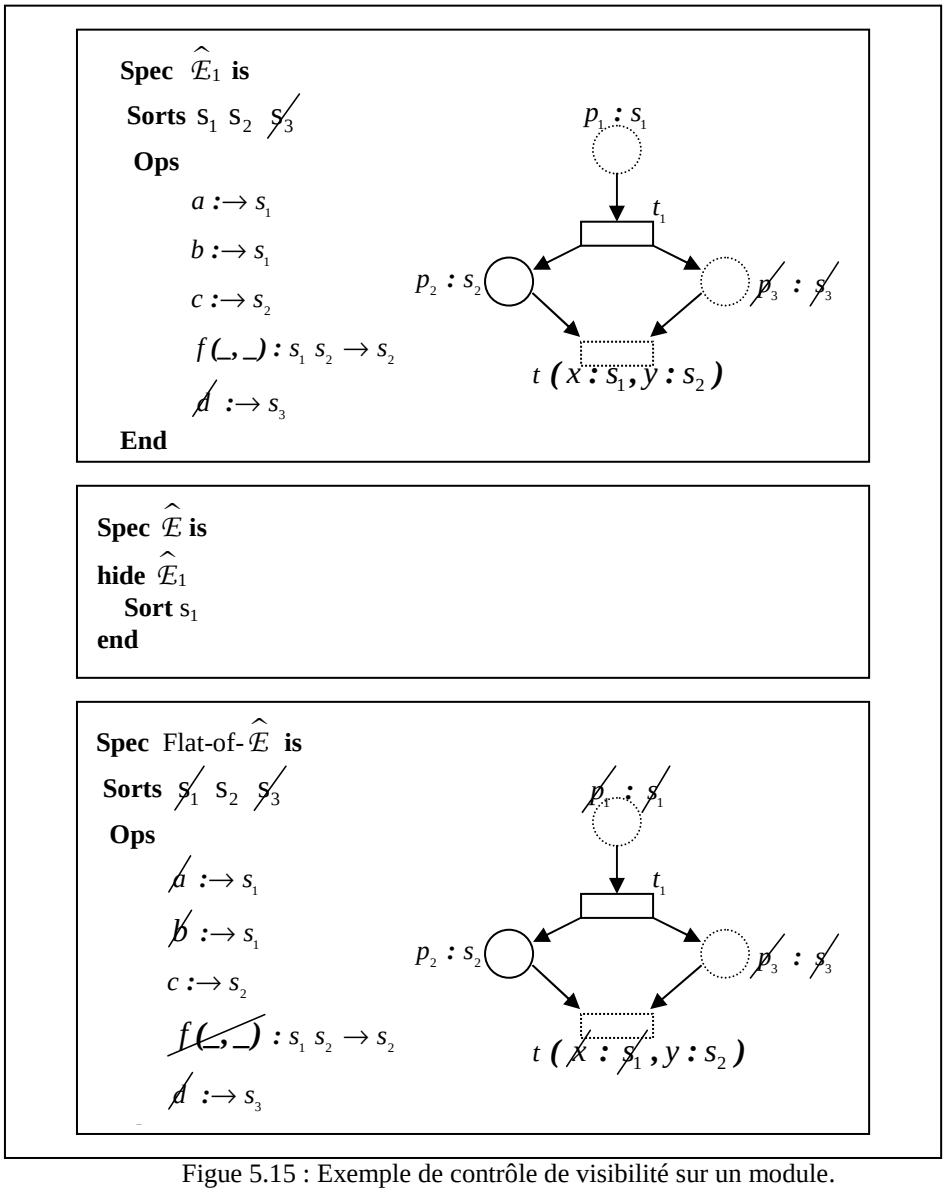
Une place interface rendue non visible est inaccessible pour son environnement. Ainsi si une place interface *p* est non visible dans un module $\widehat{\mathcal{E}}$, la composition de $\widehat{\mathcal{E}}$ et d'un module $\widehat{\mathcal{E}}'$ (ayant une place interface visible identifiée par *p*) résulte en un module ayant une place interface *p* et une place non visible *p*. Sémantiquement, il ne s'agit pas d'une même place et par conséquent elle ne peuvent être considérée comme équivalentes.

2-4 Contrôle de visibilité des transitions

De manière analogue aux places, une transition interface rendue non visible devient inaccessible pour son environnement. Il en résulte que son contexte devient inaccessible.

3- Exemple

Soit le module de spécification $\widehat{\mathcal{E}}_1$ représenté dans la Figure 5.15 et dans lequel la sorte s_3, l'opération *d* et la place p_3 sont non visibles. Le module $\widehat{\mathcal{E}}$ rend non visible la sorte s_1 de $\widehat{\mathcal{E}}_1$ et il est équivalent au module Flat-of-$\widehat{\mathcal{E}}$.



Figue 5.15 : Exemple de contrôle de visibilité sur un module.

4-Syntaxe du contrôle de visibilité

4.1 Syntaxe abstraite

Un module construit par l'application d'une primitive de visibilité sur un autre module est défini par le quadruplet :

-Le type de la primitive de visibilité (*hide/export*)
- L'identificateur du module construit par l'application de la primitive de visibilité: $id\text{-}module_1$
- L'identificateur du module sur lequel a été appliquée la primitive de visibilité : $id\text{-}module_2$
- Les identificateurs des éléments (sortes, opérations, places, transitions) concernées par la primitive de visibilité.

La syntaxe abstraite du contrôle de visibilité est donnée par les règles suivantes:

Visibilité = (*type-visibilité, id-module$_1$, id-module$_2$, visibilité-elem*)
type-visibilité= *hide* | *export*
Visibilité-elem = [*identificateurs-de-sortes* | *identificateurs-de-opérations* | *identificateurs-de-places* | *identificateurs-de-transitions*]$^+$

4.2 Syntaxe concrète

Comme dans le cas du renommage et pour éviter toute confusion, il s'avère nécessaire de préciser le type de l'objet concerné par la primitive de visibilité (sorte, opération, place, transition). Par conséquent, chaque objet est précédé de son type (sort, ops, pl, tr) dans la définition de la primitive de contrôle de visibilité.

La syntaxe concrète des primitives **hide** et **export** est définie par les règles suivantes:

<Visibilité> ::= <hiding> | <exporting>
<hiding>::= **hide**<expression-de-module><visibilité-elem>
<exporting> ::= **export** <expression-de-module><visibilité-elem>
<visibilité-elem> : := ⟦ <visibilité-sortes> ⟧*
⟦ < visibilité-opérations> ⟧*
⟦ < visibilité-places> ⟧*
⟦ < visibilité-transitions> ⟧*

< visibilité-sortes> : := {**sort, sorts**} ⟦ <identificateur de sorte> ⟧ $^+$
< visibilité-opérations> : :={**ops, opns**} ⟦ <opérateur> ⟧ $^+$
< visibilité-places> : := {**place, places**} ⟦ <identificateur de place> ⟧ $^+$
< visibilité-transitions> : := {**transition, transitions**} ⟦ <identificateur de transition > ⟧ $^+$

VI- Conclusion

- Nous avons présenté un ensemble de primitives du langage CIRTA permettant l'élaboration progressive des spécifications complexes selon l'approche *top-down* et/ou *bottom-up :*

  - Le système peut d'abord être spécifié globalement au niveau du module principal. Les détails peuvent être reportés à des modules sous-ordonnés qui peuvent eux aussi être subdivisés en sous-modules et ainsi de suite. De cette façon, nous obtenons une démarche de spécification descendante (*top-down*).
  - Les modules correspondants aux composants du système peuvent être spécifiés puis enrichis (ou composés) progressivement jusqu'à avoir la spécification globale du système. Ce qui permet une démarche de spécification ascendante (*'bottom-up').*

Ces deux approches, même si elles adoptent des démarches différentes, sont basées sur le principe de structuration et de modularité. Elles permettent d'appréhender la réalité selon les points de vue différents. En pratique, on aboutit généralement à une méthode hybride, qui construit une spécification en considérant aussi bien le système à spécifier que les modules de spécification préalablement définis.

- Les primitives définies dans le langage CIRTA peuvent être vues comme des extensions des primitives connues dans le domaine des spécifications algébriques et des mécanismes de structuration des réseaux de Petri. Elles constituent un ensemble fondamental d'opérations sur les modules de spécification CIRTA.

- Dans le chapitre suivant, nous présentons le mécanisme de *paramétrisation* qui enrichit cet ensemble de primitives fondamentales et favorise la définition de spécifications génériques. La sémantique formelle des différentes primitives du langage sera détaillée dans le chapitre9.

Chapitre 6

Mécanisme de paramétrisation

Sommaire

I- Introduction

Le concept d'enrichissement des modules de spécification, présenté dans le chapitre précédent est une primitive indispensable à tout langage de spécification. Un autre concept nullement moins important est celui de la paramétrisation des spécifications.
La paramétrisation d'une spécification permet de définir des systèmes génériques, et de ce fait elle évite de respécifier un module qui l'a été sous une forme légèrement différente. On introduit alors une liste de paramètres formels représentant d'autres spécifications dans la définition d'une spécification paramétrée. L'utilisation d'une spécification paramétrée pour définir la spécification d'un système se fera par l'instanciation de tout ou d'une partie de ses paramètres par d'autres modules de spécification paramétrés ou non.
Ainsi la paramétrisation s'avère être un moyen naturel de réutiliser aisément des spécifications et fait partie, comme l'enrichissement, des mécanismes indispensables à tout langage de spécification modulaire.
Dans le présent chapitre nous présentons le mécanisme de paramétrisation des spécifications dans le langage CIRTA. Il est inspiré d'une part du travail présenté dans [ZEG-96] sur la paramétrisation des réseaux de Petri ; et d'autre part des travaux effectués sur la paramétrisation des spécifications algébriques [GS-94][BID-89]. Nous montrons que la paramétrisation d'une spécification peut être de deux types différents mais non exclusifs: paramétrisation par les données (dite *paramétrisation statique*) et paramétrisation par le comportement (dite *paramétrisation dynamique*). Pour chacune de ces deux formes de paramétrisation, nous précisons les concepts de *module paramètre formel* et *module paramètre effectif*. Nous montrons également que pour chaque type de paramétrisation considéré, l'instanciation d'une spécification paramétrée peut être *partielle* ou *totale*. Elle peut être également *formelle* ou *effective*.

II- Principe de paramétrisation

Une spécification complexe $\widehat{E}$ est souvent vu comme une certaine composition d'un ensemble de sous-spécifications $\{\widehat{E}_1,\ldots, \widehat{E}_n\}$. Chaque sous-spécification $\widehat{E}_i$ décrit un composant bien défini du système spécifié. Les propriétés (comportementales ou structurelles) de $\widehat{E}$ se déduisent ainsi à partir de celles de l'ensemble $\{\widehat{E}_1,...,\widehat{E}_n\}$. Dans certaines spécifications, on peut autoriser qu'une sous-spécification $\widehat{E}_i$ (dite spécification *paramètre formel*) soit substituée par une autre spécification $\widehat{E}_i'$ (dite spécification *paramètre effectif*) de structure interne propre (plus complexe ou moins complexe que celle de $\widehat{E}_i$) mais vérifiant les propriétés minimales définies par $\widehat{E}_i$. Chaque substitution de $\widehat{E}_i$ dans $\widehat{E}$ donne une nouvelle spécification et on peut dire que $\widehat{E}$ est paramétré par $\widehat{E}_i$. L'utilisation de modules paramétrés dans la langage CIRTA permet de:

- Exprimer le fait qu'un sous-spécification $\widehat{E}_i$ de $\widehat{E}$ peut être remplacée par une autre spécification vérifiant les propriétés spécifiées par $\widehat{E}_i$.
- Offrir aux concepteurs la possibilité de spécifier par un seul module une classe de modules (toutes les instanciations du module paramétré).
- Eviter de revérifier certaines propriétés d'un module à chaque fois qu'on substitue l'un de ses sous-modules par un autre module (le module paramètre effectif).

A titre d'exemple, dans la spécification $\widehat{\mathcal{E}}$ (modélisée par son graphe des modules) ci-dessous, on admet que les sous-spécifications $\widehat{\mathcal{E}}_3$ et $\widehat{\mathcal{E}}_5$ soient substituées par d'autres sous-spécifications appartenant à l'ensemble des modules admissibles comme paramètres effectifs de $\widehat{\mathcal{E}}_i$ (i=3 ou i=5). Chaque substitution de $\widehat{\mathcal{E}}_i$ donne une nouvelle spécification $\widehat{\mathcal{E}}$; et $\widehat{\mathcal{E}}$ est dit paramétré par $\widehat{\mathcal{E}}_3$ et $\widehat{\mathcal{E}}_5$.

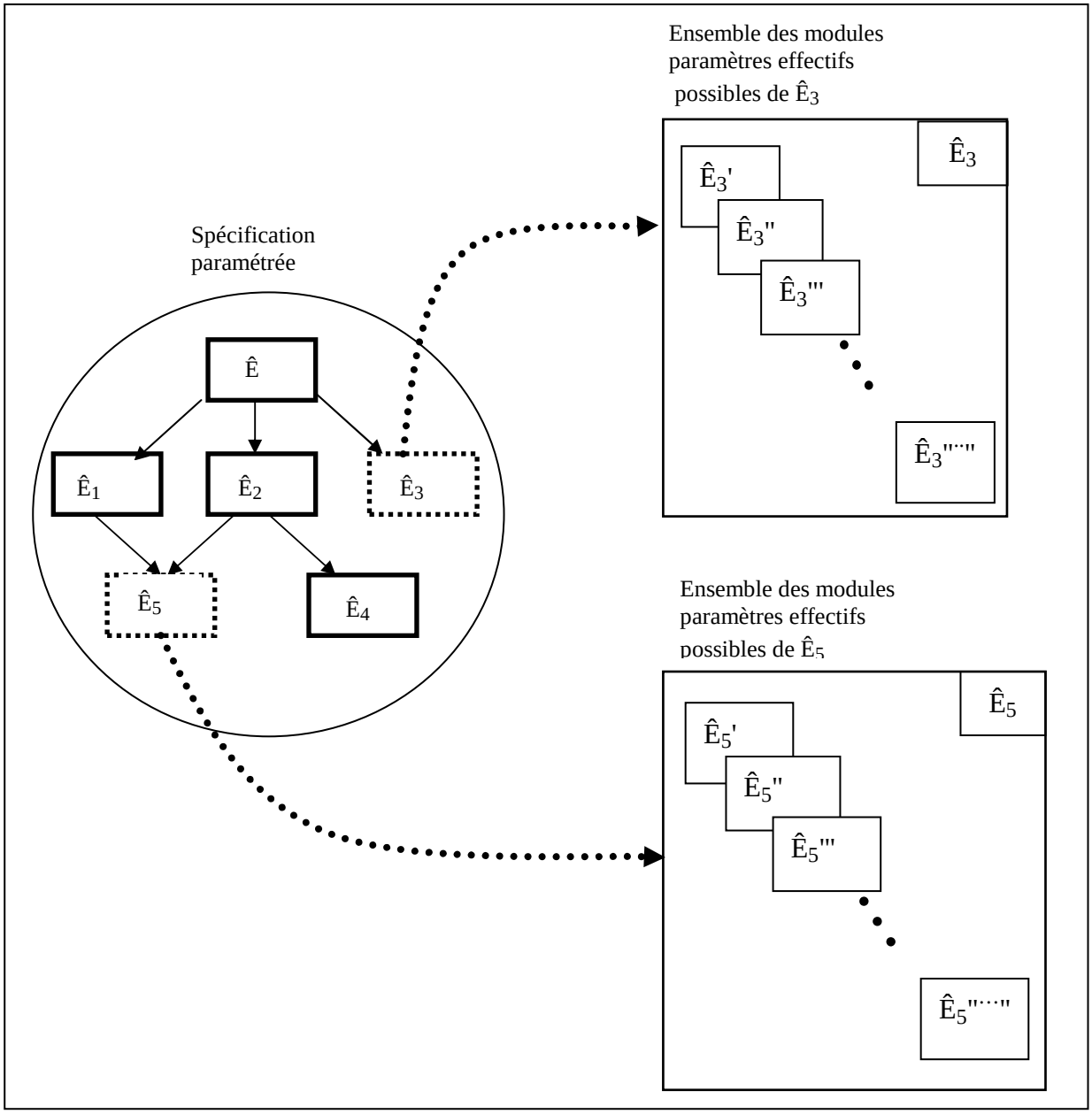


Figure 6.1 : Principe du mécanisme de paramétrisation.

La distinction entre les modules paramétrés (ex: $\hat{\mathcal{E}}$), les modules paramètres formels (ex: $\hat{\mathcal{E}}_3$ et $\hat{\mathcal{E}}_5$) et les modules ordinaires (non paramétrés et non paramètres formels) est reflétée dans le langage CIRTA au niveau de la syntaxe par l'utilisation de mots clé spécifique pour introduire chaque type de module:

- La spécification d'un module paramètre formel est introduite par le mot clé **Formal.**
- La spécification d'un module paramétré (générique) commence par le mot clé **Gen.**
- La spécification d'un module ordinaire débute par le mot clé **Spec.**

Une spécification paramétrée met en jeu trois composants:

- Un ensemble de modules de spécification paramètres formels $\{\hat{\mathcal{E}}_1,..,\hat{\mathcal{E}}_k\}$ dont le rôle est de spécifier les propriétés (statiques et dynamiques) minimales devant être satisfaites par les spécifications paramètres effectifs correspondants.
- Un ensemble de modules de spécification ordinaires $\{\hat{\mathcal{E}}_{k+1},..,\hat{\mathcal{E}}_n\}$ qu'elle requiert par enrichissement, composition, renommage ou contrôle de visibilité.
- La spécification paramétrée proprement dit: elle contient la spécification $\Delta\mathcal{E}$ des éléments (sortes, opérations, équations, réseaux) qui sont suffisamment généraux pour permettre la spécification de toutes les instances possibles. Cette partie de spécification ne constitue pas en général par elle-même un module de spécification.

Les spécifications paramètres formels peuvent être constituées d'un seul module (c'est le cas le plus fréquent car ces spécifications sont généralement très succinctes) ou structurées en plusieurs modules. Elles ne sont pas liées à une spécification paramétrée particulière mais peuvent être réutilisées dans différentes spécifications paramétrées.

Un module $\hat{\mathcal{E}}$ paramétré par $\{\hat{\mathcal{E}}_1,..,\hat{\mathcal{E}}_k\}$ et enrichissant $\{\hat{\mathcal{E}}_{k+1},..,\hat{\mathcal{E}}_n\}$ n>k par $\Delta\mathcal{E}$ est noté par $\hat{\mathcal{E}}\ [\hat{\mathcal{E}}_1,..,\hat{\mathcal{E}}_k]= \cup_{i=k+1..n}\hat{\mathcal{E}}_i + \Delta\mathcal{E}$ et il est schématisé par la Figure 6.2.

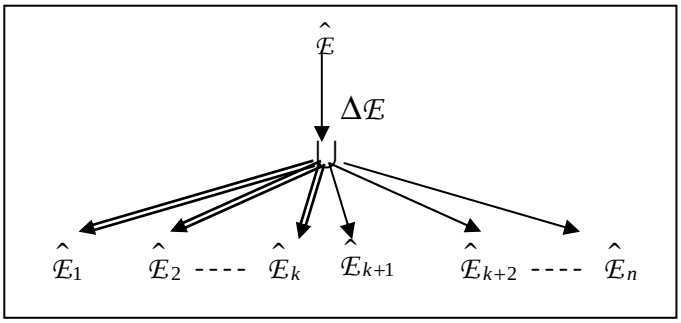


Figure 6. 2: Graphe des modules associé à un module générique.

Le mécanisme de paramétrisation est basé d'une part sur la qualité des propriétés minimales pouvant être spécifiées par un paramètre formel; et d'autre part sur les types d'instanciations possibles.

III- Principe de l'instanciation

Le processus d'instanciation d'un module paramétré peut être conçu comme la succession de deux étapes:

i) La vérification de l'adéquation entre paramètre formel et paramètre effectif.
ii) La substitution du paramètre effectif au paramètre formel.

1- Vérification de l'adéquation entre paramètre formel et paramètre effectif

Dans un premier temps, on décrit la manière suivant laquelle le paramètre effectif vérifie les propriétés minimales spécifiées par le paramètre formel. Cette description est réalisée grâce à un morphisme (appelé "*morphisme d'adéquation*") qui explicite les correspondances entre les sortes, les opérations, les places et les transitions du paramètre formels; et celles du paramètre effectif en précisant exactement quel élément du paramètre effectif correspond à quel élément du paramètre formel.

Ce morphisme d'adéquation est nécessaire pour les raisons suivantes:

- Les sortes, les opérations, les places et les transitions du paramètre effectif et celles du paramètre formel n'ont pas en général les mêmes identificateurs (dans le cas contraire il existe au moins un morphisme implicite)
- Il peut exister plusieurs morphismes possibles pour lesquels un paramètre effectif vérifie un paramètre formel.

Le morphisme d'adéquation entre paramètre formel et paramètre effectif est réalisé dans le langage CIRTA par des spécifications dites *"view"* qui explicite sous quel *point de vue* un paramètre effectif est valide.

La forme générale d'un morphisme est définie par les règles syntaxiques suivantes:

<morphisme>: := **view** <identificateur de view> **is**
<identificateur de paramètre formel> → <identificateur de paramètre effectif>
⟦ <morphisme de sortes> ⟧*
⟦ <morphisme d'opérations> ⟧*
⟦ <morphisme de places> ⟧*
⟦ <morphisme de transitions> ⟧*
end

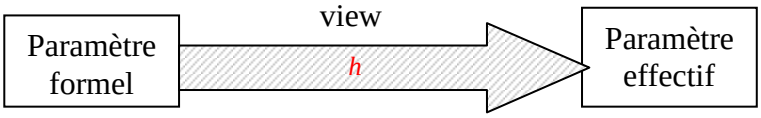
<morphisme de sortes> : := { **sort, sorts**}
⟦ **(**<identificateur de sorte paramètre formel> →
<identificateur de sorte paramètre effectif>**)** ⟧+

<morphisme d'opérations> : :={**ops, opns**}
⟦ **(** <opérateur paramètre formel > →
<forme d'opération paramètre effectif >**)** ⟧+

<morphisme de places> : := {**place, places**}
⟦ **(**<identificateur de place paramètre formel > → <identificateur de place paramètre effectif >**)** ⟧ $^{+}$

<morphisme de transitions> : := {**transition, transitions**}
⟦ **(**<identificateur de transition paramètre formel > → <identificateur de transition paramètre effectif >**)** ⟧ $^{+}$

<identificateur de sorte paramètre formel> : := <identificateur de sorte>
<identificateur de sorte paramètre effectif> : := <identificateur de sorte>
<opérateur paramètre formel> : := <opérateur>
<forme d'opération paramètre effectif> : := <forme d'opération>
<identificateur de place paramètre formel> : := <identificateur de place>
<identificateur de place paramètre effectif> : := <identificateur de place>
<identificateur de transition paramètre formel> : := <identificateur de transition>
<identificateur de transition paramètre effectif> : := <identificateur de transition>
<opérateur> : := <forme d'opération>⟦ **:** <domaine> -> <codomaine> ⟧
<identificateur de transition> : := <identificateur> ⟦ **(** <contexte> **)** ⟧
<contexte >::= ⟦ <identificateur de variable>**:** <identificateur de sorte> ⟧$^{+}$

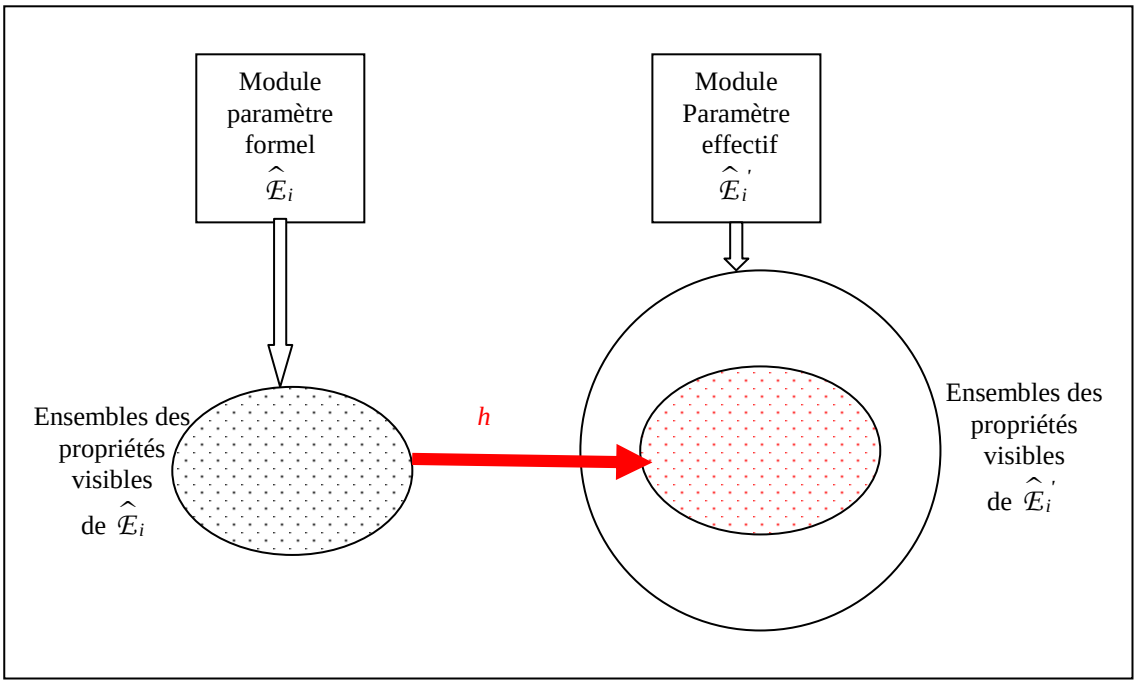


Informellement, un module $\widehat{\mathcal{E}'_i}$ est un paramètre effectif valide de $\hat{\mathcal{E}}_i$ selon un morphisme d'adéquation h_i si est seulement si :

- A chaque sorte visible de $\hat{\mathcal{E}}_i$ correspond une sorte visible de $\widehat{\mathcal{E}'_i}$
  $\forall s \in Sort(\hat{\mathcal{E}}_i) \cap \Sigma_v(\hat{\mathcal{E}}_i)\ \exists s' \in Sort(\hat{\mathcal{E}}_i') \cap \Sigma_v(\hat{\mathcal{E}}_i')\ /\!/\ \ h_i(s) = s'$
- A chaque opération visible de $\hat{\mathcal{E}}_i$ correspond une opération visible de $\widehat{\mathcal{E}'_i}$
  $\forall f \in Op(\hat{\mathcal{E}}_i) \cap \Sigma_v(\hat{\mathcal{E}}_i)\ \exists f' \in Op(\hat{\mathcal{E}}_i') \cap \Sigma_v(\hat{\mathcal{E}}_i')\ /\!/\ \ h_i(f) = f'$
- A chaque place visible de $\hat{\mathcal{E}}_i$ correspond une place visible de $\widehat{\mathcal{E}'_i}$
  $\forall p \in P_v(\hat{\mathcal{E}}_i)\ \ \exists p' \in P_v(\hat{\mathcal{E}}_i')\ /\ \ h_i(p) = p'$
- A chaque transition visible de $\hat{\mathcal{E}}_i$ correspond une transition visible de $\widehat{\mathcal{E}'_i}$
  $\forall t \in T_v(\hat{\mathcal{E}}_i)\ \ \exists t' \in T_v(\hat{\mathcal{E}}_i')\ /\ \ h_i(t) = t'$
- Toutes les propriétés visibles (statiques et comportementales) de $\hat{\mathcal{E}}_i$ sont des propriétés (au renommage prés via h_i) visibles et valides dans $\widehat{\mathcal{E}'_i}$ comme le décrit le schéma ci-dessous.

Notation :

$Sort(\hat{\mathcal{E}})$ et $Op(\hat{\mathcal{E}})$ désignent respectivement l'ensemble global des sortes, et l'ensemble global des opérations du module $\hat{\mathcal{E}}$. $\Sigma_v(\hat{\mathcal{E}})$ est la signature visible de $\hat{\mathcal{E}}$. $P_v(\hat{\mathcal{E}})$ et $T_v(\hat{\mathcal{E}})$ sont respectivement l'ensemble global des places visibles et l'ensemble global des transitions visibles du module $\hat{\mathcal{E}}$.



Le terme « *propriété* » utilisé désigne ici une formule *valide* (vérifiée par le module). Cette notion sera explicitée de façon plus détaillée dans les paragraphes IV-2 et V-2 de ce chapitre, puis elle sera détaillée de façon plus formelle au chapitre 9.

2- Substitution du paramètre effectif au paramètre formel

La seconde étape de l'instanciation consiste à remplacer la spécification paramètre formel par la spécification paramètre effectif. Ainsi on peut considérer la spécification résultante du processus d'instanciation comme une "copie" de la spécification paramétrée et toute référence à la spécification formelle (et à ses éléments) a été remplacée par un enrichissement de spécification paramètre effectif.

Soient $\hat{\mathcal{E}}$: un module paramétré par un module paramètre formel $\hat{\mathcal{E}}_i$
$\hat{\mathcal{E}}'_i$: un module paramètre effectif valide de $\hat{\mathcal{E}}_i$ par le morphisme h_i

L'instanciation de $\hat{\mathcal{E}}$ par $\hat{\mathcal{E}}'_i$ consiste à générer un module $\hat{\mathcal{E}}'$ tel que:

- *Spec*$(\hat{\mathcal{E}}')$ est l'instanciation de *Spec*$(\hat{\mathcal{E}})$ par *Spec*$(\hat{\mathcal{E}}_i)$ selon la formalisme des spécifications algébriques
- R_0 est un réseau obtenu en renommant dans $R(\hat{\mathcal{E}})$ les termes algébriques, les sortes, les opérations les places et les transitions par leurs correspondant du paramètre effectif $\hat{\mathcal{E}}'_i$ suivant le morphisme d'adéquation h_i
- $R(\hat{\mathcal{E}}')$ est obtenu en substituant dans R_0 , $R(\hat{\mathcal{E}}_i)$ par $R(\hat{\mathcal{E}}'_i)$.

Par exemple, une instanciation possible du module $\hat{E}$ décrit dans la Figure 6.1, pourrait être alors le module défini par spécification $\hat{E}'$ dont le graphe des modules est représenté par la Figure 6.3.

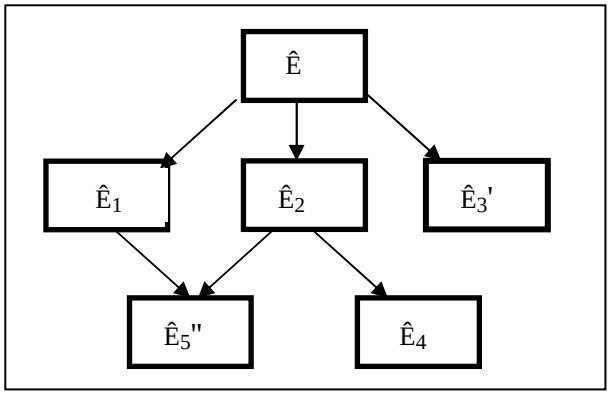


Figure 6.3 : Principe d'instanciation d'un module paramétré.

3- Types d'instanciation

Dans le langage CIRTA, l'instanciation d'un module paramétré peut concerner une partie et pas la totalité de l'ensemble des paramètres formels. Quand ceci se produit on parle d'instanciation *partielle* (sinon elle est dite *totale*). Par ailleurs, lorsque le paramètre formel est substitué par un autre module paramètre formel, l'instanciation est qualifiée de *formelle* (sinon elle est *effective*).

3-1 Instanciation effective partielle

L'instanciation effective partielle d'un module générique $\hat{E}\ [\hat{E}_1,..,\ \hat{E}_k]$ consiste à instancier par des modules *ordinaires* (non paramétrés et non paramètres formels) un *sous-ensemble* des modules paramètres formels:

Soient $\hat{E}\ [\hat{E}_1,..,\ \hat{E}_k]$ un module paramétré,

$\forall i=1...p \quad p < k \qquad \hat{E}'_i$ paramètre effectif valide de $\hat{E}_i$

L'instanciation partielle et effective de $\hat{E}$ par les modules $\hat{E}_i$ donne le module *paramétré* $\hat{E}'\left[\hat{E}_{p+1},...,\hat{E}_k\right]$

3-2 Instanciation effective totale

L'instanciation effective totale d'un module générique $\hat{E}\ [\hat{E}_1,..,\ \hat{E}_k]$ consiste à instancier par des modules *ordinaires tous les modules paramètres formels.*

Soient $\hat{E}\ [\hat{E}_1,..,\ \hat{E}_k]$ un module paramétré

$\forall i=1..k \qquad \hat{E}'_i$ paramètre effectif valide de $\hat{E}_i$

L'instanciation effective totale de $\hat{E}$ par les modules $\hat{E}'_i$ donne un module *ordinaire* $\hat{E}'$.

3-3 Instanciation formelle partielle

L'instanciation formelle partielle d'un module générique $\hat{\mathcal{E}}\ [\hat{\mathcal{E}}_1,..,\ \hat{\mathcal{E}}_k]$ consiste à instancier par des modules *paramètres formels* (non paramétrés) un *sous-ensemble* des modules paramètres formels de $\hat{\mathcal{E}}$.

Soient $\hat{\mathcal{E}}\ [\hat{\mathcal{E}}_1,..,\ \hat{\mathcal{E}}_k]$ un module paramétré

$\forall i=1..p \quad p < k \quad \hat{\mathcal{E}}'_i$ est un module de type formel et qui est un paramètre effectif valide de $\hat{\mathcal{E}}_i$

L'instanciation formelle partielle de $\hat{\mathcal{E}}$ par les modules $\hat{\mathcal{E}}'_i$ donne un module *paramétré* $\hat{\mathcal{E}}'\left[\widehat{\mathcal{E}'}_1,...,\widehat{\mathcal{E}'}_p,\hat{\mathcal{E}}_{p+1},...,\hat{\mathcal{E}}_k\right]$.

3-4 Instanciation formelle totale

L'instanciation formelle totale d'un module générique $\hat{\mathcal{E}}\ [\hat{\mathcal{E}}_1,..,\ \hat{\mathcal{E}}_k]$ consiste à instancier par des modules *paramètres formels* (non paramétrés) *tous* les modules paramètres formels de $\hat{\mathcal{E}}$.

Soient $\hat{\mathcal{E}}\ [\hat{\mathcal{E}}_1,..,\ \hat{\mathcal{E}}_k]$ un module paramétré

$\forall i=1..k \quad \hat{\mathcal{E}}'_i$ est un module de type formel et qui est un paramètre effectif valide de $\hat{\mathcal{E}}_i$

L'instanciation formelle totale de $\hat{\mathcal{E}}$ par les modules $\hat{\mathcal{E}}'_i$ donne le module *paramétré* $\hat{\mathcal{E}}'\left[\widehat{\mathcal{E}'}_1,...,\widehat{\mathcal{E}'}_k\right]$.

Ces types d'instanciation sont enrichis par différents types de paramétrisation, qui dépendent en particulier de la qualité des propriétés décrites par le paramètre formel.
Nous définissons, dans le langage CIRTA, deux types de paramétrisations: la paramétrisation par les données (dite *paramétrisation statique*) et la paramétrisation par le comportement (dite *paramétrisation dynamique*). Pour chacun de ces types de paramétrisation, l'instanciation peut être partielle ou totale, et effective ou formelle.

IV- Paramétrisation statique

Un module est paramétré *statiquement* si et seulement si tous ses paramètres formels sont des modules ayant chacun *un réseau vide*. Les contraintes spécifiées par le paramètre formel concernent dans ce cas les propriétés statiques (relatives aux structures de données) et non comportementales du système spécifié.
La paramétrisation statique permet de spécifier par un module paramétré un ensemble de systèmes qui ont la même structure de contrôle (le "même" réseau) mais manipulent des données "différentes". La différence entre ces structures de données réside d'une part dans l'ensemble des sortes et opérations définies en plus de celles requise par le paramètre formel et d'autre part dans l'ensemble des propriétés des différentes opérations.

1- Paramètre formel

Le but de la spécification paramètre formel est de préciser les propriétés minimales requises pour qu'une spécification paramètre effectif soit un paramètre admissible. Ces propriétés minimales requises peuvent concerner la définition des sortes, des opérations ou des propriétés que doivent satisfaire ces opérations. Par conséquent, dans ce type de paramétrisation, le paramètre formel se réduit à une *spécification algébrique.*
Ce type de paramétrisation est inspiré du domaine des spécifications algébriques. C'est également le type de paramétrisation généralement connu dans les formalismes combinant les réseaux de Petri et les spécifications algébriques [KRA-87] [BG-91a].

Les paramètres formels $\widehat{\mathcal{E}}_i$ d'un module paramétré statiquement sont des modules tels que $P(\widehat{\mathcal{E}}_i) = T(\widehat{\mathcal{E}}_i) = \varnothing$ (où $P(\widehat{\mathcal{E}}_i)$ *et* $T(\widehat{\mathcal{E}}_i)$ sont respectivement l'ensemble global des places et l'ensemble global des transitions du module structuré $\widehat{\mathcal{E}}_i$).

Exemple:

Le module ELEMENT de la Figure 6.4 est un paramètre formel du module paramétré BUFFER. Toute spécification algébrique ayant au moins une sorte (*elem*) est un paramètre effectif valide du module générique BUFFER. Ainsi il sera possible de générer par exemple les spécifications relatives aux buffers d'entiers, buffer de booléens, buffer de listes d'entiers, ou encore des buffers de listes de listes d'entiers.

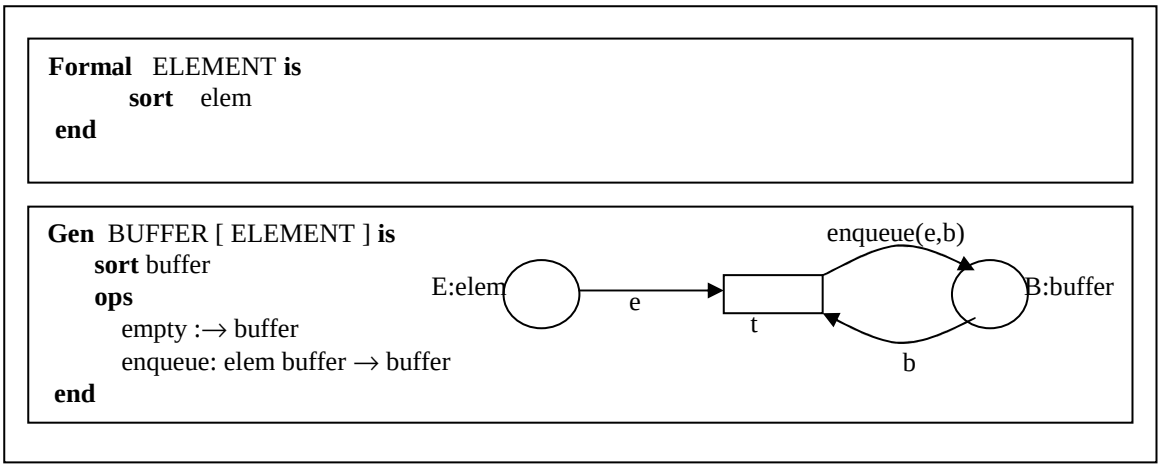


Figure 6.4 : Exemple de paramétrisation statique.

2- Paramètre effectif

Dans le cas de la paramétrisation statique, le paramètre effectif est un module de *spécification algébrique* (structurée ou non). Tous les éléments (les sortes et les opérations) de cette spécification doivent être *visibles*.

Définition:

Soit $\hat{E}_i$ un module paramètre formel tel que $P(\hat{E}_i) = T(\hat{E}_i) = \varnothing$. Un module $\hat{E}_i'$ est un paramètre effectif valide via le morphisme d'adéquation h_i si et seulement si les propriétés suivantes sont vérifiées:

- A chaque sorte de $\hat{E}_i$ correspond une sorte visible de $\hat{E}'_i$
  $\forall s \in Sort(\hat{E}_i)\ \exists s' \in Sort(\hat{E}_i') \cap \Sigma_v(\hat{E}_i')\ //\ \ h_i(s) = s'$
- A chaque opération visible de $\hat{E}_i$ correspond une opération visible de $\hat{E}'_i$
  $\forall f \in Op(\hat{E}_i)\ \exists f' \in Op(\hat{E}_i') \cap \Sigma_v(\hat{E}_i')\ //\ \ h_i(f) = f'$
- $\hat{E}_i'$ a un réseau vide $P(\hat{E}_i') = T(\hat{E}_i') = \varnothing$
- Toutes les propriétés statiques de $\hat{E}_i$ sont des propriétés (au renommage prés via h_i) visibles et valides dans $\hat{E}'_i$:
  $\forall \alpha, \beta \in T_{\Sigma(\hat{E}_i)}(V)$

  $$\alpha =_{E(\hat{E}_i)} \beta \ \Rightarrow \exists \alpha', \beta' \in T_{\Sigma_v(\hat{E}_i')}(V)\ /\ \left(h_i(\alpha) = \alpha'\right) \wedge \left(h_i(\beta) = \beta'\right) \wedge \left(\alpha' =_{E(\hat{E}_i')} \beta'\right)$$

Exemple:

Les exemples ci-dessous sont des paramètres effectifs valides pour ELEMENT puisqu'ils définissent chacun au moins une sorte qui correspond à la sorte *elem* par les morphismes respectifs *morph1* et *morph2*.

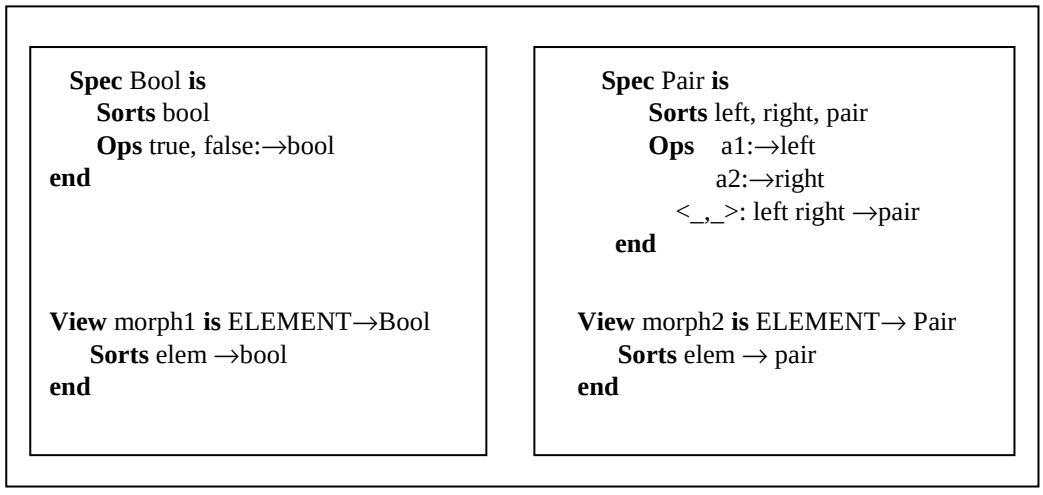


Figure 6.5: Exemples de paramètres effectifs valides d'un paramètre formel dans une paramétrisation statique.

Le morphisme *morph1* est évident alors que *morph2* n'est pas l'unique morphisme d'adéquation possible entre ELEMENT et PAIR (car la sorte *elem* peut aussi correspondre par d'autres morphismes à la sorte *left* ou la sorte *right*).

V- Paramétrisation dynamique

Un module paramétré par le comportement (paramétrisation dynamique) engendre la spécification d'une classe de systèmes qui opèrent sur les mêmes structures de données (au niveau de leur interface) et qui ont des comportements comparables: chaque paramètre formel décrit un comportement minimal requis et toutes les instances de ce paramètre sont caractérisées par un comportement minimal commun.

1- Paramètre formel

Dans le cas de la paramétrisation dynamique, le module paramètre formel décrit un comportement et par conséquent *ne doit pas avoir un réseau vide*. Les inscriptions de ce réseau faisant référence à des termes algébriques, la spécification algébrique ne peut être également réduite à la spécification algébrique vide.

Exemple:
Le module *MACHINE1* présenté dans la Figure 6.6 spécifie un système qui effectue pour chaque élément identifié par e, la liste des opérations qui lui est associée. Les spécification *NAT* des entiers naturels et *IDENT* des identificateurs sont supposées prédéfinies.

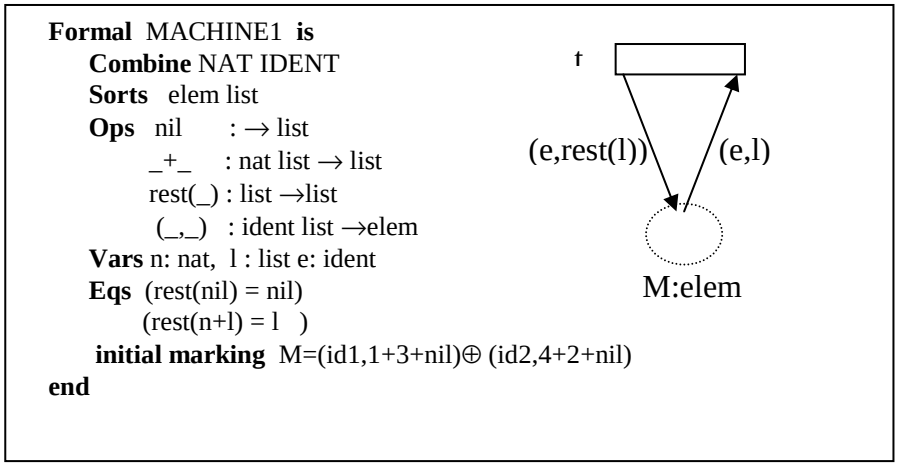


Figure 6.6 : Exemple de paramètre formel dans une paramétrisation dynamique.

2- Paramètre effectif

Dans le cas de la paramétisation dynamique, un module $\widehat{\mathcal{E}}_i'$ est un paramètre valide pour le paramètre formel $\widehat{\mathcal{E}}_i$ via le morphisme d'adéquation h_i si et seulement si les conditions suivantes sont vérifiées simultanément:

a) A chaque sorte visible de $\hat{\mathcal{E}}_i$ correspond une sorte visible de $\widehat{\mathcal{E}'}_i$

$$\forall s \in Sort(\widehat{\mathcal{E}}_i) \cap \Sigma_v(\widehat{\mathcal{E}}_i)\ \exists s' \in Sort(\widehat{\mathcal{E}}_i') \cap \Sigma_v(\widehat{\mathcal{E}}_i')\ /\!/\ \ h_i(s) = s'$$

b) A chaque opération visible de $\hat{\mathcal{E}}_i$ correspond une opération visible de $\widehat{\mathcal{E}'}_i$

$$\forall f \in Op(\widehat{\mathcal{E}}_i) \cap \Sigma_v(\widehat{\mathcal{E}}_i)\ \exists f' \in Op(\widehat{\mathcal{E}}_i') \cap \Sigma_v(\widehat{\mathcal{E}}_i')\ /\!/\ \ h_i(f) = f'$$

c) A chaque place visible de $\hat{\mathcal{E}}_i$ correspond une place visible de $\widehat{\mathcal{E}'}_i$

$$\forall p \in P_v(\widehat{\mathcal{E}}_i)\ \ \exists p' \in P_v(\widehat{\mathcal{E}}_i')\ /\ \ h_i(p) = p'$$

d) A chaque transition visible de $\hat{\mathcal{E}}_i$ correspond une transition visible de $\widehat{\mathcal{E}'}_i$

$$\forall t \in T_v(\widehat{\mathcal{E}}_i)\ \ \exists t' \in T_v(\widehat{\mathcal{E}}_i')\ /\ \ h_i(t) = t'$$

e) Chaque propriété statique visible valide dans $\hat{\mathcal{E}}_i$ est une propriété statique (au renommage prés via h_i) visible et valide dans $\widehat{\mathcal{E}'}_i$:

$$\forall \alpha, \beta \in T_{\Sigma_v(\hat{\mathcal{E}}_i)}(V)$$

$$\alpha =_{E(\hat{\mathcal{E}}_i)} \beta \ \ \Rightarrow \begin{bmatrix} \exists \alpha', \beta' \in T_{\Sigma_v(\hat{\mathcal{E}}_i')}(V)\ / \\ \left(h_i(\alpha) = \alpha'\right) \wedge\ \left(h_i(\beta) = \beta'\right) \wedge\ \left(\alpha' =_{E(\hat{\mathcal{E}}_i')} \beta'\ \right) \end{bmatrix}$$

f) Chaque propriété dynamique visible dans $\hat{\mathcal{E}}_i$ est une propriété dynamique (au renommage prés via h_i) visible valide dans $\widehat{\mathcal{E}'}_i$:

f-1) Chaque place p visible dans $\hat{\mathcal{E}}_i$ a le même marquage initial (au renommage près via h_i) que la place $h_i(p)$: $\forall p \in P_v(\widehat{\mathcal{E}}_i)\ \ h_i(M_0(p)) = M_0'(h_i(p))$

où M_0 et M_0' sont respectivement les marquages initiaux de $\hat{\mathcal{E}}_i$ et $\widehat{\mathcal{E}'}_i$.

f-2) Pour chaque couple de marquage M_1 et M_2 de $\hat{\mathcal{E}}_i$, si M_2 est accessible à partir de M_1 par une séquence de transitions $\lambda_1.t.\lambda_2$ (où t est visible et λ_1 et λ_2 sont des séquences de transitions non visibles) alors il existe un couple de marquages M_1' et M_2' de $\widehat{\mathcal{E}'}_i$ (vérifiant les conditions $M_1'_{|h_i(P_v(\hat{\mathcal{E}}_i))} = h_i(M_{1|P_v(\hat{\mathcal{E}}_i)})$ *et* $M_2'_{|h_i(P_v(\hat{\mathcal{E}}_i))} = h_i(M_{2|P_v(\hat{\mathcal{E}}_i)})$) tels que M_2' est accessible à partir de M_1' par une séquence de transitions $\lambda_1'.h_i(t).\lambda_2'$ (où λ_1' et λ_2' sont des séquences de transitions non visibles ou visibles non image par h_i d'une transition visible de $\hat{\mathcal{E}}_i$).

$$\forall M_1\ M_2\ \forall t \in T_V(\widehat{\mathcal{E}}_i)\ \ \forall \lambda_1\ \lambda_2 \in \left(T(\widehat{\mathcal{E}}_i) - T_v(\widehat{\mathcal{E}}_i)\right)^*$$

$$M_1[\lambda_1.t.\lambda_2 > M_2 \quad \Rightarrow \begin{cases} \exists M_1'\ / M_1'_{|h_i(P_v(\widehat{\mathcal{E}}_i))} = h_i(M_{1|P_v(\widehat{\mathcal{E}}_i)}) \\ \exists M_2'\ / M_2'_{|h_i(P_v(\widehat{\mathcal{E}}_i))} = h_i(M_{2|P_v(\widehat{\mathcal{E}}_i)}) \\ \exists t' \in T_V(\widehat{\mathcal{E}}_i')/t' = h_i(t) \\ \exists \lambda_1'\ \lambda_2' \left(\left(T(\widehat{\mathcal{E}}_i') - T_v(\widehat{\mathcal{E}}_i')\right) \cup \left(T_v(\widehat{\mathcal{E}}_i') - h_i\left(T_v(\widehat{\mathcal{E}}_i)\right)\right)\right)^* \\ \quad M_1'[\lambda_1'.t'.\lambda_2' > M_2' \end{cases}$$

f-3) Pour chaque couple de marquage M_1 et M_2 de $\widehat{\mathcal{E}}_i$, si M_2 est accessible à partir de M_1 par une séquence de transitions λ non visible alors il existe un couple de marquages de M_1' et M_2' de $\widehat{\mathcal{E}'}_i$ (où $M_1'_{|h_i(P_v(\widehat{\mathcal{E}}_i))} = h_i(M_{1|P_v(\widehat{\mathcal{E}}_i)})$ *et* $M_2'_{|h_i(P_v(\widehat{\mathcal{E}}_i))} = h_i(M_{2|P_v(\widehat{\mathcal{E}}_i)})$) tels que M_2' est accessible à partir de M_1' par une séquence de transitions λ' (où λ' est une séquence de transitions non visibles, ou visibles mais non images par h_i de transitions visibles de $\widehat{\mathcal{E}}_i$).

$$\forall M_1\ M_2 \quad \forall \lambda \in \left(T(\widehat{\mathcal{E}}_i) - T_v(\widehat{\mathcal{E}}_i)\right)^*$$

$$M_1[\lambda > M_2 \quad \Rightarrow \begin{cases} \exists M_1'\ / M_1'_{|h_i(P_v(\widehat{\mathcal{E}}_i))} = h_i(M_{1|P_v(\widehat{\mathcal{E}}_i)}) \\ \exists M_2'\ / M_2'_{|h_i(P_v(\widehat{\mathcal{E}}_i))} = h_i(M_{2|P_v(\widehat{\mathcal{E}}_i)}) \\ \exists \lambda' \in \left(\left(T(\widehat{\mathcal{E}}_i') - T_v(\widehat{\mathcal{E}}_i')\right) \cup \left(T_v(\widehat{\mathcal{E}}_i') - h_i\left(T_v(\widehat{\mathcal{E}}_i)\right)\right)\right)^* \\ \quad M_1'[\lambda' > M_2' \end{cases}$$

f-4)Pour chaque séquence de transitions $\lambda_1.t_1.\lambda_2.t_2.....\lambda_n.t_n.\lambda_{n+1}$ franchissable de $\widehat{\mathcal{E}}_i$ (où $t_1, t_2, t_3..., t_n$ sont des transitions visibles et $\lambda_1, \lambda_2, ..., \lambda_n, \lambda_{n+1}$ sont des séquences de transitions non visibles), il existe une séquence de transitions $\lambda_1'.h_i(t_1).\lambda_2'.h_i(t_2).....\lambda_n'.h_i(t_n).\lambda_{n+1}'$ franchissable dans $\widehat{\mathcal{E}'}_i$ où $\lambda_1', \lambda_2', ..., \lambda_n', \lambda_{n+1}'$ sont des séquences de transitions non visibles ou visibles mais non images par h_i de transitions visibles de $\widehat{\mathcal{E}}_i$

$$\forall \{t_1, t_2, ..., t_n\} \subseteq T_v(\widehat{\mathcal{E}}_i)\ \forall j = 1..n+1\ \ \lambda_j \in \left(T(\widehat{E}_i) - T_v(\widehat{E}_i)\right)^*$$

$$\lambda_1.t_1.\lambda_2.t_2.....\lambda_n.t_n.\lambda_{n+1} \in SeqF(\widehat{\mathcal{E}}_i) \Rightarrow \begin{cases} \exists \lambda_1', \lambda_2', ..., \lambda_{n+1}' \in \left(\left(T(\widehat{\mathcal{E}}_i') - T_v(\widehat{\mathcal{E}}_i')\right) \cup \left(T_v(\widehat{\mathcal{E}}_i') - h_i\left(T_v(\widehat{\mathcal{E}}_i)\right)\right)\right)^* / \\ \lambda_1'.h_i(t_1).\lambda_2'.h_i(t_2).....\lambda_n'.h_i(t_n).\lambda_{n+1}' \in SeqF(\widehat{\mathcal{E}}_i') \end{cases}$$

Notation :

$SeqF(\widehat{\mathcal{E}})$: est l'ensemble des séquences de transitions franchissables dans $\widehat{\mathcal{E}}$.

Exemple: Le module MACHINE2 est un paramètre effectif valide pour MACHINE1 via le morphisme *morph1* précisé par la Figure 6.7.

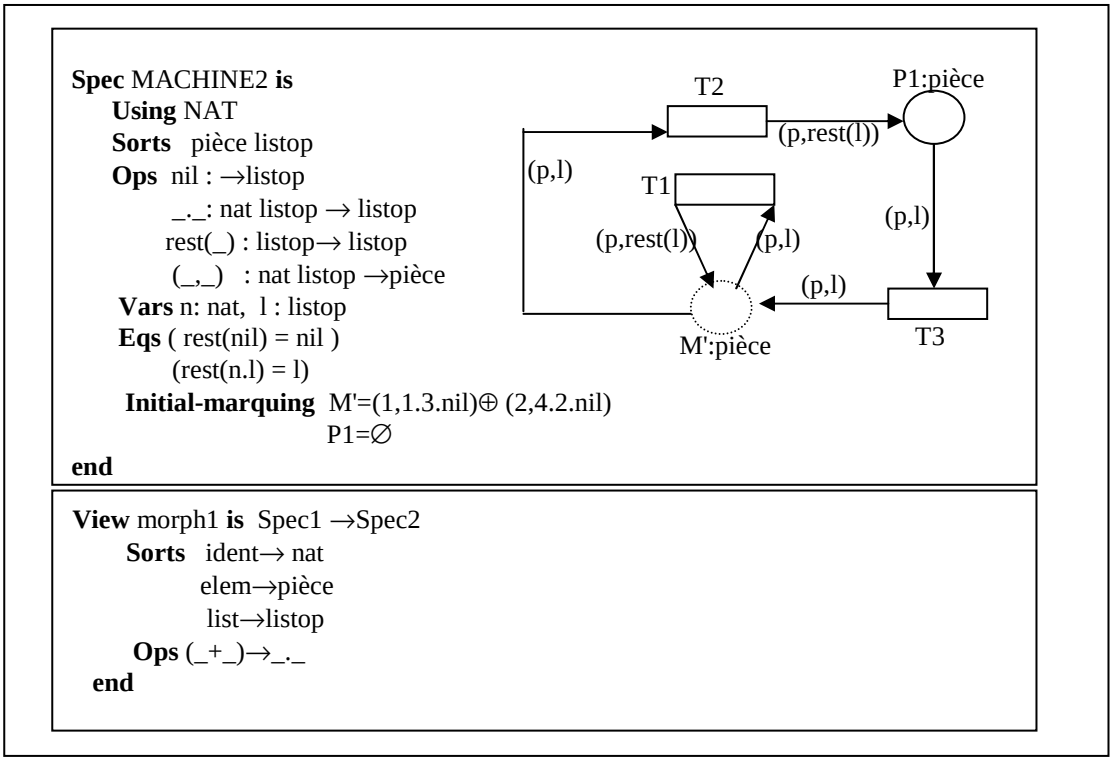


Figure 6.7 : Exemple de paramètre effectif valide dans une paramétrisation dynamique.

Nous pouvons déduire à partir des deux approches de paramétrisation des ECATNets une approche mixte où un module peut être paramétré par les données et le comportement. Ces deux types de paramétrisation peuvent concerner le même paramètre formel ou des paramètres formels différents d'un même module paramétré.

VI- Syntaxe

La différence entre les types de paramétrisation d'un module n'est pas reflétée au niveau syntaxe du module paramétré. En fait c'est plutôt la définition du module paramètre formel qui détermine le type de paramétrisation considéré. Dans le cas où le module paramètre formel est un module de spécification algébrique alors la paramétrisation par ce module est statique. Si le module paramètre formel est un module ayant un réseau non vide, la paramétrisation est alors dynamique.

1- Syntaxe des modules paramétrés

1-1 Syntaxe abstraite des modules paramétrés

Un module paramétré est caractérisé par:

- L'identificateur du module paramétré
- La liste *non vide* des modules paramètres formels
- Le corps du module

La syntaxe abstraite d'un module paramétré est définie par la règle:

< module générique>::=(<nom du module>,[<noms des paramètres formels>]$^{+}$,<corps >)

1-2 Syntaxe concrète des modules paramétrés

Nous associons à la syntaxe abstraite précédente la syntaxe concrète suivante:

<module générique>::= **Gen** <identificateur de module générique>
[<déclaration de paramètres formels>**]** **is**
<corps de module> **end**
<déclaration de paramètres formels> : := <déclaration d'un paramètre>
⟦ **,**<déclaration d'un paramètre> ⟧ *
<déclaration d'un paramètre>: := <identificateur module paramètre formel>

<corps de module > : := ⟦ <importation de modules> ⟧*
⟦ <déclaration de sortes> ⟧*
⟦ <déclaration des sous-sortes> ⟧*
⟦ <déclaration des opérations > ⟧*
⟦ <déclaration des variables> ⟧*
⟦ <déclaration des équations > ⟧*
⟦ <déclarations de réseau> ⟧ *

2- Syntaxe de l'instanciation

2-1 Syntaxe abstraite de l'instanciation

La génération d'un module par instanciation d'un module générique est définie par:

- L'identificateur du module résultat de l'instanciation (ou généré): "mod-id"
- L'identificateur du module paramétré à instancier: "nom du module paramétré"
- La liste non vide des paramètres effectifs et les morphismes correspondants: "instance"

Les règles de la syntaxe abstraite sont:

<instanciation>::= (mod-id, <nom du module paramétré> , [< instance>]$^{+}$)
<instance>::= (<nom du paramètre formel>,<nom de paramètre effectif>,
⟦ <morphisme d'adéquation> ⟧)
<morphisme-d'adéquation> :: =⟦ couples-de-sortes | couples-de-opérations |
couples-de-places | couples-de-transitions-contexte ⟧$^{+}$

2-2 Syntaxe concrète de l'instanciation

Les règles de la syntaxe concrète de l'instanciation sont:

<instanciation> : := <identificateur de module paramétré> **[**<liste des paramètres effectifs>**]**
<liste de paramètres effectifs> : := <paramètre effectif> ⟦ , <paramètre effectif> ⟧*
<paramètre effectif> : := <identificateur de view> | <identificateur de module non paramétré>

VII- Conclusion

Nous avons présenté le mécanisme de paramétrisation des ECATNets qui permet de spécifier des systèmes génériques. Deux approches de paramétrisation des ECATNets ont été définies:

- La première dite *paramétrisation statique* est inspirée principalement des travaux sur la paramétrisation des spécifications algébriques permet de spécifier des systèmes ayant la même structure de contrôle mais qui manipulent des données "différentes".

- La seconde est la *paramétrisation dynamique* qui permet de décrire des systèmes ayant des comportements visibles comparables. Cette approche est une généralisation de la méthode de paramétrisation d'un réseau de Petri par un autre réseau que nous avons présenté dans [ZEG-96][ZM-97].

Nous avons également défini le principe d'instanciation pour les modules paramétrés et nous avons montré que cette instanciation peut être totale ou partielle. Elle peut être également formelle ou effective.

Dans le chapitre 9 nous donnerons la sémantique formelle des modules paramétrés (génériques) et la sémantique du principe d'instanciation.

Dans le futur il serait intéressant d'élaborer des outils d'analyse des modules paramétrés permettant de vérifier qu'un module est un paramètre effectif valide (vérifie les contraintes statiques et dynamiques décrite par un module paramètre formel). Des travaux sur la réduction des réseaux de Petri et leur abstraction [BER-83][DES-91] peuvent faciliter de telles vérifications. Une autre perspective de cette approche de paramétrisation est sa généralisation à d'autres types de réseaux de Petri.

Chapitre 7

Syntaxe du langage CIRTA

Sommaire

I- Introduction

Dans ce chapitre, nous allons décrire les aspects syntaxiques du langage de spécification CIRTA. Nous précisons en premier lieu, les aspects lexicographiques du langage, ainsi que les notations utilisées pour définir les règles syntaxiques du langage CIRTA.

II- Aspects lexicographiques du langage CIRTA

1. Notations utilisées

<A> : symbole non terminal A.
A : symbole terminal (en caractères gras).
| : opérateur indiquant une alternative.
{A1,A2,...,An} : sélection de A1,A2,..ou An
⟦ A ⟧ : zéro ou une occurrence de A.
⟦ A ⟧* : un nombre quelconque (nul ou non) d'occurrences de A.
⟦ A ⟧$^{+}$: au moins une occurrence de A.

2. Jeu de caractères

<caractère> : := <lettre>| <chiffre> | <autres caractères>
<lettres> : := <lettre majuscule> | <lettre minuscule>
<lettre majuscule> : := **A**|**B**|**C**|...|**Z**
<lettre minuscule> : := **a**|**b**|**c**|...|**z**
<chiffre> : := **0**|**1**|**2**|...|**9**
<autres caractères> : := **+**|**-**|*****|**/**|**_**|**'**| **»**| **«** | **#**|**&**|**(**|**)**|**<**|**>**|**.**|**,**| **;**| **?**| **:**|**=>**|**<=**| ....

3. Identificateurs

Les identificateurs sont constitués d'une séquence de caractères contigus sans séparateurs et sans délimiteurs. Ces caractères sont soient des lettres, des chiffres ou des tirets. Un identificateur commence obligatoirement par une lettre. A la différence des mots réservés, les majuscules et les minuscules sont distinguées dans les identificateurs. Par exemple, toto et ToTo sont deux identificateurs différents.

<identificateur> : := <lettre > ⟦ <lettre>|<chiffre>|**_** ⟧ *

4. Commentaires

Un commentaire est une séquence éventuellement vide de caractères. Cette séquence est mise entre deux guillemets. Les commentaires peuvent contenir n'importe quel caractère. Aucun contrôle n'est effectué jusqu'au caractère fermant le commentaire.

<commentaire> : := **«**⟦ <caractère> ⟧* **»**

5. Mots réservés

Les mots réservés du langage CIRTA sont :

combine, else, end, eqns, eqs, export, formal, gen, hide, if, initial-marking, interface, interfaces, is, let, link, marking, opns, ops, pl, place, places, rename, sort, sorts, spec, subnode, subsort, subsorts, then, to, tr, transition, transitions, use, using, var, vars, variable, variables, view.

Les majuscules et les minuscules ne sont pas distinguées dans les mots réservés. Par exemple les mots ComBine, comBINE et COMBINE désignent le même mot réservé.

III- Syntaxe concrète du langage CIRTA

Dans ce paragraphe nous détaillons les règles de la syntaxe concrète du langage CIRTA.

Spécification

<spécification> ::= <module> | <morphisme-view>
<module> : := <module ordinaire> | <module formel> | <module générique>

Module ordinaire

<module ordinaire>: := **Spec** <identificateur de module> **is** <corps de module> **end** |
Spec <identificateur de module> **is** <expression de module > **end**
<identificateur de module> : := <identificateur>

Module formel

<module formel> : := **Formal** <identificateur de module> **is** <corps de module> **end** |
Formal <identificateur de module> **is** <expression de module> **end**

Module générique

<module générique >: := **Gen** <identificateur de module> **[**<déclaration de paramètres >**] is**
<corps de module> **end** |
Gen <identificateur de module> **[**<déclaration de paramètres>**] is**
<expression de module > **end**

<déclaration de paramètres> : := <identificateur de module formel>
⟦ **,**<identificateur de module formel> ⟧*

<identificateur de module formel> : := <identificateur>

Corps d'un module

<corps de module > : := ⟦ <importation de modules> ⟧ *
⟦ <déclaration de sortes> ⟧*
⟦ <déclaration des sous-sortes> ⟧*
⟦ <déclaration des opérations > ⟧*
⟦ <déclaration des variables> ⟧*
⟦ <déclaration des équations > ⟧*
⟦ <déclarations de réseau> ⟧*

Déclaration de sortes

<déclaration des sortes> : := { **sort, sorts**} ⟦ <identificateur d'une sorte> ⟧$^{+}$
<identificateur d'une sorte> : := <identificateur>

Déclaration des sous-sortes

<déclaration des sous-sortes> : := {**subsort, subsorts**}
⟦ <identificateur d'une sorte> ⟦ < <identificateur d'une sorte> ⟧ $^{+}$ ⟧$^{+}$

Déclaration des opérations

<déclaration des opérations > : := { **ops, opns**} ⟦ <déclaration d'une opération> ⟧ $^{+}$

<déclaration d'une opération> : := <forme d'opération> **:** <domaine> → <codomaine>
<domaine> : := ⟦ <identificateur de sorte> ⟧ *
<codomaine> : := <identificateur de sorte>

Déclaration des variables

<déclaration des variables> : := {**var, vars, variable, variables** }
⟦ < variable> ⟦ **,**< variable > ⟧ * **:** <identificateur de sorte> ⟧$^{+}$
< variable> : :=<identificateur>

Déclaration des équations

<déclaration des équations > : := { **eqs, eqns**} ⟦ <déclaration d'une équation> ⟧ $^{+}$
<déclaration d'une équation> : := **(**<terme >**=** <terme> **)** |
(<terme>**=**<terme> **if** <terme>**)** |
(<terme>**= if** <terme> **then** <terme> **else** <terme>**)**

Les non terminaux <terme> et <forme d'opération> ne sont pas définis car ils dépendent étroitement de la syntaxe introduite par l'utilisateur.

Importation de module

<importation de modules> : := <primitive use> |
<primitive combine> |
<primitive rename> |
<primitive visibilité> |
<instanciation>

<primitive use> ::= {**use, using**} <expression de module> |
<primitive combine> ::= **combine** <expression de module> ⟦ <expression de module> ⟧$^{+}$
<primitive rename> ::= **rename** < expression de module> <morphisme de renommage>
<primitive visibilité> ::= {**hide, export**} <expression de module> ⟦ <visibilité> ⟧$^{+}$
<instanciation> ::= <identificateur de module> **[**<liste des paramètres effectifs>**]**

<expression de module> : := <expression élémentaire> ⟦ **+** <expression élémentaire> ⟧*
<expression élémentaire> : := <module simple> ⟦ ***(**<morphisme de renommage>**)** ⟧
<module simple> : := <identificateur de module> ⟦ **[**<liste des paramètres effectifs>**]** ⟧

<liste de paramètres effectifs> : := <paramère effectif> ⟦ , <paramètre effectif> ⟧ *
<morphisme de renommage> : := ⟦ <morphisme de renommage des sortes> ⟧ *
⟦ <morphisme de renommage des opérations> ⟧ *
⟦ <morphisme de renommage des places> ⟧ *
⟦ <morphisme de renommage des transitions> ⟧*

<morphisme de renommage des sortes> : := {**so, sort, sorts**}
⟦ **(**<identificateur de sorte> → <identificateur de sorte>**)** ⟧ $^{+}$
<morphisme de renommage des opérations> : :={**ops, opns**}
⟦ **(** <opérateur>→ <forme d'opération>**)** ⟧ $^{+}$
<morphisme de renommage des places> : := {**place, places**}
⟦ **(**<identificateur de place> → <identificateur de place>**)** ⟧ $^{+}$
<morphisme de renommage des transitions> : := {**transition, transitions**}
⟦ **(**<identificateur de transition et contexte>→
<identificateur de transition et contexte>**)** ⟧ $^{+}$

<identificateur de transition et contexte> : := <identificateur de transition> ⟦ **(**<contexte>**)** ⟧
<contexte> : := <identificateur de variable>**:** <identificateur de sorte>
⟦ **,** <identificateur de variable>**:** <identificateur de sorte> ⟧$^{+}$
<opérateur> : := <forme d'opération> ⟦ **:** <domaine> → <codomaine> ⟧
<forme d'opération> ::= ⟦ <caractère> ⟧$^{+}$

<paramètre effectif> : := <identificateur de view> | <identificateur de module>

<visibilité> **::=** **{Sort, sorts}** ⟦ <identificateur de sorte> ⟧$^{+}$ |
{ops, opns} ⟦ <identificateur d'opération> ⟧$^{+}$ |
{place, places} ⟦ <identificateur de place> ⟧$^{+}$ |
{transition, transitions} ⟦ <identificateur de transition> ⟧$^{+}$

Morphisme d'adéquation

<morphisme-view> : := {**view, views**} <identificateur de view> **is**
<identificateur de module> → <identificateur de module>
< morphisme d'adéquation>
end
<identificateur de view>::= <identificateur>
< morphisme d'adéquation> : := ⟦ <morphisme adéquation de sortes> ⟧*
⟦ <morphisme adéquation d'opérations> ⟧*
⟦ <morphisme adéquation de places> ⟧*
⟦ <morphisme adéquation de transitions> ⟧*

<morphisme adéquation de sortes> : := {**sort, sorts**}
⟦ **(**<identificateur de sorte> → <identificateur de sorte>**)** ⟧$^{+}$
<morphisme adéquation d'opérations> : :={**ops, opns**}
⟦ **(** <opérateur>→<forme d'opération>**)** ⟧$^{+}$
<morphisme adéquation de places> : := {**place, places**}
⟦ **(**<identificateur de place> → <identificateur de place>**)** ⟧$^{+}$
<morphisme adéquation de transitions> : := {**transition, transitions**}
⟦ **(**<identificateur de transition et contexte>→
<identificateur de transition et contexte>**)** ⟧$^{+}$

Déclaration de réseau
<déclaration de réseau > : := <déclaration textuelle de réseau> |
<déclaration graphique de réseau>

Déclaration textuelle de réseau

<déclaration textuelle de réseau> : := ⟦ <déclaration de places> ⟧
⟦ <déclaration de transitions > ⟧
⟦ <déclaration des variables> ⟧
⟦ <déclaration d'abstractions syntaxiques> ⟧
⟦ <déclaration d'interfaces> ⟧
⟦ <déclaration de nœuds de substitution> ⟧
⟦ <marquage initial> ⟧

Déclaration de places

<déclaration de places> : := {**place, places**}⟦ <déclaration d'une liste de places> ⟧$^{+}$

<déclaration d'une liste de places> : := <identificateur de place>⟦ **,**<identificateur de place> ⟧*****
: <identificateur de sorte> ⟦ **(** <capacité>**)** ⟧
<identificateur de place> : := <identificateur>
<capacité> : := <entier naturel positif >

Déclaration de transitions

<déclaration de transitions> : := {**transition, transitions**}⟦ <déclaration d'une transition> ⟧$^{+}$

<déclaration d'une transition> : :=
<identificateur de transition>**:**<précondition> → <postcondition>
⟦ **if** <condition de transition> ⟧⟦ **[**<temps>**]** ⟧

<identificateur de transition> : := <identificateur>
<précondition> : := ⟦ **(**<identificateur de place>**,** <condition d'entrée>**,** <jetons détruits>**)** ⟧ *****
<postcondition> : := ⟦ **(**<identificateur de place>**,** <jetons créés>**)** ⟧*****
<condition de transition> : := <terme de sorte bool>
<condition d'entrée> : := **empty** | <multi-ensemble> | **~**<multi-ensemble>
<jetons détruits> : := ∀ | <multi-ensemble>
<jetons créés> : := <multi-ensemble>
<multi-ensemble> : := <terme> | < multi-ensemble> ⊕ <multi-ensemble> |
<entier naturel positif> ⊗ <multi-ensemble>
<temps> ::= <entier naturel>**,** <entier naturel>

Déclaration d'interface

<déclaration d'interfaces> : := {**interface, interfaces**}⟦ <place ou transition interface> ⟧$^{+}$

<place ou transition interface> : := <place interface> | <transition interface>
<place interface> : := {**place, places**} ⟦ <identificateur de place> ⟧ $^{+}$
<transition interface> : := {**transition, transitions**}
⟦ < identificateur de transition et contexte> ⟧ $^{+}$

Déclaration d'abstractions syntaxiques :

<déclaration d'abstractions syntaxiques >: := **let** ⟦ < affectation> ⟧ $^{+}$
<affectation> : := **(**<identificateur de variable> **=** <multi-ensemble>**)**

Déclaration de nœuds de substitution:

<déclaration de nœuds de substitution> ::= **subnode** ⟦ <un nœud de substitution> ⟧$^{+}$

<un nœud de substitution ::= <identificateur de noeud> **:**<identificateur de module>
⟦ **link** ⟦ **(**<morphisme de nœuds interfaces>**)** ⟧$^{+}$ ⟧

<morphisme de nœuds interfaces> ::=
pl <identificateur de place>→<identificateur de place> |
tr <identificateur de transition>→<identificateur de transition>

<identificateur de noeud> ::=<identificateur>

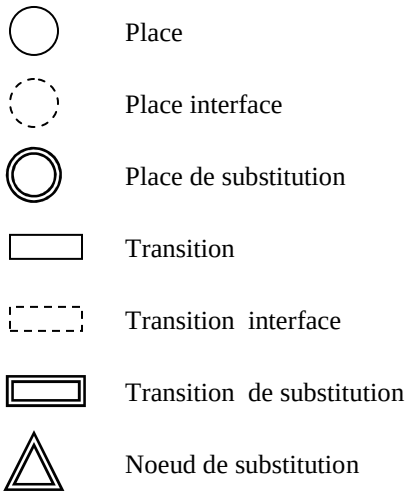
Marquage initial

<marquage initial> : := {**marking, initial-marking**}
⟦ <marquage d'une place> ⟧$^{+}$

<marquage d'une place> : := <identificateur de place> **=** <multi-ensemble>

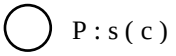
Déclaration graphique du réseau

Le réseau est représenté par un graphe ayant différents types de noeuds. La forme des nœud (cercle, rectangle, triangle) et le type de trait (continu, discontinu, double) réfère à ces nœuds différentes signification qui se résument comme suit:

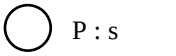


Inscription des places

L'identificateur de la place P, sa sorte s et sa capacité c (si elle est définie) sont écrits à côté du cercle correspondant, sous la forme :



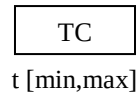
Place P de sorte s et de capacité c



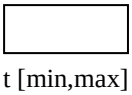
Place P de sorte s et de capacité infinie

Inscription des transitions

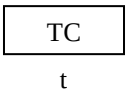
Une transition t ayant éventuellement la condition de transition *TC* (« Transition Condition ») et l'intervalle temporel statique [min, max] sera représentée selon la figure suivante :



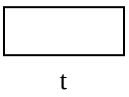
Transition t ayant une condition de transition TC et un intervalle temporel statique [min,max]



Transition t sans condition de transition et ayant un intervalle temporel statique [min,max]



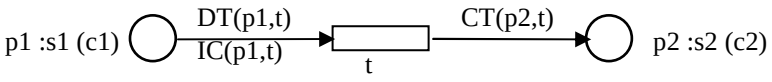
Transition t sans contraintes temporelles et ayant une condition de transition TC



Transition t sans contraintes temporelles et sans condition de transition

Inscription des arcs

La condition d'entrée IC («Input Condition »), les jetons détruits DT(« Destroyed Tokens ») et les jetons créés CT (« Created Tokens ») sont écrits sur les arcs sous la forme :



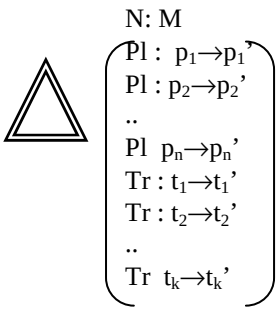
IC : à gauche de l'arc d'entrée à la transition pour un observateur situé au niveau de celle-ci.

DT : à droite de l'arc d'entrée à la transition pour un observateur situé au niveau de cette transition.

CT : à gauche ou à droite de l'arc de sortie d'une transition.

Inscription des nœuds de substitution

Un nœud de substitution N représentant un module M tel que les places p_1, p_2…p_n du réseau courant correspondent respectivement dans M à p_1' p_2' …p_n' ; et les transitions t_1,t_2,..t_k correspondent à t_1',t_2'…t_k'



Les arcs adjacents à un nœud de substitution ne sont pas orientés et n'ont pas d'inscriptions.

IV- Conclusion

La syntaxe concrète du langage présente une souplesse d'utilisation caractérisée par:

- Une combinaison de la représentation textuelle et une représentation graphique (ou le graphe peut être décrit sous forme de texte)
- Une liberté totale dans le choix des formes des opérations définies par l'utilisateur: chacune de ces opérations peut être préfixée, postfixée, mixfixée, distribuée, anonyme, surchargée,..etc.
- La possibilité de la représentation multiple d'une même place ou une même transition dans un même module. Ceci permet de ne pas avoir des arcs qui vont "d'un bout du réseau à l'autre".
- La possibilité de définition d'abstractions syntaxiques qui permettent d'éviter la manipulation de réseaux ayant des inscriptions (*IC,DT,CT, marquage*) complexes. A chaque inscription est alors associé un nom (l'abstraction syntaxique de cette inscription) qui rappelle la déclaration de constantes dans les langages de programmation.

Chapitre 8

Sémantique formelle des ECATNets

Sommaire

I- Introduction

Dans le *chapitre 1* le comportement d'un ECATNet a été décrit de façon informelle en précisant les conditions et le type de franchissement des transitions et l'effet de ce franchissement sur l'état (ou le marquage) de l'ECATNet. La définition formelle de la sémantique du comportement dynamique des premières versions des ECATNet a été donnée dans [BM-91][BMSB-93] [BM-95] en utilisant la logique de réécriture. Cependant ces dernières années, les ECATNets ont évolué et de nouveaux concepts y ont été introduits tels que : notion de *capacité, contraintes contextuelles* et *contraintes temporelles.*

L'objectif de ce chapitre est de montrer comment on peut associer à un ECATNet enrichi avec ces nouveaux concepts, une théorie de réécriture pour décrire sa sémantique dénotationnelle et déduire ensuite sa classe de modèles. Cette sémantique sera présentée progressivement en introduisant pas à pas chacun des concepts tels que la capacité, contraintes contextuelles et contraintes temporelles.

Avant d'entamer l'objectif principal de ce chapitre, il est nécessaire de rappeler la définition formelle de quelques notions fondamentales de la logique de réécriture.

II- Théorie de réécriture

La logique de réécriture a été évoquée par J. Meseguer [Mes-90][Mes-91] comme un modèle unificateur de la concurrence qui permet de raisonner correctement sur les systèmes à états évoluant au moyen de transitions. La logique de réécriture constitue un modèle générique de la concurrence à partir duquel plusieurs autres modèles peuvent être obtenus par spécialisations.

Ainsi tout système dynamique défini par un ensemble d'états et un ensemble de transitions élémentaires entre eux, peut être spécifié par une théorie de réécriture.

- La structure particulière de l'état du système est définie par une signature ($Sig, \mathcal{E}$) telle que :
  Sig: est l'alphabet de symboles de fonctions qui décrivent l'état du système,
  $\mathcal{E}$: sont les axiomes structurels.
- Une transition spécifiant le passage d'un état élémentaire vers un nouvel état est décrite par une règle de réécriture de la forme $r:[t]\rightarrow[t']\ if\ C$ qui exprime le fait que le terme $[t]$ se réécrit (se transforme) en $[t']$ si la condition *C* est vérifiée. L'ensemble de ces règles est $\mathcal{R}$.

1- Définition [Mes-90]

Une théorie de réécriture étiquettée $\mathcal{T}$ est un 4-tuple $\mathcal{T} = (Sig, \mathcal{E}, \mathcal{L}, \mathcal{R})$ où

Sig : est un alphabet ordonné de symboles de fonctions.

$\mathcal{E}$: est un ensemble de *Sig*-équations.

$\mathcal{L}$: un ensemble d'étiquettes.

$\mathcal{R}$: est un ensemble de règles de réécriture. Il est défini par un ensemble de couples $\mathcal{R} \subseteq \mathcal{L}\times T_{\mathrm{Sig},\mathcal{E}}(X)^2)^+$ tel que:

- Le premier composant est une étiquette,
- le second est formé des paires de classes d'équivalence de termes modulo $\mathcal{E}$ (où X est un ensemble fini et dénombrable de variables).

Notation:

- Une règle: $(r,\ ([t],[t']),([u_1],[v_1]),...,([u_k],[v_k]))$
  sera notée $r:\ [t]\to[t'])$ if $[u_1]\to[v_1]\wedge...\wedge[u_k]\to[v_k]$
  ou bien $r:[t]\to[t']\ if\ \ C$ où $C=[u_1]\to[v_1]\wedge...\wedge[u_k]\to[v_k]$ est la condition de le règle

- Les règles de la forme $(r,\ ([t],[t']))$ sont dites inconditionnelles (la condition *C* est vide) et sont notées par $r:\ [t]\to[t']$.

- Une règle de réécriture utilisant les variables $x_1,x_2,...,x_n$ sera notée:

$$r:\ [t(x_1,x_2,...,x_n)]\to[t'(x_1,x_2,...,x_n)]\ \ if\ \ C(x_1,x_2,...,x_n)$$

ou bien $r:\left[t(\bar{x}^n)\right]\to\left[t'(\bar{x}^n)\right]$ if $C(\bar{x}^n)$

2- Déduction dans la logique de réécriture

Le calcul dans un système dynamique (concurrent ou non) est une séquence de transitions exécutées à partir d'un état initial donné. Ce calcul correspond à une preuve ou une déduction dans la logique de réécriture. Cette déduction s'effectue grâce à une réécriture (concurrente ou non) des classes d'équivalence de termes modulo les axiomes structuraux d'associativité, de commutativité et d'identité.

Etant donnée une théorie de réécriture $\mathcal{T}$, on dit qu'un séquent $[t]\to[t']$ est prouvable dans $\mathcal{T}$ et on note $\mathcal{T}\vdash [t]\to[t']$ si et seulement si $[t]\to[t']$ peut être obtenue par une application finie les règles de déduction suivantes :

(i) Réflexivité : $\forall[t]\in \mathrm{T}_{Sig,\varepsilon}(X)\ \ on\ \ a\ \ \overline{t:\ [t]\to[t]}$

(ii) Congruence :

$$\forall f\in\ Sig_n\ \ \frac{\pi_1:\left[t_1\right]\to\left[t_1'\right],...,\pi_n:\left[t_n\right]\to\left[t_n'\right]}{f(\pi_1,...\pi_n):\ \ \left[f(t_1,...,t_n)\right]\to\left[f(t_1',...,t_n')\right]}$$

où Sig_n est l'ensemble des fonctions de *Sig* d'arité *n.*

(iii) Substitution :

$$\forall\ (\ \pi:[t(x_1,...,x_n)]\to[t'(x_1,...,x_n)]\ \text{if}\ \wedge_{i=1..k}\left[u_i(x_1,...,x_n)\right]\to\left[v_i(x_1,...,x_n)\right]\ )\in R$$

$$\frac{\pi_1:\ \left[w_1\right]\to\left[w_1'\right],...,\ \pi_n:\left[w_n\right]\to\left[w_n'\right]\quad \left[u_1(\bar{w}/\bar{x})\right]\to\left[v_1(\bar{w}/\bar{x})\right],...,\left[u_k(\bar{w}/\bar{x})\right]\to\left[v_k(\bar{w}/\bar{x})\right]}{\pi(\pi_1,..,\pi_n):\left[t(\bar{w}/\bar{x}\right]\to\left[t'(\bar{w'}/\bar{x})\right]}$$

où $\bar{w}/\bar{x}$ est une substitution des x_i par les $w_i\quad 1\le i\le n$

(iv) Transitivité :

$$\forall\left[t_1\right]\left[t_2\right]\left[t_3\right]\in \mathrm{T}_{Sig,\varepsilon}(X)\quad \frac{\pi_1:\left[t_1\right]\to\left[t_2\right]\ \ \pi_2:\left[t_2\right]\to\left[t_3\right]}{\pi_1;\pi_2:\ \left[t_1\right]\to\left[t_2\right]..}$$

De manière générale, la déduction dans la logique de réécriture est une itération des étapes suivantes:

a) La règle de *substitution* identifie toutes les règles de réécriture applicables à l'état global courant (ce sont les règles dont le membre gauche correspond à un sous-terme de l'état global courant). Comme la logique de réécriture est une logique de changement, la règle de réflexivité, appliquée aux sous-termes non concernés par les règles de réécritures identifiées, permet de les transformer en eux-mêmes.

b) Les règles de réécriture, identifiées par la règle de *substitution* dans a) ainsi que la règle de *réflexivité*, sont exécutées en concurrence et indépendamment les unes des autres. La règle de *congruence* compose les résultats de ces exécutions (membres droits, de ces règles) pour construire un nouvel état (terme) global.

c) Les étapes a) et b) sont réitérés jusqu'à ce qu'il n'y ait plus de règles applicables.

d) Enfin, la règle de *transitivité* construit la séquence de réécritures exécutée de l'état initial à l'état final. Cette séquence correspond à un calcul possible dans le système concurrent.

Les concepts fondamentaux de la logique de réécriture présentés ci-dessus nous permettent de présenter la théorie de réécriture des ECATNets. Pour plus de détails concernant cette logique, le lecteur peut consulter les travaux de [Mes-90] [Mes-91].

III- Sémantique des ECATNets simples

Un ECATNet simple est un ECATNet où toutes les places ont des capacités infinies, aucune transition n'exige pour son franchissement des contraintes contextuelles (positives ou négatives) ni détruit lors de son franchissement tout le contenu d'une place.

1- Définition

$\mathcal{E}$ est un ECATNet *simple* si et seulement si les conditions suivantes sont vérifiées simultanément :

$$\begin{cases} - \ dom(Cap) = \varnothing \\ - \ dom(IC) = \varnothing \\ - \ \forall (p,t) \ \ \big((p,t) \in dom(DT) \ \rightarrow \ DT(p,t) \neq \forall\big) \end{cases}$$

Exemple :

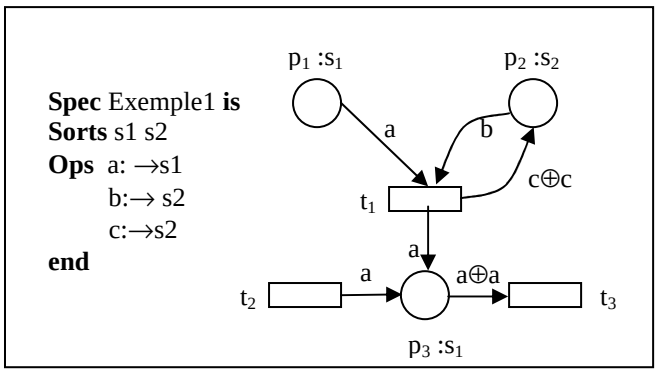


Figure 8.1 : Exemple d'un ECATNet simple.

2- Etat d'un ECATNet simple

Le concept d'état est fondamental dans un système dynamique. Il permet, en effet, de décrire l'évolution du système, et sa structure détermine si un système est concurrent ou non: si l'état est atomique, le système est séquentiel, mais s'il est distribué alors le système a le potentiel d'être concurrent.
Dans le cas des ECATNets simples, l'état est défini par une structure de *multi-ensemble* de termes. Chaque terme a la forme $< p, M(p) >$ où p est une place et *M(p)* son marquage. Le symbole $\otimes$ est utilisé pour décrire l'union des multi-ensembles de cette sorte.

Ainsi, si $P = \{p_1, p_2, .., p_n\}$ est l'ensemble des places d'un ECATNet simple marqué ($\mathcal{E}$, *M*). L'état de $\mathcal{E}$ est défini par:

$$st = \otimes_{i=1..n} < p_i, M(p_i) > \; = \; < p_1, M(p_1) > \otimes < p_2, M(p_2) > \otimes ..... \otimes < p_n, M(p_n) >$$

tel que $\forall i = 1..n$

$M(p_i) = \left[t_i^1\right] \oplus \left[t_i^2\right] \oplus ... \oplus \left[t_i^k\right]$ est le marquage de la place p_i

t_i^j est le $j^{ème}$ terme de la place p_i i.e. $t_i^j \in T_{\Sigma_{\sigma(p_i)}}(\varnothing)$

$\left[t_i^j\right]$ est le multi-ensemble contenant le le terme t_i^j.

Les opération $\otimes$ et $\oplus$ sont commutatives et associatives. De plus l'opération $\otimes$ est distributive par rapport à $\oplus$. Ceci signifie que l'on a la propriété:

$\forall i = 1..n \quad < p_i, M(p_i) > \; = \; < p_i, \oplus_{j=1..k} \left[t_i^j\right] > \; = \; \otimes_{j=1..k} < p_i, \left[t_i^j\right] >$

Cette structure distribuée de l'état d'un ECATNet permet d'exécuter en parallèle un ensemble de transitions franchissables. La concurrence et l'auto-concurrence sont alors facilement supportées par une telle structure.

3- Théorie de réécriture $\mathcal{T}_{\text{simp}}$ d'un ECATNet simple

3.1 Définition

Soit $\mathcal{E} = (Spec, R)$ un ECATNet simple où $Spec = (\Sigma, E)$ et $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$. La théorie de réécriture de l'ECATNet $\mathcal{E}$ est une théorie de réécriture étiquettée $\mathcal{T}_{\text{simp}} = (Sig_{simp}, \mathcal{E}_{simp}, \mathcal{L}_{simp}, \mathcal{R}_{simp})$ telle que Sig_{simp}, $\mathcal{E}_{simp}$, $\mathcal{L}_{simp}$, $\mathcal{R}_{simp}$ sont définis ci-dessous.

a) Signature Sig_{simp}

La signature de la théorie de réécriture $\mathcal{T}_{\text{simp}}$ d'un ECATNet simple est basée sur l'extension de la signature $m\Sigma$ de la spécification algébrique *mSpec* définie dans le *Chapitre 1*, par la signature relative à la définition de la structure de l'état d'un ECATNet simple telle qu'elle est présentée par la Figure 8.2.

$$Sig_{simp} = m\Sigma \cup Sig_{state_1}$$

$$\text{où } Sig_{state_1} = \left(\begin{array}{ll} \textbf{Sorts} & placemarking\ \ state \\ \textbf{Subsort} & placemarking < state \\ \textbf{Ops} & \\ & < p_i, \_ >: m_{\sigma(p_i)} \rightarrow placemarking \quad \{\text{pour chaque place } p_i\ \} \\ & \varnothing : \rightarrow state \\ & \_ \otimes \_ : state\ state \rightarrow state \end{array} \right)$$

Figure 8.2 : Signature de la théorie $\mathcal{T}_{\text{simp}}$.

b) Les équations $\mathcal{E}_{simp}$

L'ensemble des équations de la théorie $\mathcal{T}_{\text{simp}}$ d'un ECATNet simple est l'union d'une part des équations *mE* de la spécification *mSpec* définie dans le Chapitre 1, et d'autre part des ACI équations relatives à l'opération $\_ \otimes \_$ de composition des états.

$$\mathcal{E}_{simp} = mE \cup \mathcal{E}_{state_1}$$

$$\text{où } \mathcal{E}_{state_1} = \left(\begin{array}{l} \textbf{vars}\ \ s\ \ s_1\ s_2\ s_3 : state \\ \textbf{eqns} \\ \quad s \otimes \varnothing = s \\ \quad s_1 \otimes (s_2 \otimes s_3) = (s_1 \otimes s_2) \otimes s_3 \\ \quad s_1 \otimes s_2 = s_2 \otimes s_1 \end{array} \right)$$

Figure 8.3 : Equations de la théorie $\mathcal{T}_{\text{simp}}$.

c) Les étiquettes $\mathcal{L}_{simp}$

Les étiquettes $\mathcal{L}_{simp}$ de la théorie $\mathcal{T}_{\text{simp}}$ d'un ECATNet sont les identificateurs des transitions T, i.e. $\mathcal{L}_{simp}$ =T.

d) Les règles de réécritures $\mathcal{R}_{simp}$

Dans [BM-91][BMSB-93][BM-95] on trouve la façon suivant laquelle on associe les règles de réécriture à un ECATNet donné. Ces règles permettent de déduire l'état d'un ECATNet à partir d'un état précédent:

L'effet d'une transition *t* est représenté par une règle de réécriture de la forme :

$$t : L \rightarrow R \ \ if \ \ C .$$

L : représente les jetons *consommés* par la transition *t.*
R : représente les jetons *créés* par la transition *t.*
C : est la condition de la transition *t.*

De façon générale, à toute transition *t* adjacentes aux places $p_1, p_2, ..., p_n$, on associe une règle dont la forme générale est présentée par la Figure 8.4.

$$rl\ \ t: \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, DT(p_i,t) >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing >) \ \otimes \\ (\otimes_{i=n_2+1..n} < p_i, DT(p_i,t) >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing >) \ \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i,t) > \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i,t) >) \end{pmatrix}$$

$$\text{if } TC(t) \rightarrow true$$

où $\{p_1, p_2, ...p_{n_1}\} = {}^\bullet t - t^\bullet \quad \{p_{n_1+1}, p_{n1+2}, ...p_{n_2}\} = t^\bullet - {}^\bullet t \quad \{p_{n_2+1}, p_{n_2+2}, ...p_n\} = {}^\bullet t \cap t^\bullet$

$n_1 \le n_2 \le n$

Figure 8.4 : Règles de réécriture de la théorie $\mathcal{T}_{simp}$.

Notation :

${}^\bullet t$ *:* l'ensemble des places en entrée de *t.*

$t^\bullet$: l'ensemble des places en sortie de *t.*

Dans le cas où $t \notin dom(TC)$, la règle de réécriture associée à *t* est inconditionnelle.

Exemple

Les règles de réécriture associées au réseau de la Figure 8.1 sont :

$$rl\ \ t_1 :< p_1, a > \otimes < p_2, b > \otimes < p_3, \varnothing > \ \rightarrow \ < p_1, \varnothing > \otimes < p_2, c \oplus c > \otimes < p_3, a >$$

$$rl\ \ t_2 :< p_3, \varnothing > \ \rightarrow \ < p_3, a >$$

$$rl\ \ t_3 :< p_3, a \oplus a > \ \rightarrow \ < p_3, \varnothing >$$

3.2 Déduction dans $\mathcal{T}_{simp}$

Un système passe d'un état global *s* vers un autre état *s'* qui résulte d'une composition parallèle d'un ensemble de « *changements élémentaires* ». Ces changements sont décrits à l'aide des règles de réécriture modélisant le comportement dynamique de l'ECATNet. Ainsi la règle $u \rightarrow v$ if *c*, exprime que l'état partiel *u*, extrait de l'état global *s*, est à remplacer par l'état partiel *v* si la condition *c* est vérifiée. Ce qui fait passer l'état global *s* vers *s'*. Cette exécution correspond à une preuve du séquent $[s] \rightarrow [s']$ dans la théorie de réécriture $\mathcal{T}_{simp}$ **.**

a- Règles de déduction de $\mathcal{T}_{simp}$

Etant donné la théorie de réécriture $\mathcal{T}_{simp}$ d'un ECATNet simple, un séquent $[s] \rightarrow [s']$ est prouvable dans $\mathcal{T}_{simp}$ et on note $\mathcal{T}_{simp} \vdash [s] \rightarrow [s']$ si et seulement si $[s] \rightarrow [s']$ peut être obtenu par une application finie et concurrente des règles de déduction suivantes:

(i) Réflexivité: $\forall [s] : state \qquad \overline{s : [s] \rightarrow [s]}$

(ii) Congruence:

$$\frac{\forall [s_1],...,[s_n] : state \quad \forall f \in Sig_{simp_n} \qquad t_1 : [s_1] \rightarrow [s_1'] \ ... \ t_n : [s_n] \rightarrow [s_n']}{f(t_1,...,t_n) : f([s_1],..,[s_n]) \rightarrow f([s_1'],..,[s_n'])}$$

où Sig_{simp_n} est l'ensemble des fonctions de Sig_{simp} d'arité *n*.

(iii) Substitution:

$$\frac{\forall\ (\ r : [s(x_1,...,x_n)] \rightarrow [s'(x_1,...,x_n)]\ \ if \wedge_{i=1..k} [c_i(x_1,...,x_n)] \rightarrow [c_i'(x_1,...,x_n)]\ ) \in \mathcal{R}_{simp} \qquad t_1 : [w_1] \rightarrow [w_1'],..., \ t_n : [w_n] \rightarrow [w_n'] \qquad [c_1(\overline{w}/\overline{x})] \rightarrow [c_1'(\overline{w}/\overline{x})],...,[c_k(\overline{w}/\overline{x})] \rightarrow [c_k'(\overline{w}/\overline{x})]}{r(t_1,...,t_n) : [s(\overline{w}/\overline{x})] \rightarrow [s'(\overline{w'}/\overline{x})]}$$

où $\overline{w}/\overline{x}$ est une substitution des x_i *par* les w_i $\quad 1 \le i \le n$

(iv) Transitivité:

$$\forall [s_1],[s_2],[s_3] : state \quad \frac{t_1 : [s_1] \rightarrow [s_2] \quad t_2 : [s_2] \rightarrow [s_3]}{t_1 ; t_2 : [s_1] \rightarrow [s_3]}$$

(v) Décomposition $\quad \forall [s] : state \quad \forall p \in P \quad \forall\ m\ m_i \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \forall n \in \mathbb{N}$

$$\frac{\mu : [m_1 \oplus m_2 \oplus .. \oplus m_n] \rightarrow [m] \qquad t : [s] \rightarrow [< p, m >]}{d(\mu, t) : [s] \rightarrow [< p, m_1 > \otimes < p, m_2 > \otimes ... \otimes < p, m_n >]}$$

(vi) Composition $\quad \forall [s] : state \quad \forall p \in P \quad \forall\ m_i \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \forall n \in \mathbb{N}$

$$\frac{t : [s] \rightarrow [< p, m_1 > \otimes < p, m_2 > \otimes ... \otimes < p, m_n >]}{c(t) : [s] \rightarrow [< p, m_1 \oplus m_2 \oplus .. \oplus m_n >]}$$

Les règles de déductions permettent par une décomposition et une recomposition judicieuse de différents multi-ensembles des classes d'équivalence de termes, de trouver le calcul des ECATNets qui met en évidence le maximum de parallélisme.

b- Exemple de déduction

Par exemple, soit l'ECATNet de la Figure 8.1, la preuve ci-dessous dans la théorie $\mathcal{T}_{simp}$ permet de démontrer que l'état $< p_1, a > \otimes < p_2, c \oplus c \oplus c \oplus c > \otimes < p_3, a \oplus a \oplus a \oplus a >$ est accessible à partir de l'état initial $s_0 = < p_1, a \oplus a > \otimes < p_2, b \oplus c \oplus c > \otimes < p_3, a >$.

(1) $\tau_{simp} \vdash [s_0] \rightarrow [< p_1, a \oplus a > \otimes < p_2, b \oplus c \oplus c > \otimes < p_3, a >]$

règle de déduction reflexivité

(2) $\tau_{simp} \vdash [s_0] \rightarrow [< p_1, a > \otimes < p_1, a > \otimes < p_2, b \oplus c \oplus c > \otimes < p_3, a >]$

règle de déduction décomposition sur $< p_1, a \oplus a >$ *de* **(1)**

(3) $\tau_{simp} \vdash [s_0] \rightarrow [< p_1, a > \otimes < p_1, a > \otimes < p_2, b > \otimes < p_2, c \oplus c > \otimes < p_3, a >]$

règle de décomposition sur $< p_2, b \oplus c \oplus c >$ *de* **(2)**

(4) $\tau_{simp} \vdash [s_0] \rightarrow [< p_1, a > \otimes < p_1, a > \otimes < p_2, b > \otimes < p_2, c \oplus c > \otimes < p_3, a > \otimes < p_3, \varnothing > \otimes < p_3, \varnothing >]$

règle de déduction décomposition sur $< p_3, a >$ *de* **(3)**

(5) $\tau_{simp} \vdash [s_0] \rightarrow [\underline{< p_1, a > \otimes < p_2, b > \otimes < p_3, \varnothing >} \quad \otimes \underline{< p_3, \varnothing >} \otimes \underline{< p_3, \varnothing >} \otimes < p_1, a > \otimes < p_2, c \oplus c > \otimes < p_3, a >]$

$\downarrow \qquad \downarrow \qquad \downarrow$ *commutativité de* $\otimes$ *sur* **(4)**

(6) $\tau_{simp} \vdash [s_0] \rightarrow [\overbrace{< p_1, \varnothing > \otimes < p_2, c \oplus c > \otimes < p_3, a >}^{\downarrow t_1} \otimes \overbrace{< p_3, a >}^{\downarrow t_2} \otimes \overbrace{< p_3, a >}^{\downarrow t_2} \otimes < p_1, a > \otimes < p_2, c \oplus c > \otimes < p_3, a >]$

application concurrentes des règles de réécriture t_1 t_2 t_2 *sur* **(5)**

(7) $\tau_{simp} \vdash [s_0] \rightarrow [< p_1, \varnothing > \otimes < p_1, a > \otimes < p_2, c \oplus c > \otimes < p_2, c \oplus c > \otimes < p_3, a > \otimes < p_3, a > \otimes < p_3, a > \otimes < p_3, a >]$

commutativité de $\otimes$ *sur* **(6)**

(8) $\tau_{simp} \vdash [s_0] \rightarrow [< p_1, a > \otimes < p_2, c \oplus c \oplus c \oplus c > \otimes < p_3, a \oplus a \oplus a \oplus a >]$

règle de déduction composition sur **(7)**

Après avoir donné la sémantique formelle d'un ECATNet simple, nous allons montrer dans les trois sections suivantes comment modifier la notion d'état d'un ECATNet et sa théorie de réécriture pour considérer des aspects relatifs à la notion de *capacité*, de *contraintes contextuelles* et de *contraintes temporelles.*

IV- Sémantique des ECATNets à capacité

Un ECATNet à capacité est un ECATNet ayant éventuellement au moins une place de capacité *finie*, et tel que aucune de ses transitions n'exige pour son franchissement des contraintes contextuelles ni détruit lors de son franchissement tout le contenu d'une place.

1- Définition

$\mathcal{E}$ est un ECATNet à capacité si et seulement si les conditions suivantes sont vérifiées simultanément :

$$\begin{cases} - \ dom(IC) = \varnothing \\ - \ \forall (p,t) \quad \big((p,t) \in dom(DT) \ \rightarrow \ DT(p,t) \neq \forall\big) \end{cases}$$

Exemple :

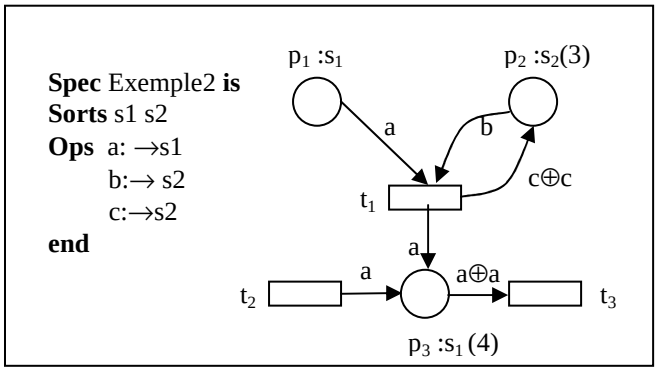
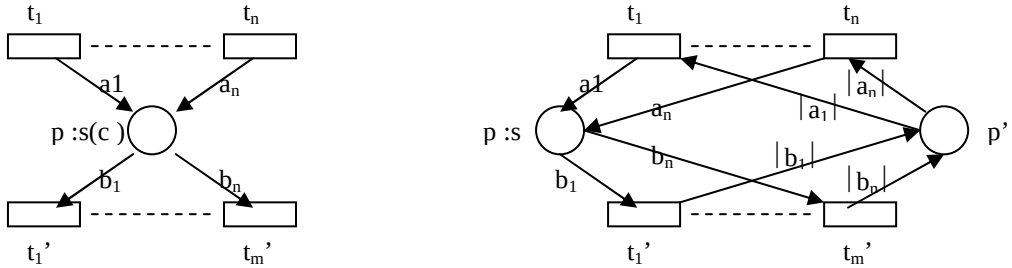


Figure 8.5 : Exemple d'un ECATNet à capacité.

La sémantique d'un ECATNet à capacité peut être donnée en définissant l'ECATNet simple (sans capacité) équivalent selon l'approche générale proposée dans [BRA-82] pour les réseaux de Petri. Cette approche nécessite l'ajout d'une place pour chaque place à capacité selon le schéma suivant :



Notre approche est différente et consiste à associer à chaque ECATNet à capacité une théorie de réécriture sans ajout de nouvelles places ni de nouveaux arcs à l'ECATNet. L'idée clef réside dans une définition adéquate de la notion d'état d'un ECATNet et une modification appropriée des règles de réécriture et des règles de déduction de la théorie de réécriture d'un tel ECATNet.

2- Etat d'un ECATNet à capacité

Dans le cas des ECATNets à capacité, l'état est défini par une structure de multi-ensemble de termes. Chaque terme a l'une des deux formes suivantes :

$$\begin{cases} < p, M(p) > \quad si \quad p \notin dom(Cap) \\ ou \\ < p, M(p), c > \quad si \quad p \in dom(Cap) \end{cases}$$

Dans un terme de la forme $< p, M(p), c >$ où p est une place et $M(p)$ son marquage, le nombre c représente le nombre de termes (jetons) pouvant être ajoutés dans la place p sans dépasser la capacité de p. Le symbole $\otimes$ est utilisé pour décrire l'union des multi-ensembles de cette sorte.

Ainsi, si P est l'ensemble des places d'un ECATNet marqué ($\mathcal{E}$, M). L'état de $\mathcal{E}$ est défini par:

$$st = (\otimes_{i=1..n_1} < p_i, M(p_i) >) \otimes (\otimes_{i=n_1+1..n} < p_i, M(p_i), c_i >)$$

où $\left(\text{P} = \{p_1, p_2, ..., p_n\} \quad p_{i_{1 \le i \le n_1}} \notin dom(Cap) \quad p_{i_{n_1+1 \le i \le n}} \in dom(Cap) \quad n_1 \le n \right)$

$$\left(c_i = Cap(p_i) - |M(p_i)|_{\sigma(p_i)} \right)$$

Exemple :

Soit le réseau à capacité représenté dans la Figure 8.5 où p_1 est de capacité infinie et p_2 et p_3 ont respectivement la capacité 3 et 4. Si l'on considère que le réseau est marqué par M tel que : $M(p_1) = a \oplus a \quad M(p_2) = b \oplus c \oplus c \quad M(p_3) = a \oplus a$, alors l'état de l'ECATNet correspondant est $< p_1, a \oplus a > \otimes < p_2, b \oplus c \oplus c, 0 > \otimes < p_3, a \oplus a, 2 >$.

3- Théorie de réécriture $\mathcal{T}_{cap}$ d'un ECATNet à capacité

3.1 Définition

Soit $\mathcal{E} = (Spec, R)$ un ECATNet tel que $Spec = (\Sigma, E)$ et $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$. La théorie de réécriture de l'ECATNet à capacité $\mathcal{E}$ est une théorie de réécriture étiquettée $\mathcal{T}_{cap} = (Sig_{cap}, \mathcal{E}_{cap}, \mathcal{L}_{cap}, \mathcal{R}_{cap})$ telle que Sig_{cap}, $\mathcal{E}_{cap}$, $\mathcal{L}_{cap}$, $\mathcal{R}_{cap}$ sont définis ci-dessous.

a) signature Sig_{cap}

La signature de la théorie de réécriture $\mathcal{T}_{cap}$ d'un ECATNet à capacité est une extension de la signature Sig_{simp} (des ECATNets simples) par les sortes et opérations décrites dans la Figue 8.6.

$$Sig_{cap} = Sig_{simp} \cup Sig_{state_2}$$

$$Sig_{state_2} = \left(\begin{array}{l} \textbf{Sorts} \quad cap.placemarking \quad nat_\infty \\ \textbf{Subsorts} \\ \qquad placemarking < cap.placemarking < state \\ \qquad nat < nat_\infty \\ \textbf{Ops} \\ \qquad \infty : \ \rightarrow nat_\infty \\ \qquad < p_i, \_ > : m_{\sigma(p_i)}\, nat_\infty \rightarrow cap.placemarking \quad \{\text{pour chaque place } p_i\} \end{array} \right)$$

Figure 8.6 : Signature de la théorie $\mathcal{T}_{cap}$.

La sorte nat_∞ définit l'ensemble des capacités possibles. C'est une supersorte des entiers naturels *nat*. L'opération ∞ est utilisée pour spécifier une capacité infinie.

b) Les équations $\mathcal{E}_{cap}$

L'ensemble des équations de la théorie de réécriture $\mathcal{T}_{cap}$ est construit en enrichissant les équations de $\mathcal{T}_{simp}$ par les équations qui décrivent les propriétés de l'opération ∞ selon la Figure 8.7.

$$\mathcal{E}_{cap} = \mathcal{E}_{cap} \cup \mathcal{E}_{state_2}$$

où $\mathcal{E}_{state_2} =$

vars $n : nat_\infty$

$m_i : m_{\sigma(p_i)}$ {pour chaque place p_i}

eqns

$\infty + n = \infty$

$\infty - n = \infty$

$\infty * n = \infty$

$< p_i, m_i >=< p_i, m_i, \infty >$ {pour chaque place p_i }

Figure 8.7 : Equations de la théorie $\mathcal{T}_{cap}$.

c) Les étiquettes $\mathcal{L}_{cap}$

Comme dans le cas de la théorie $\mathcal{T}_{simp}$, les étiquettes $\mathcal{L}_{cap}$ de la théorie $\mathcal{T}_{cap}$ d'un ECATNet à capacité sont les identificateurs des transitions *T* (i.e. $\mathcal{L}_{cap} = T$).

d) Les règles de réécritures $\mathcal{R}_{cap}$

A toute transition *t* de l'ECATNet à capacité on associe une règle de réécriture dont la forme est décrite par la Figure 8.8.

$$rl\ \ t : \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, \left|CT(p_i,t)\right|_{\sigma(p_i)} >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, DT(p_i,t), \left|CT(p_i,t)\right|_{\sigma(p_i)} - \left|DT(p_i,t)\right|_{\sigma(p_i)} >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, \left|DT(p_i,t)\right|_{\sigma(p_i)} >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i,t), 0 >) \end{pmatrix}$$

if $TC(t) \rightarrow true$

où $\{p_1, p_2, ...p_{n_1}\} = {}^\bullet t - t^\bullet$ $\{p_{n_1+1}, p_{n_1+2}, ...p_{n_2}\} = t^\bullet - {}^\bullet t$ $\{p_{n_2+1}, p_{n_2+2}, ...p_n\} = {}^\bullet t \cap t^\bullet$

$n_1 \le n_2 \le n$

Figure 8.8 : Règles de réécriture de la théorie $\mathcal{T}_{cap}$.

Exemple

Les règles de réécriture associées à l'ECATNet de la Figure 8.5 sont :

$rl\ \ t_1 :< p_1, a, 0 > \otimes < p_2, b, 1 > \otimes < p_3, \varnothing, 1 > \ \rightarrow\ < p_1, \varnothing, 1 > \otimes < p_2, c \oplus c, 0 > \otimes < p_3, a, 0 >$

$rl\ t_2 :< p_3, \varnothing, 1 > \ \rightarrow\ < p_3, a, 0 >$

$rl\ t_3 :< p_3, a \oplus a, 0 > \ \rightarrow\ < p_3, \varnothing, 2 >$

3.2 Déduction dans $\mathcal{T}_{cap}$

a- Règles de déduction de $\mathcal{T}_{cap}$

Les règles de déduction de $\mathcal{T}_{cap}$ sont les suivantes:

(i) Réflexivité: $\forall [s] : state \qquad \overline{s : [s] \rightarrow [s]}$

(ii) Congruence:

$$\frac{\forall [s_1],...,[s_n] : state \quad \forall f \in Sig_{cap_n} \qquad t_1 : [s_1] \rightarrow [s_1'] \ ... \ t_n : [s_n] \rightarrow [s_n']}{f(t_1,...,t_n) : f([s_1],..,[s_n]) \rightarrow f([s_1'],..,[s_n'])}$$

où Sig_{cap_n} est l'ensemble des fonctions de Sig_{cap} d'arité *n.*

(iii) Substitution:

$$\frac{\forall\ (r : [s(x_1,...,x_n)] \rightarrow [s'(x_1,...,x_n)]\ if \wedge_{i=1..k} [c_i(x_1,...,x_n)] \rightarrow [c_i'(x_1,...,x_n)]\ ) \in \mathcal{R}_{cap} \qquad t_1 : [w_1] \rightarrow [w_1'],..., t_n : [w_n] \rightarrow [w_n'] \qquad [c_1(\overline{w}/\overline{x})] \rightarrow [c_1'(\overline{w}/\overline{x})],...,[c_k(\overline{w}/\overline{x})] \rightarrow [c_k'(\overline{w}/\overline{x})]}{r(t_1,..,t_n) : [s(\overline{w}/\overline{x}] \rightarrow [s'(\overline{w'}/\overline{x})]}$$

où $\overline{w}/\overline{x}$ est une substitution des x_i *par* les $w_i \quad 1 \le i \le n$

(iv) Transitivité: $\forall [s_1],[s_2],[s_3] : state \quad \dfrac{t_1 : [s_1] \rightarrow [s_2] \quad t_2 : [s_2] \rightarrow [s_3]}{t_1;t_2 : [s_1] \rightarrow [s_3]}$

(v) Décomposition $\forall [s] : state \quad \forall p \in P \quad \forall\ m\ m_i \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \forall c\ c_i \in nat_\infty \quad \forall n \in \mathbb{N}$

$$\frac{\mu : [m_1 \oplus m_2 \oplus .. \oplus m_n] \rightarrow [m] \quad c : [c_1 + c_2 + .. + c_n] \rightarrow [c] \qquad t : [s] \rightarrow [< p, m, c >]}{d(\mu, c, t) : [s] \rightarrow [< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ... \otimes < p, m_n, c_n >]}$$

(vi) Composition $\forall [s] : state \quad \forall p \in P \quad \forall\ m_i \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \forall c_i \in nat_\infty \quad \forall n \in \mathbb{N}$

$$\frac{t : [s] \rightarrow [< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ... \otimes < p, m_n, c_n >]}{c(t) : [s] \rightarrow [< p, m_1 \oplus m_2 \oplus .. \oplus m_n, c_1 + c_2 + .. + c_n >]}$$

(vii) Ajout-capacité $\forall [s] : state \quad \forall p \in P \quad \forall\ m \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \qquad \dfrac{t : [s] \rightarrow [< p, m >]}{ac(t) : [s] \rightarrow [< p, m, \infty >]}$

(viii) Suppression-capacité $\forall [s] : state \quad \forall p \in P \quad \forall\ m \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \qquad \dfrac{t : [s] \rightarrow [< p, m, \infty >]}{sc(t) : [s] \rightarrow [< p, m >]}$

Notons que la différence entre les règles de déduction de $\mathcal{T}_{simp}$ et celles de $\mathcal{T}_{cap}$ réside dans la décomposition d'un état en sous-états 'élémentaires' et la composition de ces états en un état moins élémentaire. Dans la première théorie $\mathcal{T}_{simp}$ la décomposition d'un état consiste en une distribution des jetons entre les états élémentaires (et inversement pour la composition). Par contre dans la théorie $\mathcal{T}_{cap}$, la décomposition concerne également la distribution du nombre *c* (en un ensemble d'entiers) de jetons pouvant être ajoutés dans chaque place.
Les règles *Ajout-capacité* et *Suppression--capacité* permettent de considérer les places qui n'appartiennent pas au domaine de la fonction *Cap* comme des places à capacité infinie.

b- Exemple de déduction

Soit par exemple le réseau d'ECATNet de la Figure 8.5, la preuve ci-dessous dans la théorie $\mathcal{T}_{cap}$ permet de démontrer que l'état $< p_1, a > \otimes < p_2, c \oplus c \oplus c, 0 > \otimes < p_3, a \oplus a \oplus a \oplus a, 0 >$ est accessible à partir de l'état initial $s_0 = < p_1, aa > \otimes < p_2, b \oplus c, 1 > \otimes < p_3, a \oplus a, 2 >$

(1) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow [< p_1, a \oplus a > \otimes < p_2, b \oplus c, 1 > \otimes < p_3, a \oplus a, 2 >]$

règle de déduction refléxivité

(2) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow [< p_1, a \oplus a, \infty > \otimes < p_2, b \oplus c, 1 > \otimes < p_3, a \oplus a, 2 >]$

règle de déduction Ajout - capacité sur **(**1**)**

(3) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow [< p_1, a, 0 > \otimes < p_1, a, \infty > \otimes < p_2, b \oplus c, 1 > \otimes < p_3, a \oplus a, 2 >]$

règle de décomposition sur $< p_1, a \oplus a, \infty >$ *de* **(**2**)**

(4) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow [< p_1, a, 0 > \otimes < p_1, a, \infty > \otimes < p_2, b, 1 > \otimes < p_2, c, 0 > \otimes < p_3, a \oplus a, 2 >]$

règle de décomposition sur $< p_2, b \oplus c, 1 >$ *de* **(**3**)**

(5) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0 > \otimes < p_1, a, \infty > \otimes < p_2, b, 1 > \otimes < p_2, c, 0 > \otimes \\ < p_3, \varnothing, 1 > \otimes < p_3, \varnothing, 1 > \otimes < p_3, a \oplus a, 0 > \end{bmatrix}$

règle de décomposition $< p_3, a \oplus a, 2 >$ *sur de* **(**4**)**

(6) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0 > \otimes < p_2, b, 1 > \otimes < p_3, \varnothing, 1 > \otimes < p_3, \varnothing, 1 > \otimes \\ < p_1, a, \infty > \otimes < p_2, c, 0 > \otimes < p_3, a \oplus a, 0 > \end{bmatrix}$

commutativité de $\otimes$ *sur* **(**5**)**

(7) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1 > \otimes < p_2, c \oplus c, 0 > \otimes < p_3, a, 0 > \otimes < p_3, a, 0 > \otimes \\ < p_1, a, \infty > \otimes < p_2, c, 0 > \otimes < p_3, a \oplus a, 0 > \end{bmatrix}$

règles de réécriture t_1 *et* t_2 *en parallèle sur* **(**6**)**

(8) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1 > \otimes < p_1, a, \infty > \otimes < p_2, c \oplus c, 0 > \otimes < p_2, c, 0 > \otimes \\ < p_3, a, 0 > \otimes < p_3, a, 0 > \otimes < p_3, a \oplus a, 0 > \end{bmatrix}$

commutativité de $\otimes$ *sur***(**7**)**

(9) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow [< p_1, a, \infty > \otimes < p_2, c \oplus c \oplus c, 0 > \otimes < p_3, a \oplus a \oplus a \oplus a, 0 >]$

règle de composition sur **(**8**)**

(10) $\mathcal{T}_{simp} \vdash [s_0] \rightarrow [< p_1, a > \otimes < p_2, c \oplus c \oplus c, 0 > \otimes < p_3, a \oplus a \oplus a \oplus a, 0 >]$

règle de déduction Supprimer - capacité sur **(**9**)**

Théorème 8.1:

Si $\mathcal{E}$ est un ECATNet simple alors :

$$\forall [s],[s'] \in State_{\mathcal{T}_{simp}}(\mathcal{E}) \qquad \mathcal{T}_{simp} \vdash [s] \rightarrow [s'] \Rightarrow T_{cap} \vdash [s] \rightarrow [s']$$

où $State_{\mathcal{T}_{simp}}(\mathcal{E})$ est l'ensemble des états de $\mathcal{E}$ dans la théorie $\mathcal{T}_{simp}$.

Preuve : Annexe-1 preuve8.1.

V- Sémantique des ECATNets contextuels

Dans les ECATNets contextuels la condition d'entrée *IC* permet de tester la présence (contexte positif) ou l'absence (contexte négatif) de certains termes dans une place. Ceci permet de modéliser de façon naturelle les cas où certaines ressources peuvent être *lues sans être consommées* par une transition telle que par exemple dans un système de bases de données où des lectures simultanées d'une même donnée sont possibles. Par opposition aux ECATNets sans contraintes contextuelles, plusieurs transitions peuvent accéder au *même* jeton de façon concurrente.

1- Définition

$\mathcal{E}$ est un ECATNet contextuel si et seulement si l'une des conditions suivantes est *éventuellement* vérifiées:

$$\begin{cases} - \; dom(IC) \neq \varnothing \\ ou \\ - \; \exists (p,t) \quad \left((p,t) \in dom(DT) \; \wedge \; DT(p,t) = \forall \right) \end{cases}$$

Exemple :

Dans l'ECATNet contextuel de la Figure 8.9, la transition t_3 n'est franchissable que si la place p_3 ne contient aucun terme *b (contrainte contextuelle négative)*, et son franchissement détruit tous les termes contenus dans p_3. Par contre, le franchissement de la transition t_1 nécessite au moins la présence de deux termes *a* dans la place p_1 *(contrainte contextuelle positive),* mais elle n'en détruit qu'un seul si elle est franchie.

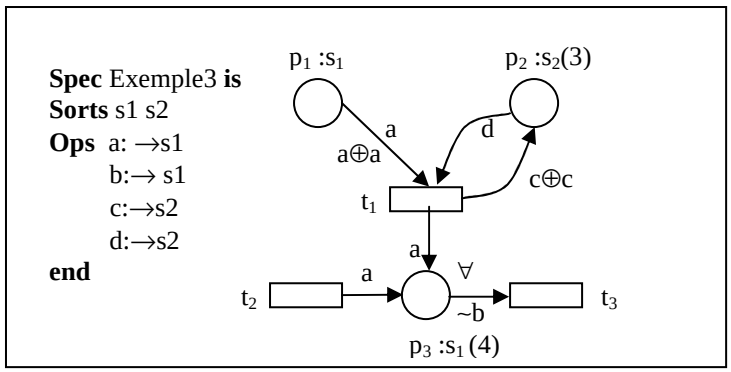


Figure 8.9 : Exemple d'un ECATNet contextuel.

2- Etat d'un ECATNet contextuel

Dans un ECATNet contextuel, l'action d'une transition ne se réduit pas à une destruction et/ou création de jetons dans les places qui lui sont adjacentes. En effet, elle peut tester la présence ou l'absence d'un jeton dans les places voisines. Cela suppose la possibilité à toute transition d'accéder au marquage *global* d'une place. Pour cela il est nécessaire de *dupliquer* le contenu *global* d'une place pour toutes les transitions qui l'utilisent pour des tests contextuels (présence/absence de jetons).

Par conséquent l'état d'un ECATNet contextuel est défini par un multi-ensemble de termes.
Chacun de ces termes a l'une des formes fondamentales suivantes :

$<p,m>$, $<p,m,c>$ ou $<p,m,c,M>$.

- Une occurrence d'un terme $<p,m>$ signifie la disponibilité du marquage m dans la place p (qui est de capacité infinie) pour une étape de réécriture.

- Une occurrence du terme $< p, m, c >$ indique la disponibilité du marquage m dans la place p qui peut recevoir encore un nombre de termes inférieur ou égal à c lors de la réécriture de m.

- Une occurrence du terme $< p, m, c, M >$ signifie que la place p a m comme un *sous-marquage* disponible pour une étape de réécriture (donc accessible en lecture et écriture) sachant que M est le marquage global de p (accessible en lecture seulement). Le nombre c précise le nombre de jetons qui peuvent être créés dans p lors de la réécriture de m.

- Pour permettre une réécriture simultanée des sous marquages d'une même place, on considère que le terme $<p,m,c>$ est équivalent à la composition par $\_\otimes\_$ d'un nombre arbitraire de termes de la forme $<p,m_i,c_i,m>$. Ceci peut être réalisé en ajoutant à la spécification un opérateur $[\![p|\_]\!] : nat \rightarrow state$ permettant de donner pour chaque place p, le nombre de termes de la forme $<p,m,c,M>$.
Une occurrence du terme $[\![p|n]\!]$ signifie que le marquage de p a été partitionné en n sous-marquages.

Ainsi, si P est l'ensemble des places d'un ECATNet contextuel marqué ($\mathcal{E}$, M). L'état de $\mathcal{E}$ est défini par:

$$st = \left(\otimes_{i=1..n_1} < p_i, m_i >\right) \otimes \left(\otimes_{i=n_1+1..n_2} < p_i, m_i, c_i >\right) \otimes \left(\otimes_{i=n_2+1..n} (\otimes_{j=1..k_i} < p_i, m_i^j, c_i^j, m_i >)\right) \otimes \left(\otimes_{i=n_2+1..n} [\![p_i | k_i]\!]\right)$$

$$\text{où} \quad \begin{pmatrix} P = \{p_1, p_2, ..., p_n\} & 0 \le n_1 \le n_2 \le n & \\ c_i = Cap(p_i) - |m_i|_{\sigma(p_i)} & +_{j=1..k_i} c_i^j = c_i & +_{j=1..k_i} m_i^j = m_i \end{pmatrix}$$

L'état initial d'un ECATNet a la forme $st = \left(\otimes_{i=1..n_1} < p_i, m_i >\right) \otimes \left(\otimes_{i=n_1+1..n} < p_i, m_i, c_i >\right)$. Lors de la réécriture, l'état d'un ECATNet passe par des états intermédiaires contenant des termes de la forme $< p, m, c, M >$ et $[\![p \mid n]\!]$, permettant aux transitions de procéder à des tests sur le contenu global des places.

Suite à cette définition, on peut établir les égalités suivantes :

$$(\otimes_{j=1..k_i} < p_i, m_i^j, c_i^j, M_i >) \otimes [\![p_i \mid k_i]\!] =< p_i, M_i, +_{j=1..k_i} c_i^j, M_i > [\![p_i \mid 1]\!]$$
$$< p_i, M_i, c_i >) =< p_i, M_i, c_i, M_i > [\![p_i \mid 1]\!]$$

Exemple :

Un état possible du réseau contextuel de la Figue 8.9 est :

$< p_1, a > \otimes < p_2, d, 0, d \oplus c > \otimes < p_2, c, 1, d \oplus c > \otimes < p_3, a \oplus a, 0, a \oplus a > \otimes < p_3, \varnothing, 2, a \oplus a > \otimes [\![p_2 \mid 2]\!] \otimes [\![p_3 \mid 2]\!]$

3- Théorie de réécriture $\mathcal{T}_{cont}$ d'un ECATNet contextuel

3.1 Définition

Soit $\mathcal{E} = (Spec, R)$ un ECATNet tel que $Spec = (\Sigma, E)$ et $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$. La théorie de réécriture de l'ECATNet contextuel $\mathcal{E}$ est une théorie de réécriture étiquettée $\mathcal{T}_{cont} = (Sig_{cont}, \mathcal{E}_{cont}, \mathcal{L}_{cont}, \mathcal{R}_{cont})$ telle que Sig_{cont}, $\mathcal{E}_{cont}$, $\mathcal{L}_{cont}$, $\mathcal{R}_{cont}$ sont définis ci-dessous.

a) Signature Sig_{cont}

La signature de la théorie de réécriture $\mathcal{T}_{cont}$ d'un ECATNet contextuel est Sig_{cont} définie par la Figure 8.10.

$$Sig_{cont} = Sig_{cap} \cup Sig_{state_3}$$

$Sig_{state_3} =$

Sorts $cont.placemarking \quad counter$

$gm_{\sigma(pi)}$ {pour chaque place p_i}

Subsorts

$placemarking < cap.placemarking < cont.placemarking < state$

$counter < state$

$m_{\sigma(pi)} < gm_{\sigma(pi)}$ { pour chaque place p_i }

Ops

$< p_i, \_, \_, \_ >: m_{\sigma(p_i)}\ nat_\infty\ gm_{\sigma(p_i)} \rightarrow cont.placemarking$

$[\![p_i \mid \_]\!]: nat \rightarrow counter$

? $: \rightarrow gm_{\sigma(p_i)}$

{pour chaque place p_i }

où ?: désigne l'état global indéfini

Figure 8.10 : Signature de la théorie $\mathcal{T}_{cont}$.

L'opération « ? » sert à représenter le marquage global indéfini d'une place. Cette opération est nécessaire notamment quand plusieurs transitions s'exécutent en parallèle et changent le marquage d'une place partagée *p*. Pour chacune des transitions le nouveau marquage global de la place *p* est indéfinie (« inconnu »).

b) Les équations $\mathcal{E}_{cont}$

L'ensemble des équations $\mathcal{E}_{cont}$ de la théorie $\mathcal{T}_{cont}$ est défini en enrichissant les équations $\mathcal{E}_{cap}$ par les équations relatives à l'opération « ? » et décrites dans la Figure 8.11.

$$\mathcal{E}_{cont} = \mathcal{E}_{cap} \cup \mathcal{E}_{state_3}$$

$$\mathcal{E}_{state_3} = \left(\begin{array}{l} \textbf{vars} \quad c_1\ c_2 : nat_\infty \\ \qquad m : gm_{\sigma(p)} \qquad m_1\ m_2 : m_{\sigma(p)} \quad \{\text{pour chaque place } p\} \\ \textbf{eqns} \\ \left[\begin{array}{l} < p, m_1, c_1, ? > \otimes < p, m_2, c_2, m > = < p, m_1, c_1, ? > \otimes < p, m_2, c_2, ? > \\ ? \ \oplus \ m = ? \\ < p, m, c > = < p, m, c, m > \otimes [\![p \,|1]\!] \end{array} \right] \text{pour chaque place p} \end{array} \right)$$

Figure 8.11 : Equations de la théorie $\mathcal{T}_{cont}$.

c) Les étiquettes $\mathcal{L}_{cont}$: Comme dans les théories précédentes $\mathcal{L}_{cont}$ =T.

d) Les règles de réécritures $\mathcal{R}_{cont}$

A toute transition *t* d'un l'ECATNet contextuel on associe une règle de réécriture de la forme :

vars $M_i : m_{\sigma(p_i)}$

$$rl \ \ t : \left(\begin{array}{l} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i,t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i,t)| - |DT(p_i,t)|, M_i >) \end{array} \right) \rightarrow \left(\begin{array}{l} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i,t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i,t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i,t), 0, ? >) \end{array} \right)$$

$$\textbf{if} \ \ TC(t) \wedge (\wedge_{i=1..n_1} ic_i) \wedge (\wedge_{i=n_2+1..n} ic_i) \rightarrow true$$

$$\text{où} \quad \left(\begin{array}{ll} \{p_1, p_2, ... p_{n_1}\} = {}^\bullet t - t^\bullet & \{p_{n_1+1}, p_{n_1+2}, ... p_{n_2}\} = t^\bullet - {}^\bullet t \quad \{p_{n_2+1}, p_{n_2+2}, ... p_n\} = {}^\bullet t \cap t^\bullet \\ dt_i \equiv M_i & \text{si } DT(p_i,t) = \forall \\ dt_i \equiv DT(p_i,t) & \text{si } DT(p_i,t) \neq \forall \\ ic_i \equiv \alpha \subseteq_{\sigma(p_i)} M_i & \text{si } IC(p_i,t) = \alpha \wedge \alpha \in mT_{\Sigma_{\sigma(p_i)}}(\varnothing) \\ ic_i \equiv not(\alpha \subseteq_{\sigma(p_i)} M_i) & \text{si } IC(p_i,t) = \sim \alpha \wedge \alpha \in mT_{\Sigma_{\sigma(p_i)}}(\varnothing) \\ ic_i \equiv M_i ==_{\sigma(p_i)} \varnothing_{\sigma(p_i)} & \text{si } IC(p_i,t) = empty \\ ic_i \equiv true & \text{si } (p_i,t) \notin dom(IC) \end{array} \right)$$

Figure 8.12 : Règles de réécriture de la théorie $\mathcal{T}_{cont}$.

Exemple

Les règles de réécriture associées au réseau de la Figure 8.9 sont :

Vars $M_1 : m_{s_1}$ $M_2 : m_{s_2}$ $M_3 : m_{s_1}$

$rl\ \ t_1 :< p_1, a, 0, M_1 > \otimes < p_2, d, 1, M_2 > \otimes < p_3, \varnothing, 1, M_3 > \ \rightarrow < p_1, \varnothing, 1, ? > \otimes < p_2, c \oplus c, 0, ? > \otimes < p_3, a, 0, ? >$

if $(a \oplus a \subseteq M_1) \rightarrow true$

$rl\ \ t_2 :< p_3, \varnothing, 1, M_3 > \ \rightarrow\ < p_3, a, 0, ? >$

$rl\ \ t_3 :< p_3, M_3, 0, M_3 > \ \rightarrow\ < p_3, \varnothing, |M_3|, ? >$ **if** $($**not**$(b \subseteq M_3)) \rightarrow true$

3.2 Déduction dans $\mathcal{T}_{cont}$

a- Règles de déduction de $\mathcal{T}_{cont}$

Les règles de déduction de T_{cont} sont les suivantes:

(i) Réflexivité: $\forall [s] : state \qquad \overline{s : [s] \rightarrow [s]}$

(ii) Congruence:

$$\forall [s_1], ..., [s_n] : state \quad \forall f \in Sig_{cont_n}$$

$$\frac{t_1 : [s_1] \rightarrow [s_1'] \ \ldots \ t_n : [s_n] \rightarrow [s_n']}{f(t_1, ..., t_n) : f([s_1], .., [s_n]) \rightarrow f([s_1'], .., [s_n'])}$$

où Sig_{cont_n} est l'ensemble des fonctions de Sig_{cont} d'arité *n.*

(iii) Substitution:

$$\forall\ (r : [s(x_1, ..., x_n)] \rightarrow [s'(x_1, ..., x_n)]\ \ if \wedge_{i=1..k} [c_i(x_1, ..., x_n)] \rightarrow [c_i'(x_1, ..., x_n)]\ ) \in \mathcal{R}_{cont}$$

$$\frac{t_1 : [w_1] \rightarrow [w_1'], ..., \ t_n : [w_n] \rightarrow [w_n'] \qquad [c_1(\overline{w}/\overline{x})] \rightarrow [c_1'(\overline{w}/\overline{x})], ..., [c_k(\overline{w}/\overline{x})] \rightarrow [c_k'(\overline{w}/\overline{x})]}{r(t_1, .., t_n) : \ [s(\overline{w}/\overline{x}] \rightarrow [s'(\overline{w'}/\overline{x})]}$$

où $\overline{w}/\overline{x}$ est une substitution des x_i *par*les $w_i \quad 1 \le i \le n$

(iv) Transitivité:

$$\forall [s_1], [s_2], [s_3] : state \quad \frac{t_1 : [s_1] \rightarrow [s_2] \qquad t_2 : [s_2] \rightarrow [s_3]}{t_1 ; t_2 : [s_1] \rightarrow [s_3]}$$

(v) Décomposition : $\forall [s]$**:** *state* $\forall p \in P \quad \forall M\ m\ m_i \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \forall c\ c_i \in nat_\infty \ \forall n\ k \in nat$

$$\frac{\begin{array}{c}\mu : [m_1 \oplus m_2 \oplus ... \oplus m_k] \rightarrow [m] \\ c : [c_1 + c_2 + .. + c_k] \rightarrow [c] \\ t : [s] \rightarrow [< p, m, c, M > \otimes [\![p | n]\!]]\end{array}}{d(\mu, c, t) : \ [s] \rightarrow [(\otimes_{i=1..k} < p, m_i, c_i, M >) \otimes [\![p | n + (k-1)]\!]]}$$

(vi) Composition : $\forall [s]: state \ \forall p \in P \quad \forall M \ m_i \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \forall c_i \in nat_\infty \quad \forall n \ k \in nat$

$$\frac{t:[s]\rightarrow[< p,m_1,c_1,M > \otimes < p,m_2,c_2,M > \otimes ....\otimes < p,m_k,c_k,M > \otimes [\![p|n]\!]]}{c(t):\ [s]\rightarrow[< p,m_1 \oplus m_2 \oplus ... \oplus m_k,\ c_1 + c_2 + .. + c_k,\ M > \otimes [\![p|n-(k-1)]\!]]}$$

(vii) Ajout-capacité : $\forall [s]: state \ \forall p \in P \ \forall m \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \dfrac{t:[s]\rightarrow[< p,m >]}{ac(t):[s]\rightarrow[< p,m,\infty >]}$

(viii) Suppression-capacité : $\forall [s]: state \ \forall p \in P \ \forall m \in mT_{\Sigma_{\sigma(p)}}(\varnothing) \quad \dfrac{t:[s]\rightarrow[< p,m,\infty >]}{sc(t):[s]\rightarrow[< p,m >]}$

(ix) Ajout-marquage global :

$$\forall [s]: state \ \forall p \in P \ \forall M \in mT_{\Sigma_{\sigma(p)}} \ \forall c \in nat_\infty \ \frac{t:[s]\rightarrow[< p,M,c >]}{am(t):[s]\rightarrow[< p,M,c,M > \otimes [\![p|1]\!]]}$$

(x) Suppression-marquage global :

$$\forall [s]: state \ \forall p \in P \ \forall M \ M' \in mT_{\Sigma_{\sigma(p)}} \ \forall c \in nat_\infty \ \frac{t:[s]\rightarrow[< p,M,c,M' > \otimes [\![p|1]\!]]}{sm(t):[s]\rightarrow[< p,M,c >]}$$

b- Exemple de déduction

Soit le réseau d'ECATNet de la Figure 8.9, une preuve dans $\mathcal{T}_{cont}$ permettant de démontrer que l'état $s =< p_1, a, \infty > \otimes < p_2, c \oplus c, 1 > \otimes < p_3, a \oplus a, 2 >$ est accessible à partir de l'état initial $s_0 =< p_1, a \oplus a, \infty > \otimes < p_2, d, 2 > \otimes < p_3, a \oplus c, 2 >$,est donnée par la déduction suivante.

(1) $\mathcal{T}_{cont} \vdash [s_0] \rightarrow [< p_1, a \oplus a, \infty > \otimes < p_2, d, 2 > \otimes < p_3, a \oplus c, 2 >]$

règle de déduction refléxivité

(2) $\mathcal{T}_{cont} \vdash [s_0] \rightarrow [< p_1, a \oplus a, \infty, a \oplus a > \otimes [\![p_1|1]\!] \otimes < p_2, d, 2 > \otimes < p_3, a \oplus c, 2 >]$

règle de déduction Ajout - marquage global sur $< p_1, a \oplus a, \infty >$ *de* (1)

(3) $\mathcal{T}_{cont} \vdash [s_0] \rightarrow [< p_1, a \oplus a, \infty, a \oplus a > \otimes [\![p_1|1]\!] \otimes < p_2, d, 2, d > \otimes [\![p_2|1]\!] \otimes < p_3, a \oplus c, 2 >]$

règle de déduction Ajout - marquage global sur $< p_2, d, 2 >$ *de* (2)

(4) $\mathcal{T}_{cont} \vdash [s_0] \rightarrow [< p_1, a \oplus a, \infty, a \oplus a > \otimes [\![p_1|1]\!] \otimes < p_2, d, 2, d > \otimes [\![p_2|1]\!] \otimes < p_3, a \oplus c, 2, a \oplus c > \otimes [\![p_3|1]\!]]$

règle de déduction Ajout - marquage global sur $< p_3, a \oplus c, 2 >$ *de* (3)

(5) $\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1|2]\!] \otimes \\ < p_2, d, 2, d > \otimes [\![p_2|1]\!] \otimes \\ < p_3, a \oplus c, 2, a \oplus c > \otimes [\![p_3|1]\!] \end{bmatrix}$

règle de décomposition sur $< p_1, a \oplus a, \infty, a \oplus a >$ *de* (4)

(6) $\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1|2]\!] \otimes \\ < p_2, d, 1, d > \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2|2]\!] \otimes \\ < p_3, a \oplus c, 2, a \oplus c > \otimes [\![p_3|1]\!] \end{bmatrix}$

règle de décomposition sur $< p_2, d, 2, d >$ *de* (5)

$$(7)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, d, 1, d > \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, \varnothing, 1, a \oplus c > \otimes < p_3, \varnothing, 1, a \oplus c > \otimes < p_3, a \oplus c, 0, a \oplus c > \otimes [\![p_3 \mid 3]\!] \end{bmatrix}$$

règle de décomposition sur $< p_3, a \oplus c, 2, a \oplus c >$ *de* (6)

$$(8)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_2, d, 1, d > \otimes < p_3, \varnothing, 1, a \oplus c > \otimes \\ < p_3, \varnothing, 1, a \oplus c > \otimes \\ < p_3, a \oplus c, 0, a \oplus c > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes [\![p_3 \mid 3]\!] \end{bmatrix}$$

commutativité de $\otimes$ *sur* (7)

$$(9)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_2, c \oplus c, 0, ? > \otimes < p_3, a, 0, ? > \otimes \\ < p_3, a, 0, ? > \otimes \\ < p_3, \varnothing, 2, ? > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes [\![p_3 \mid 3]\!] \end{bmatrix}$$

règle de réécriture $t_1\ t_2\ t_3$ *appliquées en parallèle sur* (8)

$$(10)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, c \oplus c, 0, ? > \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, a, 0, ? > \otimes < p_3, a, 0, ? > \otimes < p_3, \varnothing, 2, ? > \otimes [\![p_3 \mid 3]\!] \end{bmatrix}$$

commutativité de $\otimes$ *sur* (9)

$$(11)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_1, a, \infty, ? > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, c \oplus c, 0, ? > \otimes < p_2, \varnothing, 1, ? > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, a, 0, ? > \otimes < p_3, a, 0, ? > \otimes < p_3, \varnothing, 2, ? > \otimes [\![p_3 \mid 3]\!] \end{bmatrix}$$

application des équations de l'opération "?" sur (10)

$$(12)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, \infty, ? > \otimes [\![p_1 \mid 1]\!] \otimes \\ < p_2, c \oplus c, 1, ? > [\![p_2 \mid 1]\!] \otimes \\ < p_3, a \oplus a, 2, ? > \otimes [\![p_3 \mid 1]\!] \end{bmatrix}$$

règle de déduction composition sur (11)

$$(13)\,\mathcal{T}_{cont} \vdash [s_0] \rightarrow [< p_1, a, \infty > \otimes < p_2, c \oplus c, 1 > \otimes < p_3, a \oplus a, 2 >]$$

règle de déduction Suppression - marquage global sur (12)

Théorème 8.2

Soit $\mathcal{E}$ un ECATNet à capacité tel que $dom(IC) = \varnothing$ *et* $\forall (p,t)\big((p,t) \in dom(DT) \rightarrow DT(p,t) \neq \forall\big)$ alors :

$$\forall [s][s'] \in State_{\mathcal{T}_{cap}}(\mathcal{E}) \quad on\ a \quad \mathcal{T}_{cap} \vdash [s] \rightarrow [s'] \quad \Rightarrow \quad \mathcal{T}_{cont} \vdash [s] \rightarrow [s']$$

où $State_{\mathcal{T}_{cap}}(\mathcal{E})$ est l'ensemble des états de $\mathcal{E}$ dans la théorie $\mathcal{T}_{cap}$.

Preuve : Annexe-1 preuve8-2.

VI- Sémantique des ECATNets temporels

1- Définition

Un ECATNet temporel est un ECATNet contextuel où éventuellement certaines transitions ont des contraintes temporelles (i.e le domaine de *IS* est éventuellement non vide)

Exemple : Dans l'ECATNet de la Figure 8.13, *IS(t1)= [0,3], IS(t2)=[2,4]* et *IS(t3)= [0,∝[*.

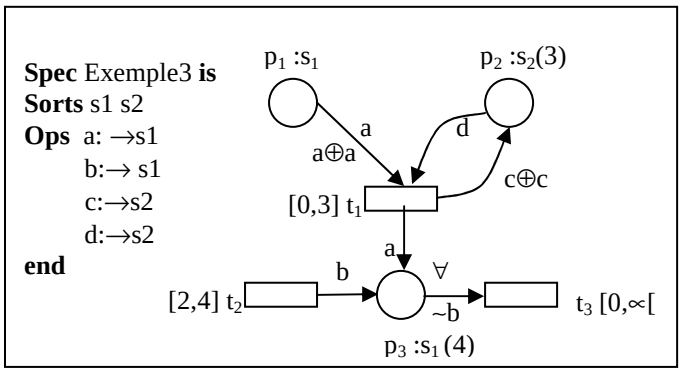


Figure 8.13 : Exemple d'un ECATNet temporel.

2- Etat d'un ECATNet temporel

Dans ce modèle, la présence d'un terme dans une place commande une action à laquelle est associé un intervalle d'exécution. Pour prendre en compte les contraintes temporelles des actions associées aux transitions, on modifie les règles de franchissement des transitions. Ainsi, si un multi-ensemble de termes *m* est créé dans une place *p* à un instant θ et si à partir de cette place, une transition dont l'intervalle de statique est $[min, max]$, devient sensibilisée pour *m*, alors *t* ne peut être franchie (et donc *m* ne peut être supprimé de *p*) que dans l'intervalle de temps $[\theta+min, \theta+max]$. C'est pourquoi nous supposons que les termes du marquage d'un ECATNet temporel ont deux états : *disponible* et *indisponible*. Le passage d'un sous-marquage *m* du réseau de l'état *disponible* à l'état i*ndisponible* s'effectue au moment où une transition *t* est sensibilisée pour *m*. Le sous-marquage *m* passe de l'état *indisponible* à l'état *disponible* si la transition *t* n'est pas franchie.

La définition de l'état d'un ECATNet temporel est présentée de façon progressive en faisant abstraction, dans un premier temps, des notions de capacité et de contraintes contextuelles.

➢ *Etat d'un terme dans le marquage d'une place*

Un terme (ou jeton) dans une place *p* a deux états :

- *Disponible* : s'il peut être utilisé pour le franchissement des transitions.
- *Indisponible* :s'il est réservé pour le franchissement d'une transition sensibilisée et dont les contraintes temporelles ne sont pas encore vérifiées. Ce jeton est alors réservé pour être utilisé par la transition en question.

➢ *Etat d'une place*

L'état d'une place *p* dans ECATNet temporel a la forme générale :

$$< p, m > \otimes < p, m_1^{'}, t_1, \tau_1 > \otimes < p, m_2^{'}, t_2, \tau_2 > \otimes ... \otimes < p, m_{n'}^{'}, t_{n'}, \tau_{n'} >$$

où $< p, m >$ est le marquage *disponible* dans p .

$< p, m_1^{'}, t_1, \tau_1 > \otimes ... \otimes < p, m_{n'}^{'}, t_{n'}, \tau_{n'} >$ est le marquage *indisponible*

Un terme de la forme $< p, m, t, \tau >$ signifie que le multi-ensemble *m* de la place *p* est réservé pour le franchissement de la transition *t* sensibilisée depuis l'instant τ .

➢ *Etat de l'ensemble des places*

L'état de l'ensemble de toutes les places d'un ECATNets temporel est donné par le terme :

$$\otimes_{i=1..n} \left(< p_i, m_i > \otimes \left(\otimes_{t_j \in Ts} < p_i, m_i^j, t_j, \tau_j > \right) \right)$$

où $P = \{ p_1, p_2, ..., p_n \}$

Ts **:** l'ensemble des transitions sensibilisées

m_i^j **:** le sous-marquage de p_i pour lequel la transition t_j est sensibilisée

A l'état initial tout le marquage est disponible et l'état de l'ensemble des places prend la forme :

$$\otimes_{i=1..n} < p_i, m_i > .$$

➢ *Forme générale de l'état d'un ECATNet Temporel*

L'état d'un ECATNet temporel est caractérisé par :

- L'état de l'ensemble des places
- L'état de l'horloge i.e : la valeur du temps de la forme $< H = h >$

$$\otimes_{i=1..n} \left(< p_i, m_i > \otimes \left(\otimes_{j=1..k} < p_i, m_i^j, t_j, \tau_j > \right) \right) \otimes < H = h >$$

Exemple :

L'état $< p_1, b > \otimes < p_2, a > \otimes < p_1, a \oplus a, t_1, 3 > \otimes < H = 4 >$ signifie qu'à l'état actuel l'horloge indique 4 unités de temps il y a un terme *b* disponible dans p_1 et un terme *a* disponible dans p_2 et il y a deux termes *a* dans p_1 réservés pour le franchissement de t_1 depuis l'instant 3.

Dans la définition de l'état d'un ECATNet temporel présentée ci-dessus les informations relatives aux capacités et aux contraintes contextuelles ont été omises. La prise en compte de telles informations donne une forme générale plus complexe définie comme suit :

$$st = \; < H = h > \; \otimes \; \left(\otimes_{i=1..n_1} \; < p_i, m_i >\right) \; \otimes \; \left(\otimes_{i=n_1+1..n_2} < p_i, m_i, c_i >\right) \otimes$$
$$\left(\otimes_{i=n_2+1..n}\left(\otimes_{j=1..k_i} < p_i, m_i^j, c_i^j, m_i >\right)\otimes\left(\otimes_{j=1..l_i} < p_i, m_i^j, c_i^j, m_i, t_j, \tau_j >\right)\right) \; \otimes \; \left(\otimes_{i=n_2+1..n} [\![p_i | k_i + l_i]\!]\right)$$

où $\begin{pmatrix} P = \{p_1, p_2, ..., p_n\} & 0 \le n_1 \le n_2 \le n \\ c_i = Cap(p_i) - |m_i|_{\sigma(p_i)} & \left(+_{j=1..k_i} c_i^j\right) + \left(+_{j=1..l_i} c_i^j\right) = c_i \qquad \left(+_{j=1..k_i} m_i^j\right) + \left(+_{j=1..l_i} m_i^j\right) = m_i \end{pmatrix}$

La signification de l'occurrence de chaque forme de terme est précisée comme suit

- $< H = h >$: L'horloge H indique h unités de temps

- $< p, m >$: le marquage m est disponible dans la place p (qui est de capacité infinie) pour une étape de réécriture.

- $< p, m, c >$: la place p a un marquage disponible m et on peut y créer encore un nombre c de termes.

- $< p, m, c, M >$: la place p ayant un marquage global M (accessible en lecture seulement),a un sous-marquage disponible m (accessible en lecture et écriture) et on peut y créer encore au moins un nombre c de termes.

- $< p, m, c, M, t, \tau >$: la place p ayant un marquage global M (accessible en lecture seulement), a un sous-marquage m (accessible en lecture et écriture) réservé par la transition t sensibilisée depuis l'instant τ . Le franchissement de la transition t ajoutera un nombre c de termes à p.

- $[\![p | k]\!]$: le nombre d'occurrences de termes de la forme $< p, m, c, M >$ ou $< p, m, c, M, t, \tau >$ est k

L'état initial d'un ECATNet temporel a la forme

$$st = < H = 0 > \otimes \left(\otimes_{i=1..n_1} < p_i, m_i >\right) \otimes \left(\otimes_{i=n_1+1..n} < p_i, m_i, c_i >\right).$$

Lors de la réécriture, l'état d'un ECATNet passe par des états intermédiaires contenant des termes de la forme $< p, m, c, M >$, $< p, m, c, M, t, \tau >$ et $[\![p | n]\!]$, permettant aux transitions de procéder à des tests sur le contenu global des places et à réserver des termes pour les transitions sensibilisées.

Exemple : Un état possible de l'ECATNet temporel de la Figure 8.13 est :

$$< p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d \oplus c, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes$$
$$< p_1, a, \infty, a \oplus a > \otimes < p_2, c, 1, d \oplus c > \otimes < p_3, \varnothing, 3, \varnothing > \otimes [\![p_1 | 2]\!] \otimes [\![p_2 | 2]\!] \otimes [\![p_3 | 2]\!]$$

3- Théorie de réécriture T_{temp} d'un ECATNet temporel

3.1 Définition

Soit $\mathcal{E}t$ =($\mathcal{E}$,*IS*) un ECATNet temporel tel que $\mathcal{E}$ est un ECATNet contextuel $\mathcal{E}$=(*Spec*,*R*) $Spec = (\Sigma, E)$ et $R = (P, T, \sigma, Cap, IC, DT, CT, TC)$. La théorie de réécriture de l'ECATNet temporel $\mathcal{E}t$ est une théorie de réécriture étiquettée $\mathcal{T}_{temp} = (Sig_{temp}, \mathcal{E}_{temp}, \mathcal{L}_{temp}, \mathcal{R}_{temp})$ telle que Sig_{temp}, $\mathcal{E}_{temp}$, $\mathcal{L}_{temp}$, $\mathcal{R}_{temp}$ sont définis ci-dessous.

a) signature Sig_{temp}

La signature de la théorie de réécriture $\mathcal{T}_{temp}$ d'un ECATNet temporel est basée sur la signature de $\mathcal{T}_{cont}$. Celle-ci est étendue par des sortes et opérations relatives au temps comme le précise la Figure 8.14.

$$Sig_{temp} = Sig_{cont} \cup Sig_{state_4}$$

$Sig_{state_4} =$

Sorts $temp.placemarking \quad transition \quad timer \quad time.copy \quad time.counter$

Subsorts

$placemarking < cap.placemarking < cont.placemarking < temp.placemarking < state$

$timer < state \quad time.copy < state \quad time.counter < state$

Ops

$t_i : \rightarrow transition \quad \{\text{pour chaque transition } t_i\}$

$< p_i, \_,\_,\_,\_ >: m_{\sigma(p_i)}\ nat_\infty\ gm_{\sigma(p_i)}\ transition\ nat \rightarrow temp.placemarking \quad \{\text{pour chaque place } p_i\}$

$< H = \_ >: nat \rightarrow timer$

$[H = \_\,] : nat \rightarrow time.copy$

$[\![H | \_]\!] : nat \rightarrow time.counter$

Figure 8.14 : Signature de la théorie $\mathcal{T}_{temp}$.

b) Les équations $\mathcal{E}_{temp}$

L'ensemble des équations de la théorie de réécriture $\mathcal{T}_{temp}$ d'un ECATNet temporel est l'ensemble des équations $\mathcal{E}_{cont}$ étendue aux équations relatives à l'horloge comme le précise la Figure 8.15.

$$\mathcal{E}_{temp} = \mathcal{E}_{cont} \cup \mathcal{E}_{state_4}$$

$\mathcal{E}_{state_4} =$ **Var** n :nat

Eqns $< H = n >= [H = n] \otimes [\![H | 1]\!]$

Figure 8.15 : Equations de la théorie $\mathcal{T}_{temp}$.

c) Les étiquettes $\mathcal{L}_{temp}$

A chaque transition *t* correspondent trois règles de réécriture ayant respectivement les étiquettes : *t* (pour franchissement de *t), sens.t* (pour la sensibilisation de *t*) et *desens.t* (pour la désensibilisation de *t*). Par conséquent, l'ensemble des étiquettes est défini par :

$$\mathcal{L} = T \cup \{sens.t / t \in T\} \cup \{desens.t / t \in T\}.$$

d) Les règles de réécritures $\mathcal{R}_{temp}$

A chaque transition *t* on associe trois règles de réécriture étiquetées *sens.t , desens.t, et t* où:
- *sens.t* : est la règle de réécriture qui sert à *réserver* un marquage pour la transition *t*.
- *desens.t* : sert à *libérer* le marquage réservé par *t* (dans le cas où *t* n'a pas été franchie)
- *t* : est la règle correspondant au *franchissement* de *t*.

vars $M_i : m_{\sigma(p_i)}$ $h, h_1 : nat$

rl *sens.t* :
$$\begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i,t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i,t)| - |DT(p_i,t)|, M_i >) \\ \otimes [H = h] \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i, t, h >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i,t)|, M_i, t, h >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i,t)| - |DT(p_i,t)|, M_i, t, h >) \\ \otimes [H = h] \end{pmatrix}$$

if $(TC(t) \rightarrow true) \wedge (\wedge_{i=1..n_1} ic_i) \wedge (\wedge_{i=n_2+1..n} ic_i) \rightarrow true$

rl *desens.t* :
$$\begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i, t, h >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i,t)|, M_i, t, h >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i,t)| - |DT(p_i,t)|, M_i, t, h >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i,t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i,t)| - |DT(p_i,t)|, M_i >) \end{pmatrix}$$

rl *t* :
$$\begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i, t, h >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i,t)|, M_i, t, h >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i,t)| - |DT(p_i,t)|, M_i >, t, h) \\ \otimes [H = h_1] \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i,t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i,t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i,t), 0, ? >) \\ \otimes [H = h_1] \end{pmatrix}$$

if $\begin{pmatrix} (TC(t) \rightarrow true) \wedge \ (\wedge_{i=1..n_1} ic_i) \wedge (\wedge_{i=n_2+1..n} ic_i) \wedge \\ (\min(t) \le h_1 - h) \ \wedge (h_1 - h \le \max(t)) \end{pmatrix} \rightarrow true$

où
$$\begin{pmatrix} \{p_1, p_2, ... p_{n_1}\} = {}^\bullet t - t^\bullet \quad \{p_{n_1+1}, p_{n_1+2}, ... p_{n_2}\} = t^\bullet - {}^\bullet t \quad \{p_{n_2+1}, p_{n_2+2}, ... p_n\} = {}^\bullet t \cap t^\bullet \\ dt_i \equiv M_i \qquad \text{si } DT(p_i,t) = \forall \\ dt_i \equiv DT(p_i,t) \qquad \text{si } DT(p_i,t) \neq \forall \\ ic_i \equiv \alpha \subseteq_{\sigma(p_i)} M_i \qquad \text{si } IC(p_i,t) = \alpha \wedge \alpha \in mT_{\Sigma\sigma(p_i)}(\varnothing) \\ ic_i \equiv not(\alpha \subseteq_{\sigma(p_i)} M_i) \qquad \text{si } IC(p_i,t) = {\sim}\alpha \wedge \alpha \in mT_{\Sigma\sigma(p_i)}(\varnothing) \\ ic_i \equiv M_i ==_{\sigma(p_i)} \varnothing_{\sigma(p_i)} \qquad \text{si } IC(p_i,t) = empty \\ ic_i \equiv true \qquad \text{si } (p_i,t) \notin dom(IC) \\ (\min(t), \max(t)) \equiv (\downarrow IS(t), \uparrow IS(t)) \end{pmatrix}$$

Figure 8.16 : Règles de réécriture de la théorie $\mathcal{T}_{temp}$.

Exemple

Les règles de réécriture associées à l'ECATNet temporel de la Figure 8.13 sont :

Vars $M_1 : m_{s_1}$ $M_2 : m_{s_2}$ $M_3 : m_{s_1}$

rl $sens.t_1$ **:** $\begin{pmatrix} < p_1, a, 0, M_1 > \otimes \\ < p_2, d, 1, M_2 > \otimes \\ < p_3, \varnothing, 1, M_3 > \otimes \\ [H = h] \end{pmatrix} \rightarrow \begin{pmatrix} < p_1, a, 0, M_1, t_1, h > \otimes \\ < p_2, d, 1, M_2, t_1, h > \otimes \\ < p_3, \varnothing, 1, M_3, t_1, h > \otimes \\ [H = h] \end{pmatrix}$ **if** $(a \oplus a \subseteq M_1) \rightarrow true$

rl $desens.t_1$ **:** $\begin{pmatrix} < p_1, a, 0, M_1, t_1, h > \otimes \\ < p_2, d, 1, M_2, t_1, h > \otimes \\ < p_3, \varnothing, 1, M_3, t_1, h > \end{pmatrix} \rightarrow \begin{pmatrix} < p_1, a, 0, M_1 > \otimes \\ < p_2, d, 1, M_2 > \otimes \\ < p_3, \varnothing, 1, M_3 > \end{pmatrix}$

rl t_1 **:** $\begin{pmatrix} < p_1, a, 0, M_1, t_1, h > \otimes \\ < p_2, d, 1, M_2, t_1, h > \otimes \\ < p_3, \varnothing, 1, M_3, t_1, h > \otimes \\ [H = h_1] \end{pmatrix} \rightarrow \begin{pmatrix} < p_1, \varnothing, 1, ? > \otimes \\ < p_2, c \oplus c, 0, ? > \otimes \\ < p_3, a, 0, ? > \otimes \\ [H = h_1] \end{pmatrix}$ **if** $\begin{pmatrix} (0 \le h_1 - h) \wedge (h_1 - h \le 3) \ \wedge \\ (a \oplus a \subseteq M_1) \end{pmatrix} \rightarrow true$

rl $sens.t_2 :< p_3, \varnothing, 1, M_3 > \otimes [H = h] \rightarrow < p_3, \varnothing, 1, M_3, t_2, h > \otimes [H = h]$

rl $desens.t_2 :< p_3, \varnothing, 1, M_3, t_2, h > \rightarrow < p_3, \varnothing, 1, M_3 >$

rl $t_2 :< p_3, \varnothing, 1, M_3, t_2, h > \otimes [H = h_1] \rightarrow < p_3, b, 0, ? > \otimes [H = h_1]$ **if** $(2 \le h_1 - h) \wedge (h_1 - h \le 4) \rightarrow true$

rl $sens.t_3 :< p_3, M_3, 0, M_3 > \otimes [H = h] \rightarrow < p_3, M_3, 0, M_3, t_3, h > \otimes [H = h]$ **if** $(not(b \subseteq M_3)) \rightarrow true$

rl $desens.t_3 :< p_3, M_3, 0, M_3, t_3, h > \rightarrow < p_3, M_3, 0, M_3 >$

rl $t_3 :< p_3, M_3, 0, M_3, t_3, h > \otimes [H = h_1] \rightarrow < p_3, \varnothing, |M_3|, ? > \otimes [H = h_1]$

if $(0 \le h_1 - h) \wedge (h_1 - h \le \infty) \wedge (not(b \subseteq M_3)) \rightarrow true$

3.2 Déduction dans $\mathcal{T}_{temp}$

a- Règles de déduction

Les règles de déduction de $\mathcal{T}_{temp}$ sont les suivantes:

(i) Réflexivité: $\forall [s] : state$ $\overline{s : [s] \rightarrow [s]}$

(ii) Congruence:

$$\frac{\forall [s_1], ..., [s_n] : state \quad \forall f \in Sig_{temp_n} \qquad t_1 : [s_1] \rightarrow [s_1'] \ ... \ t_n : [s_n] \rightarrow [s_n']}{f(t_1, ..., t_n) : f([s_1], .., [s_n]) \rightarrow f([s_1'], .., [s_n'])}$$

où Sig_{temp_n} est l'ensemble des fonctions de Sig_{temp} d'arité *n*.

(iii) Substitution:

$$\forall\ (r:\ [s(x_1,...,x_n)]\rightarrow[s'(x_1,...,x_n)]\ \textit{if}\ \wedge_{i=1..k}[c_i(x_1,...,x_n)]\rightarrow\left[c_i'(x_1,...,x_n)\right])\in\mathcal{R}_{4temp}$$

$$t_1:\left[w_1\right]\rightarrow\left[w_1'\right],...,\ t_n:\left[w_n\right]\rightarrow\left[w_n'\right]$$

$$\frac{\left[c_1(\overline{w}/\overline{x})\right]\rightarrow\left[c_1'(\overline{w}/\overline{x})\right],...,\left[c_k(\overline{w}/\overline{x})\right]\rightarrow\left[c_k'(\overline{w}/\overline{x})\right]}{r(t_1,..,t_n):\ \left[s(\overline{w}/\overline{x}\right]\rightarrow\left[s'(\overline{w'}/\overline{x})\right]}$$

où $\overline{w}/\overline{x}$ est une substitution des x_i *par* les w_i $\quad 1\le i\le n$

(iv) Transitivité: $\quad \forall[s_1],[s_2],[s_3]:\textit{state}\ \ \dfrac{t_1:\ [s_1]\rightarrow[s_2]\quad t_2:\ [s_2]\rightarrow[s_3]}{t_1;t_2:\ [s_1]\rightarrow[s_3]}$

(v) Décomposition $\forall[s]:\textit{state}\ \ \forall p\in P\quad\forall M\ m\ m_i\in mT_{\Sigma_{\sigma(p)}}(\varnothing)\ \ \forall c\ c_i\in nat_\infty\quad\forall n\ k\in nat$

$$\mu:\ [m_1\oplus m_2\oplus...\oplus m_k]\ \rightarrow[m]$$

$$c:\ [c_1+c_2+..+c_k]\rightarrow[c]$$

$$t:\ [s]\rightarrow\left[<p,m,c,M>\otimes[\![p|n]\!]\right]$$

$$\overline{d(\mu,c,t):\ [s]\rightarrow\left[<p,m_1,c_1,M>\otimes<p,m_2,c_2,M>\otimes....\otimes<p,m_k,c_k,M>\otimes[\![p|n+(k-1)]\!]\right]}$$

(vi) Composition $\forall[s]:\textit{state}\ \ \forall p\in P\quad\forall M\ m_i\in mT_{\Sigma_{\sigma(p)}}(\varnothing)\quad\forall c_i\in nat_\infty\quad\forall n\ k\in nat$

$$\frac{t:[s]\rightarrow\left[<p,m_1,c_1,M>\otimes<p,m_2,c_2,M>\otimes....\otimes<p,m_k,c_k,M>\otimes[\![p|n]\!]\right]}{c(t):\ [s]\rightarrow\left[<p,m_1\oplus m_2\oplus...\oplus m_k,\ c_1+c_2+..+c_k,\ M>\otimes[\![p|n-(k-1)]\!]\right]}$$

(vii) Ajout-capacité : $\quad \forall[s]:\textit{state}\ \ \forall p\in P\ \ \forall m\in mT_{\Sigma_{\sigma(p)}}(\varnothing)\ \ \dfrac{t:[s]\rightarrow[<p,m>]}{ac(t):[s]\rightarrow[<p,m,\infty>]}$

(viii) Suppression-capacité : $\quad \forall[s]:\textit{state}\ \ \forall p\in P\ \ \forall m\in mT_{\Sigma_{\sigma(p)}}(\varnothing)\ \ \dfrac{t:[s]\rightarrow[<p,m,\infty>]}{sc(t):[s]\rightarrow[<p,m>]}$

(ix) Ajout-marquage global :

$$\forall[s]:\textit{state}\ \ \forall p\in P\ \forall M\in mT_{\Sigma_{\sigma(p)}}\ \ \forall c\in nat_\infty\ \frac{t:[s]\rightarrow[<p,M,c>]}{am(t):[s]\rightarrow\left[<p,M,c,M>\otimes[\![p|1]\!]\right]}$$

(x) Suppression-marquage global :

$$\forall[s]:\textit{state}\ \ \forall p\in P\ \ \forall M\ M'\in mT_{\Sigma_{\sigma(p)}}\ \ \forall c\in nat_\infty\ \ \frac{t:[s]\rightarrow\left[<p,M,c,M'>\otimes[\![p|1]\!]\right]}{sm(t):[s]\rightarrow[<p,M,c>]}$$

(xi) Incémenter-horloge

$$\forall[s]:\textit{state}\ \ \forall h\ \theta{:}nat\ \ \frac{t:[s]\ \rightarrow\ [<H=h>]}{ih(t):\ [s]\ \rightarrow[<H=h+\theta>]}$$

(xii) Copier-horloge

$$\forall [s]: state \ \ \forall h{:}nat \ \ \frac{t:[s] \ \rightarrow \ [< H = h >]}{ch_0(t):\ [s] \ \rightarrow [[H = h] \otimes [\![H \,|\, 1]\!]]}$$

$$\forall [s]: state \ \ \forall h{:}nat \ \ \frac{t:[s] \ \rightarrow \ [[H = h] \otimes [\![H \,|\, n]\!]]}{ch(t):\ [s] \ \rightarrow [[H = h] \otimes [H = h] \otimes [\![H \,|\, n+1]\!]]}$$

(xiii) Fusionner-copies-horloge

$$\forall [s]: state \ \ \forall h{:}nat \ \ \frac{t:\ [s] \ \rightarrow [[H = h] \otimes [\![H \,|\, 1]\!]]}{fch_0(t):[s] \ \rightarrow \ [< H = h >]}$$

$$\forall [s]: state \ \forall h{:}nat \ \ \frac{t:\ [s] \ \rightarrow [[H = h] \otimes [H = h] \otimes [\![H \,|\, n]\!]]}{fch(t):[s] \ \rightarrow \ [[H = h] \otimes [\![H \,|\, n-1]\!]]}$$

b- Exemple de déduction

Soit $s_0 =< p_1, a \oplus a, \infty > \otimes < p_2, d, 2 > \otimes < p_3, \varnothing, 4 > \otimes < H = 0 >$ l'état initial de l'ECATNet temporel présenté par la Figure 8.13. La preuve par déduction ci-dessous démontre l'accessibilité de l'état $s_1 =< p_1, a, \infty > \otimes < p_2, c \oplus c, 1 > \otimes < p_3, a \oplus b, 2 > \otimes < H = 2 >$ à partir de s_0.

$(1)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow [< p_1, a \oplus a, \infty > \otimes < p_2, d, 2 > \otimes < p_3, \varnothing, 4 > \otimes < H = 0 >]$ *règle de déduction Réfléxivité*

$$(2)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a \oplus a, \infty, a \oplus a > \otimes [\![p_1 \,|\, 1]\!] \otimes \\ < p_2, d, 2, d > \otimes [\![p_2 \,|\, 1]\!] \otimes \\ < p_3, \varnothing, 4, \varnothing > \otimes [\![p_3 \,|\, 1]\!] \otimes \\ < H = 0 > \end{bmatrix}$$

règle de déduction Ajout - marquage global appliquée 3 fois sur (1)

$$(3)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \,|\, 2]\!] \otimes \\ < p_2, d, 2, d > \otimes [\![p_2 \,|\, 1]\!] \otimes \\ < p_3, \varnothing, 4, \varnothing > \otimes [\![p_3 \,|\, 1]\!] \otimes \\ < H = 0 > \end{bmatrix}$$

Règle de déduction Décomposition de $< p_1, a \oplus a, \infty, a \oplus a >$ *sur* (2)

$$(4)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \,|\, 2]\!] \otimes \\ < p_2, d, 1, d > \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \,|\, 2]\!] \otimes \\ < p_3, \varnothing, 4, \varnothing > \otimes [\![p_3 \,|\, 1]\!] \otimes \\ < H = 0 > \end{bmatrix}$$

Règle de déduction Décomposition de $< p_2, d, 2, d >$ *sur* (3)

$$(5)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, d, 1, d > \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, \varnothing, 1, \varnothing > \otimes < p_3, \varnothing, 1, \varnothing > \otimes \\ < p_3, \varnothing, 0, \varnothing > \otimes < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 \mid 4]\!] \otimes \\ < H = 0 > \end{bmatrix}$$

Règle de déduction Décomposition de $< p_3, \varnothing, 4, \varnothing >$ *sur* (4)

$$(6)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, d, 1, d > \otimes < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, \varnothing, 1, \varnothing > \otimes < p_3, \varnothing, 1, \varnothing > \otimes \\ < p_3, \varnothing, 0, \varnothing > \otimes < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 \mid 4]\!] \otimes \\ [H = 0] \otimes [H = 0] \otimes [H = 0] \otimes [\![H \mid 3]\!] \end{bmatrix}$$

Règle de déduction Copier - horloge sur (5)

$$(7)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a > \otimes < p_2, d, 1, d > \otimes < p_3, \varnothing, 1, \varnothing > \otimes [H = 0] \otimes \\ < p_3, \varnothing, 1, \varnothing > \otimes [H = 0] \otimes \\ < p_3, \varnothing, 0, \varnothing > \otimes [H = 0] \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 \mid 4]\!] \otimes \\ [\![H \mid 3]\!] \end{bmatrix}$$

Commutativité de $\otimes$ *sur* (6)

$$(8)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes [H = 0] \otimes \\ < p_3, \varnothing, 1, \varnothing, t_2, 0 > \otimes [H = 0] \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes [H = 0] \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 \mid 4]\!] \otimes \\ [\![H \mid 3]\!] \end{bmatrix}$$

Règles de réécriture sens.t_1 *sens.*t_2 *sens.*t_3 *en parallèle sur* (7)

$$(9)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes \\ < p_3, \varnothing, 1, \varnothing, t_2, 0 > \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 \mid 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 \mid 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 \mid 4]\!] \otimes \\ [H = 0] \otimes [H = 0] \otimes [H = 0] \otimes [\![H \mid 3]\!] \end{bmatrix}$$

Commutativité de $\otimes$ *sur* (8)

$$(10)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes \\ < p_3, \varnothing, 1, \varnothing, t_2, 0 > \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ < H = 0 > \end{bmatrix}$$

Règle de déduction *Fusionner-copies-horloge* *sur* (9)

$$(11)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes \\ < p_3, \varnothing, 1, \varnothing, t_2, 0 > \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ < H = 2 > \end{bmatrix}$$

Règle de déduction *Incrémenter-horloge* *sur* (10)

$$(12)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes \\ < p_3, \varnothing, 1, \varnothing, t_2, 0 > \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ [H = 2] \otimes [H = 2] \otimes [\![H | 2]\!] \end{bmatrix}$$

Règle de déduction *Copier-horloge* *sur* (11)

$$(13)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, 0, a \oplus a, t_1, 0 > \otimes < p_2, d, 1, d, t_1, 0 > \otimes < p_3, \varnothing, 1, \varnothing, t_1, 0 > \otimes [H = 2] \otimes \\ < p_3, \varnothing, 1, \varnothing, t_2, 0 > \otimes [H = 2] \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ [\![H | 2]\!] \end{bmatrix}$$

Commutativité de $\otimes$ *sur* (12)

$$(14)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_2, c \oplus c, 0, ? > \otimes < p_3, a, 0, ? > \otimes [H = 2] \otimes \\ < p_3, b, 0, ? > \otimes [H = 2] \otimes \\ < p_3, \varnothing, 0, \varnothing, t_3, 0 > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ [\![H | 2]\!] \end{bmatrix}$$

règle de réécriture $t_1\ t_2$ *en parallèle sur* (13)

$$(15)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_2, c \oplus c, 0, ? > \otimes < p_3, a, 0, ? > \otimes [H = 2] \otimes \\ < p_3, b, 0, ? > \otimes [H = 2] \otimes \\ < p_3, \varnothing, 0, \varnothing > \otimes \\ < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ [\![H | 2]\!] \end{bmatrix}$$

règle de réécriture desens.t_3 *sur* (14)

$$(16)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_1, a, \infty, a \oplus a > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, d > \otimes < p_2, c \oplus c, 0, ? > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 0, \varnothing > \otimes < p_3, a, 0, ? > \otimes < p_3, b, 0, ? > \otimes < p_3, \varnothing, 2, \varnothing > \otimes [\![p_3 | 4]\!] \otimes \\ [H = 2] \otimes [H = 2] \otimes [\![H | 2]\!] \end{bmatrix}$$

commutativité de $\otimes$ *sur* (15)

$$(17)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, \varnothing, 1, ? > \otimes < p_1, a, \infty, ? > \otimes [\![p_1 | 2]\!] \otimes \\ < p_2, \varnothing, 1, ? > \otimes < p_2, c \oplus c, 0, ? > \otimes [\![p_2 | 2]\!] \otimes \\ < p_3, \varnothing, 0, ? > \otimes < p_3, a, 0, ? > \otimes < p_3, b, 0, ? > \otimes < p_3, \varnothing, 2, ? > \otimes [\![p_3 | 4]\!] \otimes \\ [H = 2] \otimes [H = 2] \otimes [\![H | 2]\!] \end{bmatrix}$$

équations de ? *sur* (16)

$$(18)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, \infty, ? > \otimes [\![p_1 | 1]\!] \otimes \\ < p_2, c \oplus c, 1, ? > \otimes [\![p_2 | 1]\!] \otimes \\ < p_3, a \oplus b, 2, ? > \otimes [\![p_3 | 1]\!] \otimes \\ [H = 2] \otimes [H = 2] \otimes [\![H | 2]\!] \end{bmatrix}$$

règle de déduction Composition sur (17)

$$(19)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, \infty > \otimes \\ < p_2, c \oplus c, 1 > \otimes \\ < p_3, a \oplus b, 2 > \otimes \\ [H = 2] \otimes [H = 2] \otimes [\![H | 2]\!] \end{bmatrix}$$

règle de déduction Suppression - marquage global sur (18)

$$(20)\ \mathcal{T}_{temp} \vdash [s_0] \rightarrow \begin{bmatrix} < p_1, a, \infty > \otimes \\ < p_2, c \oplus c, 1 > \otimes \\ < p_3, a \oplus b, 2 > \otimes \\ < H = 2 > \end{bmatrix}$$

règle de déduction Fusionner-copies-horloge sur (19)

Théorème8.3 :

Si $\mathcal{E}$ un ECATNet contextuel sans contraintes temporelles alors :

$$\forall [s][s'] \in State_{\mathcal{T}_{cont}}(\mathcal{E}) \quad on\ a \quad \mathcal{T}_{cont} \vdash [s] \rightarrow [s'] \quad \Rightarrow \quad \mathcal{T}_{temp} \vdash [s] \rightarrow [s']$$

où est $State_{\mathcal{T}_{cont}}(\mathcal{E})$ l'ensemble des états de $\mathcal{E}$ dans la théorie $\mathcal{T}_{cont}$.

Preuve : Annexe-1 preuve8-3.

VII- Modèle d'un ECATNet

Le modèle d'un ECATNet peut être intuitivement vu comme un système ayant un certain nombre d'états et pouvant changer son état en exécutant une ou plusieurs transitions de façon concurrente ou non. De tels modèles ont une structure de *catégorie* où les objets correspondent aux états de l'ECATNet et les morphismes aux différentes séquences de transitions dans cet ECATNet.

Ainsi, à chaque ECATNet $\mathcal{E}$ peut être associée une catégorie $\mathcal{C}at_{\mathcal{E}}$. Cette catégorie définit et engendre la classe des modèles de $\mathcal{E}$.

1- Catégorie $\mathcal{C}at_{\mathcal{E}}$ d'un ECATNet

La catégorie d'un ECATNet $\mathcal{E}$ est la catégorie induite par la théorie de réécriture $\mathcal{T} = (Sig, \varepsilon, \mathcal{L}, \mathcal{R})$ qui définit la sémantique de $\mathcal{E}$.

Soit $\mathcal{E}$ un ECATNet et $\mathcal{T} = (Sig, \varepsilon, \mathcal{L}, \mathcal{R})$ sa théorie de réécriture, $\mathcal{C}at_{\mathcal{E}}$ est la catégorie telle que:

a) l'ensemble des objets de $\mathcal{C}at_{\mathcal{E}}$ est l'ensemble des classes d'équivalence de termes $[t] \in T_{Sig,\varepsilon}(X)$.
b) Les morphismes de $\mathcal{C}at_{\mathcal{E}}$ sont les classes d'équivalence de "termes preuves". Ces morphismes sont construits suivant la méthode décrite au paragraphe VII-2 ci -dessous.
c) La composition de morphismes correspond à la composition séquentielle de transitions.

2- Morphismes de la Catégorie $\mathcal{C}at_{\mathcal{E}}$

Chaque morphisme défini dans $\mathcal{C}at_{\mathcal{E}}$ décrit une séquence de transition possible dans le système spécifié. Les morphismes de la catégorie $\mathcal{C}at_{\mathcal{E}}$ sont générés par les règles de déduction de la théorie de réécriture correspondant à $\mathcal{E}$. Cette génération consiste à appliquer les étapes de 1) à 4) suivantes:

Notation:

$\overline{\alpha:[t_1]\to[t_2]}$ représente la génération d'un morphisme α dont le domaine est l'objet classe d'équivalence du terme $[t_1]$ et le codomaine est l'objet classe d'équivalence du terme $[t_1]$

1) *Identité:* $\forall [t]\in T_{Sig,\varepsilon}(X)\quad \overline{id^{[t]}:[t]\to[t]}$

2) *Sig -structure:*

$\forall f \in Sig_n$ où Sig_n est l'ensemble des fonctions de Sig d'arité n.

$$\frac{\alpha_1:[t_1]\to[t_1']\ \ldots\ \alpha_n:[t_n]\to[t_n']}{f(\alpha_1,\ldots,\alpha_n):\ [f(t_1,..,t_n)]\to[f(t_1',..,t_n')]}$$

3) Remplacement:

$$\forall\ (\ r:\ [t(x_1,\ldots,x_n)]\to[t'(x_1,\ldots,x_n)]\ \text{if}\ \wedge_{i=1..k}[c_i(x_1,\ldots,x_n)]\to[c_i'(x_1,\ldots,x_n)]\ )\in\mathcal{R}$$

$$\frac{\alpha_1:[w_1]\to[w_1'],\ldots,\ \alpha_n:[w_n]\to[w_n']}{\beta_1:[c_1(\overline{w}/\overline{x})]\to[c_1'(\overline{w}/\overline{x})],\ldots,\beta_k:[c_k(\overline{w}/\overline{x})]\to[c_k'(\overline{w}/\overline{x})]}$$

$$r(\alpha,\beta):\ [t(\overline{w}/\overline{x}]\to[t'(\overline{w'}/\overline{x})]$$

où $\overline{w}/\overline{x}$ est une substitution des x_i par les w_i $\quad 1\le i\le n$

4) Transitivité:

$$\frac{\alpha:[t_1]\to[t_2]\quad \beta:[t_2]\to[t_3]}{\alpha;\beta:[t_1]\to[t_3]}$$

- Les règles d'identité et de remplacement génèrent les morphismes élémentaires. La première règle définit les morphismes d'identité sur les objets (élément de $T_{Sig,\varepsilon}(X)$) de la catégorie. La règle de remplacement génère tous les morphismes résultant de la substitution des variables de chaque règle *r* dans $\mathcal{R}$ par des valeurs de $T_{Sig,\varepsilon}(X)$.

- La règle *Sig-structure* décrit le fait que la sémantique d'une preuve composée est la composition des interprétations des preuves qui la constituent. Cette règle associe ainsi une structure algébrique à l'ensemble des morphismes.

- La dernière règle indique que la composition séquentielle de deux morphismes consécutifs dans $Cat_{\mathcal{E}}$ est un morphisme dans $Cat_{\mathcal{E}}$.

VIII- Classe des modèles d'un ECATNet

La catégorie $Cat_{\mathcal{E}}$ est seulement une parmi plusieurs modèles qui peuvent être associés à un ECATNet dont la logique de réécriture est $\mathcal{T}=(Sig,\ \varepsilon,\ \mathcal{L},\ \mathcal{R})$. La notion générale d'un modèle d'ECATNet, nommé $\mathcal{T}$*-système* est défini comme suit.

Définition "$\mathcal{T}$ *-système* "

Etant donné un ECATNet $\mathcal{E}$ dont la théorie de réécriture est $\mathcal{T} = (Sig, \varepsilon, \mathcal{L}, \mathcal{R})$, un $\mathcal{T}$ *-système* S est une catégorie S avec:

- Une (Sig, ε)-structure algébrique donnée par la famille de foncteurs $\{ f_S : S^n \to S \ / \ f \in Sig_n \ n \in \mathbb{N} \}$ vérifiant les équations $\mathcal{E}$. Ceci signifie que pour chaque équation $t(x_1, x_2, ..., x_n) = t'(x_1, x_2, ..., x_n)$ dans $\mathcal{E}$, les foncteurs t_S et t'_S sont identiques $t_S = t'_S$, où le foncteur t_S est défini inductivement à partir des foncteurs f_S de manière évidente.
- Pour chaque règle de réécriture $r : [t(x)] \to [t(x)]$ *if* C dans $\mathcal{R}$ il existe une transformation naturelle $r_S : t_S \Rightarrow t'_S$.

1- Catégorie $\mathcal{M}od_{\mathcal{E}}$ de la classe des modèles d'un ECATNet

La classe des modèles d'un ECATNet $\mathcal{E}$ (dont la théorie de réécriture est $\mathcal{T} = (Sig, \varepsilon, \mathcal{L}, \mathcal{R})$), est un ensemble de systèmes définissant une catégorie $\mathcal{M}od_{\mathcal{E}}$ où:

- Les objets sont les $\mathcal{T}$ *-systèmes* modèles de $\mathcal{E}$.
- Les morphismes de cette catégorie sont définis dans le paragraphe VIII-2.

2- Morphismes de la Catégorie $\mathcal{M}od_{\mathcal{E}}$

Soient S et S' deux $\mathcal{T}$ *-systèmes* . Un homomorphisme (qui est un foncteur) $F : S \to S'$ de $\mathcal{M}od_{\mathcal{E}}$ est un homorphisme des Sig-algèbres tel que:

1) F préserve les opérations définies dans Sig : $\forall f \in Sig_n \ n \in \mathbb{N} \quad f_S * F = F^n * f_{S'}$
où : * dénote la composition des transformations naturelles

Ceci signifie que l'application d'une opération f à des termes du système S suivie de la transformation F dans S' est équivalente à une transformation de chacun des termes par F suivie de l'opération f appliquée aux images des termes dans S').

2) F préserve les règles définies dans $\mathcal{R}$
$\forall (r : [t(x)] \to [t'(x)]) \in \mathcal{R}$, nous avons l'identité des transformations naturelles
$r_S * F = F^n * r_{S'}$ (n est le nombre de variables de la règle r).

(Cette condition indique que la réécriture avec r dans S suivi de la transformation F dans S' est équivalente à la transformation des termes par F dans S' suivi de la réécriture avec r dans S').

Ainsi, le foncteur F préserve toutes les opérations et les règles de S.

IX- Conclusion

Dans ce chapitre nous avons présenté la sémantique formelle des ECATNets. Nous avons associé à chaque type d'ECATNet une théorie de réécriture.

Dans le cas de la sémantique des **ECATNets simples,** l'apport principal réside dans la définition d'une *forme générale unique pour toutes les règles de réécriture*. En effet, dans les premiers travaux sur les ECATNets, la forme de la règle de réécriture associée à une transition dépend de la relation entre *IC*, *DT* et *CT*. De plus dans certains cas, on associe à une transition deux de règles de réécriture. La définition ici, d'une forme générale unique et homogène des règles de réécriture favorise une meilleure compréhension, une plus grande souplesse dans les processus de preuve, et un prototypage plus simple.

Dans la théorie $\mathcal{T}_{cap}$ des **ECATNets à capacité**, le test de non dépassement des capacités des places est traité par une *définition judicieuse de la notion d'état* d'un ECATNet. Ce qui évite l'ajout de places supplémentaires au réseau comme dans le cas de [BRA-82].

En ce qui concerne, les **ECATNets contextuels**, nous mettons en relief deux points importants. Le premier point concerne les contraintes *contextuelles positives* : le test de présence d'un jeton particulier *x* dans une place ne revient plus (comme dans les premiers travaux sur les ECATNets) à le *consommer puis* à *le recréer* dans cette place (interdisant ainsi certains tests parallèles de contraintes contextuelles), et la sémantique présentée ici reflète plus fidèlement l'aspect d'accès *concurrent* à un même jeton pour des tests contextuels.
Le second point concerne les contraintes *contextuelles négatives :* en général, la sémantique des réseaux de Petri à contraintes contextuelles négatives (ou à arcs inhibiteurs) est souvent présentée dans d'autres cadres sémantiques que la logique de réécriture, et l'approche proposée dans ce chapitre constitue à notre connaissance l'une des premières tentatives de spécification des tels types de réseaux dans le cadre la logique de réécriture.

La théorie $\mathcal{T}_{temp}$ des **ECATNets temporels** définie dans ce chapitre, montre comment les contraintes temporelles peuvent être prises en compte par une définition adéquate de l'état d'un ECATNet temporel (horloge, jetons disponible/indisponible, ..etc), et par une spécification appropriée des transitions entre ces états (sensibilisation, désensibilisation, franchissement).

Les différentes théories présentées ne sont pas indépendantes les unes des autres. En effet, la théorie $\mathcal{T}_{temp}$ est la plus générale. Elle inclus la théorie $\mathcal{T}_{cont}$ qui est plus générale que T_{cap}, qui inclus à son tour la théorie $\mathcal{T}_{simp}$.

Cette présentation progressive de la sémantique permet de marquer les différentes étapes d'évolution des ECATNets, et de faciliter la définition éventuelle de nouvelles extensions des ECATNets .

Nous avons précisé la classe des modèles d'un ECATNet. Celle-ci servira principalement à la spécification de la sémantique du langage CIRTA qui est définie au niveau classes des modèles.

Chapitre 9

Sémantique formelle du langage CIRTA

Sommaire

I- Introduction

Le présent chapitre est consacré à la présentation de la sémantique formelle du langage CIRTA. Nous associons à chaque concept introduit (module, enrichissement, paramétrisation, instanciation, renommage,…etc.) dans la définition du langage, un objet mathématique qui définit sa sémantique. Rappelons que la sémantique du langage est définie au niveau modèle et qu'il est par conséquent nécessaire de préciser la classe des modèles de chaque spécification écrite dans le langage CIRTA.

Nous entamons ce chapitre par la définition de la sémantique statique du langage. Nous précisons ensuite l'institution qui sera utilisée pour permettre de définir de façon simple et uniforme le *système logique* servant de cadre à la définition de la sémantique du langage CIRTA. Nous définissons la sémantique des différents types de modules de base définis dans CIRTA. Enfin, nous associons à chaque primitive un concept mathématique qui décrit sa sémantique et permet de déduire la sémantique des modules structurés.

II- Sémantique statique du langage CIRTA

Une première étape de la définition de la sémantique d'un langage consiste à présenter les aspects relatifs à la sémantique statique du langage. Ces aspects précisent et explicitent les restrictions et les contraintes sémantiques du langage. Certaines de ces contraintes peuvent être détecter grâce à la syntaxe du langage; d'autres par contre nécessitent l'élaboration d'outils spécifiques sur le plan théorique ou pratique. Nous décrire ces contraintes, nous utilisons dans ce chapitre les notations suivantes :

Notations:

$requis(\widehat{\mathcal{E}})$: l'application qui pour chaque module $\widehat{\mathcal{E}}$ associe la liste des modules requis (directement ou indirectement) par $\widehat{\mathcal{E}}$.

$Type(\widehat{\mathcal{E}})$: l'application qui pour chaque module $\widehat{\mathcal{E}}$ associe son type (ordinaire, formel, générique)

$PF(\widehat{\mathcal{E}})$: la fonction qui à chaque module donne le liste des paramètres formels de $\widehat{\mathcal{E}}$.

$occur(x, L)$: la fonction qui détermine le nombre d'occurrences de l'élément *x* dans la liste *L*

$visible(x)$: prédicat qui est *vrai* si l'élément *x* est visible, et *faux* sinon.

$S(x)$: sorte de l'élément *x*.

$S(Spec(x))$: l'ensemble global des sortes définies dans la spécification algébrique du module *x*.

$Op(Spec(x))$: l'ensemble global des opérations définies dans la spécification algébrique du module *x*.

$Eqs(Spec(x))$: l'ensemble global des équations définies dans la spécification algébrique du module *x*.

$P(R(x))$ (resp. $T(R(x))$): l'ensemble des places (resp. transitions) du réseau du module x

$\overleftrightarrow{x}$: le module *plat* associé au module x.

$Itf(x)$: interface du module x.

1- Contraintes sur l'entête d'un module

- Le nom associé à un module de spécification est *unique*: dans une spécification deux modules ne peuvent avoir le même identificateur même s'ils peuvent se distinguer par leur type (*ordinaire, formel, générique*) lors de leur définition. Toute référence (enrichissement ou renommage ou composition,….etc.) à un module de nom *N* doit désigner de manière *univoque* un seul module.

- Les types des modules $\widehat{\mathcal{E}}_i$ requis (enrichis, composés, renommés,…) par un module $\widehat{\mathcal{E}}$ doivent être *compatibles*. On doit avoir

$$\forall \widehat{\mathcal{E}}_i \quad \left(\widehat{\mathcal{E}}_i \in requis\left(\widehat{\mathcal{E}}\right) \Rightarrow type\left(\widehat{\mathcal{E}}\right) \leq type\left(\widehat{\mathcal{E}}_i\right)\right)$$

  où la relation $\leq$ est un ordre partiel sur les types de modules défini par :

$$\begin{cases} \text{pour tout type } t \text{ on a : } t \leq t \\ \text{générique} \leq \text{ordinaire} \\ \text{formel} \leq \text{ordinaire} \end{cases}$$

  Ainsi un module de type ordinaire par exemple, ne peut pas enrichir un module générique alors que l'inverse est possible.

- Si un module générique $\widehat{\mathcal{E}}$ (ayant comme liste des paramètres formels la liste $PF(\widehat{\mathcal{E}})$) requiert d'autres modules génériques $\widehat{\mathcal{E}}_i$ $(i = 1..n)$ ayant chacun comme ensemble de paramètres formels l'ensemble $PF(\widehat{\mathcal{E}}_i)$, alors la condition suivante doit être vérifiée :

$$\forall i \;\; i = 1..n \quad PF(\widehat{\mathcal{E}}_i) \subseteq PF(\widehat{\mathcal{E}})$$

  Ce qui peut être formulé par :

$$\forall \hat{\mathcal{E}} \left(Type(\hat{\mathcal{E}}) = générique \Rightarrow \forall \hat{\mathcal{E}}' \left(\hat{\mathcal{E}}' \in requis(\hat{\mathcal{E}}) \wedge Type(\hat{\mathcal{E}}') = générique \Rightarrow PF(\hat{\mathcal{E}}') \subseteq PF(\hat{\mathcal{E}})\right)\right)$$

- Si le type d'un module est "ordinaire" ou "formel" alors la liste des paramètres formels de ce module est *vide :*

$$\forall \hat{\mathcal{E}} \left(\left(Type(\hat{\mathcal{E}}) = ordinaire \vee Type(\hat{\mathcal{E}}) = formel\right) \Rightarrow PF(\hat{\mathcal{E}}) = \varnothing\right)$$

- Tout module de type générique doit avoir *au moins un paramètre formel:*

$$\forall \hat{\mathcal{E}} \left(Type(\hat{\mathcal{E}}) = générique \Rightarrow PF(\hat{\mathcal{E}}) \neq \varnothing\right)$$

- Dans la liste des paramètres formels d'un module générique, chaque paramètre formel apparaît une *seule fois*.

$$\forall \hat{\mathcal{E}} \left(Type(\hat{\mathcal{E}}) = générique \Rightarrow \forall \hat{\mathcal{E}}' \in PF(\hat{\mathcal{E}}) \;\; occur(\hat{\mathcal{E}}', PF(\hat{\mathcal{E}})) = 1\right)$$

- Les modules définis dans une liste des paramètres formels d'un module générique, doivent avoir le type "*formel*"

$$\forall \hat{\mathcal{E}} \left(Type(\hat{\mathcal{E}}) = générique \Rightarrow \forall \hat{\mathcal{E}}' \in PF(\hat{\mathcal{E}}) \;\; Type(\hat{\mathcal{E}}') = formel\right)$$

- Le graphe des modules associé à une spécification doit être *sans circuit* (au sens de la théorie des graphes) : aucun module $\hat{\mathcal{E}}$ ne doit faire référence à lui même de manière directe ou indirecte ($\hat{\mathcal{E}}$ ne requiert jamais $\hat{\mathcal{E}}$). $\forall \hat{\mathcal{E}}\ \hat{\mathcal{E}} \notin requis(\hat{\mathcal{E}})$

2- Contraintes sur la signature d'un module

- Comme pour le nom d'un module, le nom d'une sorte est *unique.*
- La relation de sous-sortes est une relation *d'ordre partiel strict transitive*. Elle doit être *acyclique.* Chaque déclaration de la forme : « **subsort** s<s' » est traitée comme la déclaration d'une opération de coersion « **op** _ : s→s' ».

$$\forall s\ s'\ s'' \in S(Spec(\hat{\mathcal{E}})) \quad (s < s' \Rightarrow \neg(s' < s)) \wedge ((s < s' \ \wedge\ s' < s'') \Rightarrow s < s'')$$

- La surcharge et le polymorphisme des opérations étant autorisés, l'identificateur d'une opération est (sémantiquement) le triplet de la forme (forme d'opération, domaine, co-domaine) et est *unique*.
  Une fois cette transformation effectuée, on n'a plus (proprement dit) d'opérations surchargées, ni de coersions ou d'opérations anonymes.
- Les sortes du domaine et co-domaine d'une opération sont *des sortes définies*.

$$\forall f : s_1\ s_2\ ..s_n \rightarrow s_{n+1} \in Op(Spec(\hat{\mathcal{E}})) \quad s_1\ s_2\ ..s_n\ s_{n+1} \in S(Spec(\hat{\mathcal{E}}))$$

- Le nombre d'arguments d'une l'opération (défini dans *'forme d'opération'*) est *égal* au nombre de sortes déclarées dans son domaine.

3- Contraintes sur les équations d'un module

- Le nom d'une variable déclarée dans une spécification est *unique* et doit être différents des constantes (opérations sans arguments) de cette spécification pour éviter certaines ambiguïtés.

$$\forall (v : s) \in Var(\hat{\mathcal{E}}) \quad occur\left((v : s), Var(\hat{\mathcal{E}})\right) = 1 \wedge \neg\left(\exists (v :\rightarrow s \in Op(Spec(\hat{\mathcal{E}}))\right)$$

- La sorte d'une variable doit être une sorte préalablement *définie et unique*.

$$\forall (v : s) \in Var(\hat{\mathcal{E}}) \quad s \in S(Spec(\hat{\mathcal{E}})) \wedge \neg\left(\exists (v : s') \in Var(\hat{\mathcal{E}}) \wedge s \neq s'\right)$$

- Pour toute équation inconditionnelle $t = t'$, les termes t *et* t' sont de *même sorte.*
- Pour toute équation conditionnelle $t = t'$ *if* c, les termes t *et* t' sont de *même sorte* et le terme c est de sorte $'bool'$.

Notons que l'analyse des équations nécessite des outils appropriés comme [VOI-86][ZEG-90] permettant de transformer les équations, pour traiter convenablement les ambiguïtés impliquées par la flexibilité offerte en ce qui concerne la forme des opérations.

4- Contraintes sur les places d'un réseau

- La sorte d'une place doit être une sorte *définie*.
- Le marquage initial d'une place *p* doit être compatible avec la sorte de *p* et ne doit pas dépasser sa capacité.
- Une même place peut avoir plusieurs occurrences dans le réseau (sémantiquement il s'agit de la même place).
- Les différentes occurrences d'une place *p* ont même sorte, même capacité, même marquage et même type (interne/ interface).

5- Contraintes sur les transitions d'un réseau

- Une même transition peut avoir plusieurs occurrences dans le réseau (sémantiquement il s'agit de la même transition).
- Les différentes occurrences d'une transition *t* sont de même type (interne/ interface) et ont même intervalle temporel statique.
- Les contextes de toutes les occurrences d'une transition *t* doivent être compatibles:
  $\forall t_i\ t_j$ deux occurrences de la transition *t* on a:

$$\forall v \in VAR \quad \left((v,s) \in ctx(t_i) \ \wedge \ (v,s') \in ctx(t_j)\right) \Rightarrow s = s'$$

- Pour toute transition *t*, l'intervalle temporel statique $IS(t) = [\mathbf{min}, \mathbf{max}]$ doit vérifier la condition : $(\mathbf{min} \in \mathbb{N} \ \wedge \ \mathbf{max} \in \mathbb{N} \ \wedge \ \mathbf{min} \leq \mathbf{max}) \vee (\mathbf{min} \in \mathbb{N} \ \wedge \ \mathbf{max} = +\infty)$
- La condition d'une transition *t* (i.e. *TC(t)*) est la conjonction des conditions de transitions de toutes les occurrences de *t :*

  Soient $t_1, t_2, ..., t_n$ *toutes les occurrences de la transition t,* $TC(t) = \wedge_{i=1..n} TC(t_i)$

6- Contraintes sur les pré/post conditions

- Pour toute place *p* de sorte *s* (i.e. $\sigma(p) = s$) , les sortes des termes de *IC(p,t), DT(p,t), CT(p,t)* doivent être compatibles avec la sorte *s.*

$\forall p \in P \ \forall t \in T$

$$\begin{cases} (p,t) \in dom(IC) \Rightarrow IC(p,t) = empty \vee \left(\left(IC(p,t) = \sim \alpha \vee IC(p,t) = \alpha\right) \wedge \left(\alpha \text{ est de sorte } m_{\sigma(p)}\right)\right) \\ (p,t) \in dom(DT) \Rightarrow DT(p,t) = \forall \ \vee \left(\left(DT(p,t) = \alpha\right) \wedge \left(\alpha \text{ est de sorte } m_{\sigma(p)}\right)\right) \\ (p,t) \in dom(CT) \Rightarrow CT(p,t) \text{ est de sorte } m_{\sigma(p)} \end{cases}$$

- L'ensemble des variables intervenant dans la définition de la précondition et la postcondition de chaque transition *t* doivent vérifier les deux propriétés suivantes :
  - $Var_{CT}(t) \subseteq Var_{IC}(t) \cup Var_{DT}(t)$
  - $Var_{TC}(t) \subseteq Var_{IC}(t) \cup Var_{DT}(t)$

où $\begin{cases} Var_{IC}(t) = \cup_{p \in P} Var\left(IC(p,t)\right) \\ Var_{DT}(t) = \cup_{p \in P} Var\left(DT(p,t)\right) \\ Var_{CT}(t) = \cup_{p \in P} Var\left(CT(p,t)\right) \end{cases}$

Notation :

Var(IC(p,t)), $Var\left(DT(p,t)\right), Var\left(CT(p,t)\right)$ *et* $Var_{TC}(t)$ sont les ensembles des variables intervenant respectivement dans la définition de *IC(p,t),* $DT(p,t), CT(p,t)$ *et* $TC(t)$.

7- Contraintes sur la structure du réseau d'un module

- Le réseau d'un module peut être *vide*.
- Le réseau peut être connexe ou non (au sens de la théorie des graphes).
- Un même nœud de substitution peut avoir plusieurs occurrences dans le réseau (sémantiquement il s'agit du même noeud).
- Les différentes occurrences d'un nœud de substitution N correspondent au même module.
- Les places et transitions adjacentes à un nœud de substitution doivent être de type *interface*.
- Un nœud de substitution ne doit pas être adjacent à un autre nœud de substitution.
- Le morphisme de substitution associé à un nœud de substitution doit être un morphisme correctement défini. Ceci signifie que pour tout nœud de substitution N associé à un module $\hat{\mathcal{E}}$, le morphisme de substitution $Sub_{N,\hat{\mathcal{E}}}$ doit vérifier les propriétés suivantes :

(i) $Sub_{N,\hat{\mathcal{E}}} : P(N) \cup T(N) \rightarrow PI(\hat{\mathcal{E}}) \cup TI(\hat{\mathcal{E}})$

où $P(N)$ et $T(N)$: sont respectivement l'ensemble des places et des transitions adjacentes à N.

$PI(\hat{\mathcal{E}})$ *et* $TI(\hat{\mathcal{E}})$: sont respectivement l'ensemble des places interfaces et des transitions interfaces du module $\hat{\mathcal{E}}$.

(ii) $\forall p \in P(N)$

$$\begin{cases} \left(Sub(p) \in PI(\hat{\mathcal{E}})\right) \wedge \\ \left(\sigma(p) = \sigma_{\hat{\mathcal{E}}}(Sub(p))\right) \wedge \\ \left(Cap(p) = Cap_{\hat{\mathcal{E}}}(Sub(p))\right) \end{cases}$$

(ii) $\forall t \in T(N)$

$$\begin{cases} \left(Sub(t) \in TI(\hat{\mathcal{E}})\right) \wedge \\ \left(IS(t) = IS_{\hat{\mathcal{E}}}(Sub(t))\right) \wedge \\ \forall\ (v,s) \in ctx(t)\ \ \left((v,s') \in ctx(Sub(t)) \quad \Rightarrow s = s'\right) \end{cases}$$

8- Contraintes de consistance hiérarchique de la structure d'un module

Les modules requis par un module CIRTA $\hat{\mathcal{E}}$ (qu'ils soient des modules enrichis, renommés, combinés, paramètres formels ou paramètres effectifs) doivent vérifier les contraintes syntaxiques et sémantiques du langage.

Il est nécessaire de vérifier que la structure d'un module est cohérente. Pour cela on définit la notion de consistance hiérarchique d'un module.

Un module d'ECATNet $\hat{\mathcal{E}}$ est un module hiérarchiquement consistant si et seulement si on est dans l'un des cas suivants:

- ***Cas1***: $\widehat{\mathcal{E}}$ est un module de base (de type 'formel' ou 'ordinaire')

- ***Cas2:*** $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}' + \Delta\mathcal{E}$ tel que :
  - $\widehat{\mathcal{E}}'$ est hiérarchiquement consistant
  - $\forall s \ \left(s \in S\left(Spec\left(\Delta\mathcal{E}\right)\right) \Rightarrow s \notin S\left(Spec\left(\overleftarrow{\mathcal{E}'}\right)\right)\right)$
  - $\forall f : s_1\ s_2\ \ldots s_n \rightarrow s_{n+1} \in Op\left(Spec\left(\Delta\mathcal{E}\right)\right)$ on a:

    $\left(f : s_1\ s_2\ \ldots s_n \rightarrow s_{n+1} \notin Op\left(Spec\left(\overleftrightarrow{\mathcal{E}'}\right)\right)\right) \wedge \left(\left\{s_1, s_2, \ldots, s_n, s_{n+1}\right\} \subseteq S\left(Spec\left(\Delta\mathcal{E}\right)\right) \cup S\left(Spec\left(\overleftrightarrow{\mathcal{E}'}\right)\right)\right)$
  - $\forall v \in Var\left(\Delta\mathcal{E}\right) \Rightarrow S\left(v\right) \in S\left(Spec\left(\Delta\mathcal{E}\right)\right) \cup S\left(Spec\left(\overleftarrow{\mathcal{E}'}\right)\right)$
  - $\forall p \in P\left(R\left(\Delta\mathcal{E}\right)\right)$ on a:
    - $S\left(p\right) \in S\left(Spec\left(\Delta\mathcal{E}\right)\right) \cup S\left(Spec\left(\overleftarrow{\mathcal{E}'}\right)\right)$
    - $p \in P\left(R\left(\overleftrightarrow{\mathcal{E}'}\right)\right) \Rightarrow \left[\begin{array}{l} S\left(p\right)_{|_{\Delta\mathcal{E}}} = S\left(p\right)_{|_{\overleftarrow{\mathcal{E}}}} \\ Cap\left(p\right)_{|_{\Delta\mathcal{E}}} = Cap\left(p\right)_{|_{\overleftarrow{\mathcal{E}}}} \\ Type\left(p\right)_{|_{\Delta\mathcal{E}}} = Type\left(p\right)_{|_{\overleftarrow{\mathcal{E}}}} = interface \\ M\left(p\right)_{|_{\Delta\mathcal{E}}} = M\left(p\right)_{|_{\overleftarrow{\mathcal{E}}}} \end{array}\right]$
  - $\forall t \in T\left(R\left(\Delta\mathcal{E}\right)\right)$ on a:

    $t \in T\left(R\left(\overleftrightarrow{\mathcal{E}'}\right)\right) \Rightarrow \left[\begin{array}{l} Type\left(t\right)_{|_{\Delta\mathcal{E}}} = Type\left(t\right)_{|_{\overleftarrow{\mathcal{E}}}} = interface \\ IS\left(t\right)_{|_{\Delta\mathcal{E}}} = IS\left(t\right)_{|_{\overleftarrow{\mathcal{E}}}} \\ \forall\left(v,s\right) \in ctx\left(t\right)_{|_{\Delta E}} \quad \left(v,s'\right) \in ctx\left(t\right)_{|_{\overleftarrow{\mathcal{E}}}} \Rightarrow s = s' \end{array}\right]$
  - $\forall v\ s \ \left(\left(v,s\right) \in ctx\left(t\right)_{|_{\Delta\mathcal{E}}}\right) \Rightarrow \left(s \in S\left(Spec\left(\Delta\mathcal{E}\right)\right) \cup S\left(Spec\left(\overleftarrow{\mathcal{E}'}\right)\right)\right)$
  - $\forall (p,t) \in \left(P\left(R\left(\Delta E\right)\right) \times T\left(R\left(\Delta E\right)\right)\right) \cap \left(P\left(R\left(\overleftrightarrow{\mathcal{E}'}\right)\right) \times T\left(R\left(\overleftrightarrow{\mathcal{E}'}\right)\right)\right)$ on :

    $\left[\begin{array}{l} IC\left(p,t\right)_{|_{\Delta\mathcal{E}}} = IC\left(p,t\right)_{|_{\overleftarrow{\mathcal{E}}}} \ si \ \left(p,t\right) \in dom\left(IC\right)_{|_{\Delta\mathcal{E}}} \cap dom\left(IC\right)_{|_{\overleftarrow{\mathcal{E}}}} \\ DT\left(p,t\right)_{|_{\Delta\mathcal{E}}} = DT\left(p,t\right)_{|_{\overleftarrow{\mathcal{E}}}} \ si \ \left(p,t\right) \in dom\left(DT\right)_{|_{\Delta\mathcal{E}}} \cap dom\left(DT\right)_{|_{\overleftarrow{\mathcal{E}}}} \\ CT\left(p,t\right)_{|_{\Delta\mathcal{E}}} = CT\left(p,t\right)_{|_{\overleftarrow{\mathcal{E}}}} \ si \ \left(p,t\right) \in dom\left(CT\right)_{|_{\Delta\mathcal{E}}} \cap dom\left(CT\right)_{|_{\overleftarrow{\mathcal{E}}}} \end{array}\right]$

- ***Cas3***: $\widehat{\mathcal{E}} = \cup_{i=1..n} \widehat{\mathcal{E}}_i$ tel que :
  - $\forall i = 1..n \quad \widehat{\mathcal{E}}_i$ est hiérarchiquement consistant.
  - Il existe au moins un ordre total "<" sur les modules $\widehat{\mathcal{E}}_1 < \widehat{\mathcal{E}}_2 < \widehat{\mathcal{E}}_3 < ... < \widehat{\mathcal{E}}_n$ tel que :
    - Les interfaces $Itf\left(\widehat{\mathcal{E}}_1\right)$ *et* $Itf\left(\widehat{\mathcal{E}}_2\right)$ sont compatibles :

$$\forall p \in Itf\left(R\left(\widehat{\mathcal{E}}_1\right)\right) \cap Itf\left(R\left(\widehat{\mathcal{E}}_2\right)\right) \Rightarrow \begin{bmatrix} S(p)_{|\overleftarrow{\mathcal{E}}_1} = S(p)_{|\overleftarrow{\mathcal{E}}_2} \\ Cap(p)_{|\overleftarrow{\mathcal{E}}_1} = Cap(p)_{|\overleftarrow{\mathcal{E}}_2} \\ Type(p)_{|\overleftarrow{\mathcal{E}}_1} = Type(p)_{|\overleftarrow{\mathcal{E}}_2} = interface \\ M(p)_{|\overleftarrow{\mathcal{E}}_1} = M(p)_{|\overleftarrow{\mathcal{E}}_2} \end{bmatrix}$$

$$\forall t \in Itf\left(R\left(\widehat{\mathcal{E}}_1\right)\right) \cap Itf\left(R\left(\widehat{\mathcal{E}}_2\right)\right) \Rightarrow \begin{bmatrix} Type(t)_{|\overleftarrow{\mathcal{E}}_1} = Type(t)_{|\overleftarrow{\mathcal{E}}_2} = interface \\ IS(t)_{|\overleftarrow{\mathcal{E}}_1} = IS(t)_{|\overleftarrow{\mathcal{E}}_2} \\ \forall (v,s) \in ctx(t)_{|\overleftarrow{\mathcal{E}}_1} \quad (v,s') \in ctx(t)_{|\overleftarrow{\mathcal{E}}_2} \Rightarrow s = s' \end{bmatrix}$$

    un arc commun a les mêmes inscriptions dans les deux modules

    $\forall (p,t) \in \left(Itf\left(R\left(\widehat{\mathcal{E}}_1\right)\right) \cap Itf\left(R\left(\widehat{\mathcal{E}}_2\right)\right)\right)^2$ on :

$$\begin{bmatrix} IC(p,t)_{|\overleftarrow{\mathcal{E}}_1} = IC(p,t)_{|\overleftarrow{\mathcal{E}}_2} \ si \ (p,t) \in dom(IC)_{|\overleftarrow{\mathcal{E}}_1} \cap dom(IC)_{|\overleftarrow{\mathcal{E}}_2} \\ DT(p,t)_{|\overleftarrow{\mathcal{E}}_1} = DT(p,t)_{|\overleftarrow{\mathcal{E}}_2} \ si \ (p,t) \in dom(DT)_{|\overleftarrow{\mathcal{E}}_1} \cap dom(DT)_{|\overleftarrow{\mathcal{E}}_2} \\ CT(p,t)_{|\overleftarrow{\mathcal{E}}_1} = CT(p,t)_{|\overleftarrow{\mathcal{E}}_2} \ si \ (p,t) \in dom(CT)_{|\overleftarrow{\mathcal{E}}_1} \cap dom(CT)_{|\overleftarrow{\mathcal{E}}_2} \end{bmatrix}$$

    - $\left(P\left(R\left(\overleftarrow{E}_1\right)\right) \cup T\left(R\left(\overleftarrow{E}_1\right)\right)\right) \cap \left(P\left(R\left(\overleftarrow{E}_2\right)\right) \cup T\left(R\left(\overleftarrow{E}_2\right)\right)\right) \neq \varnothing$.
    - le module $\overleftarrow{\mathcal{E}_1 \cup \mathcal{E}_2} \cup \widehat{\mathcal{E}}_3 \cup ... \cup \widehat{\mathcal{E}}_n$ qui compose *n-1* modules est hiérarchiquement consistant.

- ***Cas 4***: $\widehat{\mathcal{E}} = Rename(\widehat{\mathcal{E}}', Morph)$:

- $\widehat{\mathcal{E}}'$ est hiérarchiquement consistant.
- Les modules $\widehat{\mathcal{E}}$ et $\widehat{\mathcal{E}}'$ sont de même type (ordinaire, formel, générique) : $Type(\widehat{\mathcal{E}}) = Type(\widehat{\mathcal{E}}')$
- Si $\widehat{\mathcal{E}}$ et $\widehat{\mathcal{E}}'$ sont des modules génériques alors ils doivent avoir la même liste des paramètres formels : $\left(Type(\widehat{\mathcal{E}}) = générique \wedge Type(\widehat{\mathcal{E}}') = générique\right) \Rightarrow PF(\widehat{\mathcal{E}}) = PF(\widehat{\mathcal{E}}')$
- Chaque objet (sorte, opération, place ou transition) renommé par le morphisme de renommage *Morph* doit être un objet visible de la spécification globale $\overleftarrow{\mathcal{E}'}$.

$$\forall ob \quad \left(ob \in dom\ (Morph) \Rightarrow \left(ob \in \overleftarrow{\mathcal{E}'} \wedge visible(ob)\right)\right)$$

- Un objet *ob* ne peut être renommé par un identificateur *ob'* (avec $ob \neq ob'$) *préalablement défini* dans $\overleftarrow{\mathcal{E}'}$: $\forall ob \in \overleftarrow{\mathcal{E}'} \quad \left(Morph(ob) = ob' \ et \ ob \neq ob'\right) \Rightarrow ob' \notin \overleftarrow{\mathcal{E}'}$.

o ***Cas 5***: $\widehat{\mathcal{E}} = Hide(\widehat{\mathcal{E}}', set)$:

- $\widehat{\mathcal{E}}'$ est hiérarchiquement consistant.
- Les modules $\widehat{\mathcal{E}}$ et $\widehat{\mathcal{E}}'$ sont de même type (ordinaire ou générique).

$$\left(Type(\widehat{\mathcal{E}}) = Type(\widehat{\mathcal{E}}') = ordinaire\right) \vee \left(Type(\widehat{\mathcal{E}}) = Type(\widehat{\mathcal{E}}') = générique\right)$$

- Si $\widehat{\mathcal{E}}$ et $\widehat{\mathcal{E}}'$ sont des modules génériques alors ils doivent avoir la même liste des paramètres formels.

$$\left(Type(\widehat{\mathcal{E}}) = générique \wedge Type(\widehat{\mathcal{E}}') = générique\right) \Rightarrow PF(\widehat{\mathcal{E}}) = PF(\widehat{\mathcal{E}}')$$

- Chaque élément (sorte, opération, place ou transition) rendu invisible par le contrôle de visibilité *set* doit être un élément visible de la spécification globale $\overleftrightarrow{\mathcal{E}'}$.

$$\forall ob \quad \left(ob \in set \Rightarrow \left(ob \in \overleftrightarrow{\mathcal{E}'} \wedge visible(ob)\right)\right)$$

o ***Cas 6***: $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}'\left[\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ..., \widehat{\mathcal{E}}_n\right]$ tel que :

- $\forall i = 1..n \quad \widehat{\mathcal{E}}_i$ est hiérarchiquement consistant.
- $\widehat{\mathcal{E}}'' = \widehat{\mathcal{E}}' + \left(\cup_{i=1..n} \widehat{\mathcal{E}}_i\right)$ est hiérarchiquement consistant.

III- Définition d'une institution

Comme nous l'avons déjà mentionné dans le chapitre 4, un module de spécification contient des déclarations de sortes, d'opérations, d'axiomes, et d'un réseau. Ces quatre types d'éléments constituent le noyau du module qui contient également des directives spécifiant comment ce module doit être assemblé à d'autres modules pour former la spécification globale.

En pratique, il est possible de considérer, dans une spécification CIRTA, plusieurs types de signatures (ordonnée ou non), plusieurs types d'axiomes (des équations, des équations conditionnelles, avec ou sans égalité,...etc.), différents types de transitions (avec ou sans conditions, temporelles ou non,…etc.).
Ces différents choix peuvent être combinés de façon relativement arbitraire, ce qui résulte évidemment en une profusion de situations possibles. Le pouvoir d'expression du langage de spécification dépend de la combinaison retenue. Le besoin se fait sentir de disposer d'un cadre général permettant de décrire de façon uniforme la combinaison retenue. Par ailleurs, il devient évident qu'il doit être possible de changer de combinaison sans avoir pour autant à redéfinir complètement la sémantique du langage de spécification.

Le concept d'*institution* introduite dans [GB-84] est un outil adéquat permettant de définir de façon simple et uniforme le *système logique* servant de cadre à la définition du langage de spécification considéré. Ce concept couvre de façon adéquate tous les ingrédients clefs d'une logique. Ceci inclut la notion de *signature*, la notion de *déduction* des formules à partir d'un ensemble *d'axiomes* et de *règles de réécriture*, la notion de *modèle*, et la notion de *validation* d'une formule par un modèle.

La théorie des institutions permet d'exprimer la relation entre les théories et les modèles sans se limiter à un système logique particulier. Un ingrédient clef dans un système logique est celui de signature et la possibilité de traduire des formules et des modèles à travers des morphismes de signature.

Une institution consiste en une collection de signatures, avec pour chaque signature Σ, un ensemble de $\Sigma-$ *formules*, une collection de $\Sigma-$*modèles* et des relations de validation entre $\Sigma-$*modèles* et $\Sigma-$ *formules*. La seule condition de nature sémantique requise par une institution est que lorsqu'on modifie une signature par un morphisme de signature, les changements induits au niveau des formules et des modèles préservent les relations de validation. Plus précisément :

Définition :

Une institution est un quadruplet $Ins = (Sig_{Ins}, Sen_{Ins}, Mod_{Ins}, \models_{Ins})$ tel que :

- Sig_{Ins} est une catégorie de *signatures.*
- Sen_{Ins} est un foncteur $Sen_{Ins} : Sig_{Ins} \to Set$ (où *Set* désigne la catégorie de tous le ensembles).
  Sen_{Ins} associe à chaque signature Σ l'ensemble $Sen_{Ins}(\Sigma)$ des $\Sigma-$ *formules*, et à chaque morphisme de signature $\rho : \Sigma \to \Sigma'$ l'application $Sen_{Ins}(\rho) : Sen_{Ins}(\Sigma) \to Sen_{Ins}(\Sigma')$ qui traduit les $\Sigma-$ *formules* en $\Sigma'-$ *formules*. $(Sen_{Ins}(\rho))(\varphi)$ est souvent noté $\rho(\varphi)$.
- Mod_{Ins} est un foncteur $Mod_{Ins} : Sig_{Ins} \to Cat^{op}$ (où Cat^{op} est la catégorie de toutes les catégories).
  Mod_{Ins} associe à chaque signature Σ une catégorie des Σ-modèles $Mod_{Ins}(\Sigma)$ et à chaque morphisme de signature $\rho : \Sigma \to \Sigma'$ le foncteur $Mod_{Ins}(\rho) : Mod_{Ins}(\Sigma') \to Mod_{Ins}(\Sigma)$ qui traduit les modèles de Σ' en modèles de Σ. $(Mod_{Ins}(\rho))(M')$ souvent noté par $M'|_{\rho}$ est appelé *foncteur d'oubli*.
- $\models_{Ins} = \{ \models_{\Sigma} / \Sigma \in Sig_{Ins} \}$ est une collection de relations de validation indexée par les signatures de Sig_{Ins}, i.e. pour chaque signature Σ, on a une relation $\models_{Ins,\Sigma} \subseteq |Mod_{Ins}(\Sigma)| \times Sen_{Ins}(\Sigma)$.

- Le quadruplet $Ins = (Sig_{Ins}, Sen_{Ins}, Mod_{Ins}, \models_{Ins})$ doit vérifier la condition :

  $\forall \varphi \in Sen_{Ins} \forall M' \in Mod_{Ins}(\Sigma')$ on a $M' \models_{Ins,\Sigma'} \rho(\varphi) \quad \Leftrightarrow \quad M'|_{\rho} \models_{Ins,\Sigma} \varphi$

  Ceci signifie que pour tout morphisme de signature $\rho : \Sigma \to \Sigma'$, les traductions $Mod_{Ins}(\rho)$ et $Sen_{Ins}(\rho)$ associées au niveau des modèles et des formules respectivement doivent préserver les relations de validation.

Il convient de rappeler que la sémantique des ECATNets est décrite par des théories de réécriture ; et de ce fait, les signatures, les formules et les modèles considérés ici sont ceux des théories de réécriture correspondantes.

Après avoir défini la classe des modèles d'un ECATNet (dans le chapitre précédent), et l'institution qui facilitera la description de la sémantique de langage, nous pouvons à présent détailler la sémantique formelle de CIRTA. Pour ce faire, il suffit d'examiner successivement chaque type de construction d'une spécification et de lui associer un objet mathématique comme sémantique. Il convient de rappeler que la sémantique du langage CIRTA est définie au niveau modèle et les modèles d'une spécification sont toutes les réalisations possibles de cette spécification.

IV- Sémantique des modules

Une spécification dans le langage CIRTA repose sur l'utilisation de modules de base et de primitives de structuration. Un module de base peut être un module *ordinaire* ou *formel*.
Dans ce qui suit nous allons donner la théorie de réécriture associée à un module de base en s'inspirant de la théorie de réécriture d'un ECATNet temporel présentée au chapitre précédent. On précisera ensuite la classe des modèles des modules ordinaires et la classe des modèles des modules formels et la sémantique des modules génériques.

1- Théorie de réécriture $\widehat{\mathcal{T}}$ d'un module de base

Un module de base $\hat{E} = (E, I)$ est un ECATNet ayant éventuellement des places et/ou des transitions interfaces $I = (P_I, T_I)$.
Un module est destiné à coopérer avec son environnement (i.e. d'autres modules). Cet environnement peut :

1- Changer le marquage des places interfaces en ajoutant (resp. supprimant) des jetons dans l'une ou l'autre de ces places interfaces.
2- Rendre infranchissable une transition interface qui est *localement* franchissable dans le module en question.

La théorie de réécriture d'un module doit décrire son comportement quelque soit l'environnement dans lequel il est utilisé, et doit prendre en considération toutes les influences possibles (points 1 et 2 cités ci-dessus) de l'environnement sur le module. Ceci est réalisé comme suit :

1- Les ajouts/suppressions de jetons dans les places interfaces sont traduits au niveau théorie de réécriture par l'ajout, pour chaque place interface *p*, des règles de déduction suivantes :

$$\underline{\textit{Ajout-jeton(p) :}}\ \forall [s] : state \quad \forall\, m, M \in mT_{\Sigma_{\sigma(p)}}(\varnothing)$$

$$tc:\ [|m|] \rightarrow [c]$$

$$t : [s] \rightarrow [< p, \varnothing, c, M >]$$

$$\overline{aj(p, tc, t) : [s] \rightarrow [< p, m, 0, M >]}$$

$$\underline{\textit{Suppression-jeton(p) :}}\ \forall [s] : state \quad \forall\, m, M \in mT_{\Sigma_{\sigma(p)}}(\varnothing)$$

$$tc:\ [|m|] \rightarrow [c]$$

$$t : [s] \rightarrow [< p, m, 0, M >]$$

$$\overline{sj(p, tc, t) : [s] \rightarrow [< p, \varnothing, c, M >]}$$

2- Le comportement d'une transition interface *t* d'un module $\hat{\mathcal{E}} = (\mathcal{E}, I)$ est défini par l'union de toutes les séquences de franchissement de *t* dans tous les environnements possibles de $\hat{\mathcal{E}}$. Ceci donne exactement les séquences de franchissement de *t* dans l'ECATNet $\mathcal{E}$.

Compte tenu de ces considérations, nous pouvons définir la théorie de réécriture $\widehat{\mathcal{T}}$ d'un module de base, en utilisant la théorie de réécriture des ECATNets temporels, comme suit :

Définition :

Soit $\hat{\mathcal{E}} = (\mathcal{E}, I)$ un module de base. La théorie de réécriture du module $\hat{\mathcal{E}}$ est une théorie de réécriture étiquettée $\widehat{\mathcal{T}} = (\widehat{Sig},\ \hat{\varepsilon},\ \widehat{\mathcal{L}},\ \widehat{\mathcal{R}})$ telle que $\widehat{Sig},\ \hat{\varepsilon},\ \widehat{\mathcal{L}}$ *et* $\widehat{\mathcal{R}}$ sont définis par :.

$$\widehat{Sig} = Sig_{temp}$$

$$\hat{\varepsilon} = \varepsilon_{temp}$$

$$\widehat{\mathcal{L}} = \mathcal{L}_{temp}$$

$$\widehat{\mathcal{R}} = = \mathcal{R}_{temp} \cup \left(\cup_{p \in P_{\mathcal{T}}} \{ Ajout - jeton(p),\ Suppression - jeton(p) \} \right)$$

Remarque :

Il convient de noter que la notion de module de base est plus générale que celle d'un ECATNet. En effet, un ECATNet peut être considéré comme un module de base ayant une interface vide (i.e. $I = (\varnothing, \varnothing)$). Dans ce cas la théorie de réécriture définie ci-dessus coïncide avec celle de l'ECATNet temporel détaillée dans le chapitre précédent.

La théorie de réécriture d'un module de base engendre une classe de modèles. Dans le langage CIRTA, les modules de base pouvant être de deux types différents (*ordinaires* ou *formels*), il est nécessaire de préciser pour chacun des types le ou les modèles particuliers qui définissent leur sémantique.

2- Sémantique des modules ordinaires de base

La spécification est composée d'un seul module $\widehat{\mathcal{E}}$. La liste des modules paramètres formels et la liste des modules enrichis sont vides.

Soient $\widehat{\mathcal{E}}$: un module ordinaire de base;
$Mod\left(\widehat{\mathcal{E}}\right)$: classe des modèles de $\widehat{\mathcal{E}}$

Deux cas peuvent se produire:

Cas1: $Mod\left(\widehat{\mathcal{E}}\right)$ ne possède pas de modèle initial:

- Le module ordinaire $\widehat{\mathcal{E}}$ est inconsistant et sa sémantique est l'ensemble vide.
- La spécification ordinaire $\widehat{\mathcal{E}}$ est déclarée inconsistante et sa sémantique est l'ensemble vide.

Cas2: $Mod\left(\widehat{\mathcal{E}}\right)$ possède un modèle initial I :

- La sémantique de la spécification ordinaire $\widehat{\mathcal{E}}$ est (par définition) la classe de modèles réduite à ce modèle initial $\{I\}$
- La sémantique du module ordinaire $\widehat{\mathcal{E}}$ est (par définition) le foncteur unique de la catégorie 1 (catégorie constituée d'un seul objet) , vers la catégorie $Mod\left(\widehat{\mathcal{E}}\right)$ qui envoie l'unique objet de la catégorie 1 sur *I.*

3- Sémantique des modules paramètres formels de base

Soient $\widehat{\mathcal{E}}$: un module paramètre formel de base;
$Mod\left(\widehat{\mathcal{E}}\right)$: classe des modèles de $\widehat{\mathcal{E}}$

La spécification paramètre formel est constituée d'un seul module $\widehat{\mathcal{E}}$. La sémantique de la spécification paramètre formel $\widehat{\mathcal{E}}$ est par définition la catégorie des modèles $Mod\left(\widehat{\mathcal{E}}\right)$.

4- Sémantique des modules génériques

Soient $\widehat{\mathcal{E}}$: un module générique;
$\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ..., \widehat{\mathcal{E}}_k$: la liste des paramètres formels de $\widehat{\mathcal{E}}$.
$\widehat{\mathcal{E}}_0 = \cup_{i=1..k} \widehat{\mathcal{E}}_i$: la composition des modules $\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ..., \widehat{\mathcal{E}}_k$
$Mod\left(\widehat{\mathcal{E}}_0\right)$: la classe des modèles de $\widehat{\mathcal{E}}_0$
$Mod\left(\widehat{\mathcal{E}}\right)$: classe des modèles qui s'oublient en des modèles de $\widehat{\mathcal{E}}_0$ par le foncteur d'oubli $\breve{f}$ déterminé par l'inclusion des signatures $\Sigma\left(\widehat{E}_0\right) \subseteq \Sigma\left(\widehat{E}\right)$ (i.e. $\breve{f}\left(Mod\left(\widehat{\mathcal{E}}\right)\right) \subseteq Mod\left(\widehat{\mathcal{E}}_0\right)$).

Deux cas peuvent se produire:

Cas1: $Mod\left(\widehat{\mathcal{E}}\right)$ est vide

Le module de spécification $\widehat{\mathcal{E}}$ est hiérarchiquement inconsistant et sa sémantique est l'ensemble vide.

Cas2: $Mod\left(\widehat{\mathcal{E}}\right)$ n'est pas vide

Soit F_0 la classe de toutes les applications f_i telles que :

- f_i est un foncteur (total) de $Mod\left(\widehat{\mathcal{E}}_0\right)$ vers $Mod\left(\widehat{\mathcal{E}}\right)$
- f_i est inverse à droite du foncteur d'oubli $\breve{f}$, i.e. $\forall M \in Mod\left(\widehat{E}_0\right)\ \breve{f}(f_i(M)) = M$

- Si la classe F_0 est vide, alors le module générique est hiérarchiquement inconsistant et sa sémantique est l'ensemble vide.
- Si la classe F_0 n'est pas vide la sémantique du module générique est par définition la classe des foncteurs F_0, la sémantique de la spécification est par définition la classe des modèles $Mod\left(\widehat{\mathcal{E}}\right)$ de tous les modèles image par un des foncteur f_i des modèles M_i de la classe $Mod\left(\widehat{\mathcal{E}}_0\right)$: $Mod\left(\widehat{\mathcal{E}}\right) = \cup_{f_i \in F_0} f_i\left(Mod\left(\widehat{\mathcal{E}}_0\right)\right)$

V- Sémantique des primitives du langage

1- Sémantique de l'enrichissement

Soit $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}' + \Delta\mathcal{E}$ un module de spécification qui enrichi le module $\widehat{\mathcal{E}}'$.

$Mod\left(\widehat{\mathcal{E}}'\right)$: la classe des modèles de $\widehat{\mathcal{E}}'$.

$Mod\left(\widehat{\mathcal{E}}\right)$: classe des modèles qui s'oublient en des modèles de $\widehat{\mathcal{E}}'$ par le foncteur d'oubli $\breve{f}$ déterminé par l'inclusion des signatures $\Sigma\left(\hat{E}'\right) \subseteq \Sigma\left(\hat{E}\right)$ (i.e. $\breve{f}\left(Mod\left(\widehat{\mathcal{E}}\right)\right) \subseteq Mod\left(\widehat{\mathcal{E}}'\right)$).

Deux cas peuvent se produire:

Cas1: $Mod\left(\widehat{\mathcal{E}}\right)$ est vide

Le module de spécification $\widehat{\mathcal{E}}$ est hiérarchiquement inconsistant et sa sémantique est l'ensemble vide.

Cas2: $Mod\left(\widehat{\mathcal{E}}\right)$ n'est pas vide

Soit F_0 la classe de toutes les applications f_i telles que :

- f_i est un foncteur (total) de $Mod\left(\widehat{\mathcal{E}}'\right)$ vers $Mod\left(\widehat{\mathcal{E}}\right)$
- f_i est inverse à droite du foncteur d'oubli $\breve{f}$, i.e. $\forall M \in Mod\left(\hat{E}'\right)\ \ \breve{f}(f_i(M)) = M$

- Si la classe F_0 est vide, alors le module $\widehat{\mathcal{E}}$ est hiérarchiquement inconsistant et sa sémantique est l'ensemble vide.
- Si la classe F_0 n'est pas vide la sémantique du module $\widehat{\mathcal{E}}$ est par définition la classe des foncteurs F_0, la sémantique de la spécification est par définition la classe des modèles $Mod\left(\widehat{\mathcal{E}}\right)$ de tous les modèles images par un des foncteur f_i des modèles M_i de la classe $Mod\left(\widehat{\mathcal{E}}'\right)$: $\quad Mod\left(\widehat{\mathcal{E}}\right) = \cup_{f_i \in F_0} f_i\left(Mod\left(\widehat{\mathcal{E}}'\right)\right)$

2- Sémantique de la composition

Soient $\widehat{\mathcal{E}} = \widehat{\mathcal{E}}_1 \cup \widehat{\mathcal{E}}_2 \cup \ldots \cup \widehat{\mathcal{E}}_k$: un module qui compose les modules $\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, \ldots, \widehat{\mathcal{E}}_k$.

$\widehat{\mathcal{E}}_0$: le module plat associé à la composition des modules $\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, \ldots, \widehat{\mathcal{E}}_k$

$Mod\left(\widehat{\mathcal{E}}_i\right)$: la classe des modèles du module $\widehat{\mathcal{E}}_i$ (pour i=0..k)

$Mod\left(\widehat{\mathcal{E}}_0\right) = \left\{M \,/\, \forall i = 1..k \ \ M_{|\widehat{\mathcal{E}}_i} \in Mod\left(\widehat{\mathcal{E}}_i\right)\right\}$:

$Mod\left(\widehat{\mathcal{E}}\right)$: classe des modèles qui s'oublient en des modèles de $\widehat{\mathcal{E}}_0$ par le foncteur d'oubli $\breve{f}$ déterminé par l'inclusion des signatures $\Sigma\left(\hat{E}_0\right) \subseteq \Sigma\left(\hat{E}\right)$ (i.e. $\breve{f}\left(Mod\left(\widehat{\mathcal{E}}\right)\right) \subseteq Mod\left(\widehat{\mathcal{E}}_0\right)$).

Deux cas peuvent se produire:

Cas1: $Mod\left(\hat{\mathcal{E}}\right)$ est vide

Le module de spécification $\hat{\mathcal{E}}$ est hiérarchiquement inconsistant et sa sémantique est l'ensemble vide.

Cas2: $Mod\left(\hat{\mathcal{E}}\right)$ n'est pas vide

Soit F_0 la classe de toutes les applications f_i telles que :

- f_i est un foncteur (total) de $Mod\left(\hat{\mathcal{E}}_0\right)$ vers $Mod\left(\hat{\mathcal{E}}\right)$
- f_i est inverse à droite du foncteur d'oubli $\breve{f}$, i.e. $\forall M \in Mod\left(\hat{E}_0\right)\ \ \breve{f}\,(\,f_i(M)) = M$

- Si la classe F_0 est vide, alors le module $\hat{\mathcal{E}}$ est hiérarchiquement inconsistant et sa sémantique est l'ensemble vide.
- Si la classe F_0 n'est pas vide la sémantique du module $\hat{\mathcal{E}}$ est par définition la classe des foncteurs F_0 , la sémantique de la spécification est par définition la classe des modèles $Mod\left(\hat{\mathcal{E}}\right)$ de tous les modèles image par un des foncteur f_i des modèles M_i de la classe $Mod\left(\hat{\mathcal{E}}_0\right)$: $\quad Mod\left(\hat{\mathcal{E}}\right) = \cup_{f_i \in F_0}\, f_i\left(Mod\left(\hat{\mathcal{E}}_0\right)\right)$

3- Sémantique du renommage

Soient $\hat{\mathcal{E}}'$: un module d'ECATNet .

$\hat{\mathcal{E}}$ un module construit par renommage de $\hat{\mathcal{E}}'$

$Mod\left(\hat{\mathcal{E}}'\right)$ la classe des modèle de $\hat{\mathcal{E}}'$

$Mod\left(\hat{\mathcal{E}}\right)$ = la classe des modèles de $\hat{\mathcal{E}}$

La sémantique de $\hat{\mathcal{E}}$ est identique à la sémantique de $\hat{\mathcal{E}}'$ au nom prés (i.e. les classes de modèles sont telles que : $\quad Mod\left(\hat{\mathcal{E}}\right) = Mod\left(\hat{\mathcal{E}}'\right)$

Ainsi si $\hat{\mathcal{E}} = Rename(\hat{\mathcal{E}}', Morph)$ est un module qui renomme $\hat{\mathcal{E}}'$ par le morphisme $Morph$, la sémantique de $\hat{\mathcal{E}}$ et $\hat{\mathcal{E}}'$ sont identiques au renommage prés.

4- Sémantique du contrôle de visibilité

La primitive de contrôle de visibilité peut être supposée de type *Hide*, le cas de la primitive de type *Export* étant traité de façon symétrique.

Le traitement du contrôle de la visibilité est tout à fait analogue au traitement du renommage : on déduit des déclarations contenues dans la partie V*isibilité* un morphisme $\hat{\rho}$ défini sur $< S, Op, P, T >$ l'ensemble global des sortes, des opérations, des places et des transitions

associés au module $\widehat{\mathcal{E}}$. La différence avec le renommage est que la partie Visibilité n'est pas constituée de couples du type <ancien-nom, nouveau-nom> , mais simplement d'une liste d'éléments qui ne doivent plus être visibles à l'extérieur du module. Le morphisme $\widehat{\rho}$ doit donc renommer les éléments à masquer en des éléments dont le nom n'est plus accessible. Pour simplifier, et afin d'éviter des considérations trop proches d'une implémentation du mécanisme de contrôle de visibilité, nous supposons donné une application injective notée $[\![\ ]\!]$, définie sur l'ensemble tous les identificateurs (des sortes, d'opérations, de places et des transitions) légaux (pour la syntaxe concrète du langage) et à valeurs dans un domaine de noms disjoint. Ainsi rendre l'élément *e* non visible revient à appliquer le renommage de *e* par $[\![e]\!]$. Si *e* est un élément à masquer alors le morphisme de renommage sera construit des éléments $(e,[\![e]\!])$. Comme dans le cas du renommage, masquer (i.e. renommer) une sorte *s* induit implicitement le masquage (renommage) de toutes les opérations dont le profil fait intervenir cette sorte. Il en est de même pour les places interfaces de sorte *s*, et les variables de sortes *s* dans les contextes des transitions interfaces. Le raisonnement est analogue pour le masquage des opérations, des places interfaces et des transitions interfaces.

5- Sémantique de l'instanciation

5-1 Correction du passage de paramètre

Pour chaque instanciation élémentaire $\left\langle \widehat{\mathcal{F}}_i, \widehat{\mathcal{E}}_i, h_i \right\rangle$ (où , $\widehat{\mathcal{F}}_i, \widehat{\mathcal{E}}_i, h_i$ sont respectivement le module paramètre formel, le module paramètre effectif et le morphisme d'adéquation), il est nécessaire de s'assurer que la spécification $\hat{\mathcal{E}}_i$ est bien un paramètre admissible pour la spécification $\widehat{\mathcal{F}}_i$.

Définition :

Le module de spécification $\hat{\mathcal{E}}_i$ est un paramètre effectif acceptable pour le module paramètre formel $\widehat{\mathcal{F}}_i$

$$\breve{f}_{h_i}\left(Mod\left(\widehat{\mathcal{E}}_i\right)\right) \subseteq Mod\left(\widehat{\mathcal{F}}_i\right)$$

où $Mod\left(\widehat{\mathcal{E}}_i\right)$: est la classe des modèles de $\hat{\mathcal{E}}_i$

$Mod\left(\widehat{\mathcal{F}}_i\right)$: est la classe des modèles de $\widehat{\mathcal{F}}_i$

$\breve{f}_{h_i}$:est le foncteur d'oubli des modèles de $Mod\left(\widehat{\mathcal{E}}_i\right)$ vers des modèles induits par le morphisme d'adéquation h_i.

5-2 Instanciation

On suppose que les contraintes relatives aux aspects de la sémantique statique sont vérifiées et que le passage de paramètre est correct au sens de la définition précédente.
Informellement l'instanciation d'un module générique se ramène à l'application de la primitive de renommage, d'une composition d'un ensemble de modules et d'un enrichissement.

Considérons le cas général de l'instanciation d'un module générique $\hat{\mathcal{E}}$ ayant une liste de paramètres formels $\left[\hat{\mathcal{F}}_1, \hat{\mathcal{F}}_2, ...\hat{\mathcal{F}}_k, \hat{\mathcal{F}}_{k+1}....\hat{\mathcal{F}}_p, \hat{\mathcal{F}}_{p+1}, ..., \hat{\mathcal{F}}_n\right]$ où :

- $L_1 = \left[\hat{\mathcal{F}}_1, \hat{\mathcal{F}}_2, ...\hat{\mathcal{F}}_k\right]$ est la liste des paramètres formels qui vont être substitués par des modules *ordinaires* $\left[\hat{\mathcal{E}}_1, \hat{\mathcal{E}}_2, ...\hat{\mathcal{E}}_k\right]$ via les morphismes d'adéquation respectifs $[h_1, h_2, ..., h_k]$ (instanciation effective).
- $L_2 = \left[\hat{\mathcal{F}}_{k+1}....\hat{\mathcal{F}}_p\right]$ est la liste des paramètres formels qui vont être substitués par des modules *formels* $\left[\hat{\mathcal{E}}_{k+1}, \hat{\mathcal{E}}_{k+2}, ...\hat{\mathcal{E}}_p\right]$ via les morphismes d'adéquation respectifs $\left[h_{k+1}, ...h_p\right]$ (instanciation formelle).
- $L_3 = \left[\hat{\mathcal{F}}_{p+1}, ..., \hat{\mathcal{F}}_n\right]$ est la liste des paramètres formels non concernés par l'instanciation. L'instanciation est totale si $L_3 = \varnothing$, sinon elle est partielle.

Selon que l'une et/ou l'autre de ces listes (L_1, L_2 *et* L_3) soit vide ou non, la sémantique de l'instanciation diffère et elle est équivalente à la sémantique d'un module ordinaire ou à celle d'un module générique. Plus précisément nous présentons les différentes six cas possibles et nous indiquons pour chacun d'eux la sémantique du type considéré d'instanciation.

1. $L_1 = \varnothing \wedge L_2 \neq \varnothing \wedge L_3 = \varnothing$ L'instanciation globale du module est formelle et totale

   L'instanciation du module générique $\hat{\mathcal{E}}\left[\hat{\mathcal{F}}_{k+1}....\hat{\mathcal{F}}_p\right]$ par les modules $\left[\hat{\mathcal{E}}_{k+1}, \hat{\mathcal{E}}_{k+2}, ...\hat{\mathcal{E}}_p\right]$ via les morphismes $\left[h_{k+1}, ...h_p\right]$ est notée $\hat{\mathcal{E}}\left[h_{k+1}, ..., h_p\right]$ et consiste à générer le module $\hat{\mathcal{E}}'$ défini par :
   - $\hat{\mathcal{E}}' = rename\left(\hat{\mathcal{E}}, \cup_{i=k+1..p} h_i\right) + combine\left(\hat{\mathcal{E}}_{k+1}, \hat{\mathcal{E}}_{k+2}, ...\hat{\mathcal{E}}_p\right)$.
   - La sémantique de l'instanciation $\hat{\mathcal{E}}\left[h_{k+1}, ..., h_p\right]$ est la sémantique du module générique $\hat{\mathcal{E}}'\left[\hat{\mathcal{E}}_{k+1}....\hat{\mathcal{E}}_p\right]$.

2. $L_1 = \varnothing \wedge L_2 \neq \varnothing \wedge L_3 \neq \varnothing$ L'instanciation globale du module est formelle partielle

   L'instanciation du module générique $\hat{\mathcal{E}}\left[\hat{\mathcal{F}}_{k+1}....\hat{\mathcal{F}}_p, \hat{\mathcal{F}}_{p+1}, ..., \hat{\mathcal{F}}_n\right]$ par les modules $\left[\hat{\mathcal{E}}_{k+1}, \hat{\mathcal{E}}_{k+2}, ...\hat{\mathcal{E}}_p\right]$ via les morphismes $\left[h_{k+1}, ...h_p\right]$ est notée $\hat{\mathcal{E}}\left[h_{k+1}, ..., h_p\right]$ et consiste à générer le module $\hat{\mathcal{E}}'$ défini par :
   - $\hat{\mathcal{E}}' = rename\left(\hat{\mathcal{E}}, \cup_{i=k+1..p} h_i\right) + combine\left(\hat{\mathcal{E}}_{k+1}, \hat{\mathcal{E}}_{k+2}, ...\hat{\mathcal{E}}_p\right)$.
   - La sémantique de l'instanciation $\hat{\mathcal{E}}\left[h_{k+1}, ..., h_p\right]$ est la sémantique du module générique $\hat{\mathcal{E}}'\left[\hat{\mathcal{E}}_{k+1}....\hat{\mathcal{E}}_p, \hat{\mathcal{F}}_{p+1}, ..., \hat{\mathcal{F}}_n\right]$.

3. $L_1 \neq \varnothing \wedge L_2 = \varnothing \wedge L_3 = \varnothing$ L'instanciation globale du module est effective totale

L'instanciation du module générique $\widehat{\mathcal{E}}\left[\widehat{\mathcal{F}}_1, \widehat{\mathcal{F}}_2, ...\widehat{\mathcal{F}}_k\right]$ par les modules $\left[\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k\right]$ via les morphismes $[h_1, ..., h_k]$ est notée $\widehat{\mathcal{E}}[\, h_1, ..., h_k]$ et consiste à générer le module $\widehat{\mathcal{E}}'$ défini par :

- $\widehat{\mathcal{E}}' = rename\left(\widehat{\mathcal{E}}, \cup_{i=1..k} h_i\right) + combine\left(\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k\right)$.
- La sémantique de l'instanciation $\widehat{\mathcal{E}}[\, h_1, ..., h_k]$ est la sémantique du module ordinaire $\widehat{\mathcal{E}}'$.

4. $L_1 \neq \varnothing \wedge L_2 = \varnothing \wedge L_3 \neq \varnothing$ L'instanciation globale du module est effective partielle

L'instanciation du module générique $\widehat{\mathcal{E}}\left[\widehat{\mathcal{F}}_1, \widehat{\mathcal{F}}_2, ...\widehat{\mathcal{F}}_k, \widehat{\mathcal{F}}_{p+1}, ..., \widehat{\mathcal{F}}_n\right]$ par les modules $\left[\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k\right]$ via les morphismes $[h_1, ..., h_k]$ est notée $\widehat{\mathcal{E}}[\, h_1, ..., h_k]$ et consiste à générer le module $\widehat{\mathcal{E}}'$ défini par :

- $\widehat{\mathcal{E}}' = rename\left(\widehat{\mathcal{E}}, \cup_{i=1..k} h_i\right) + combine\left(\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k\right)$.
- La sémantique de l'instanciation $\widehat{\mathcal{E}}[\, h_1, ..., h_k]$ est la sémantique du module générique $\widehat{\mathcal{E}}'\left[\widehat{\mathcal{F}}_{p+1}, ..., \widehat{\mathcal{F}}_n\right]$.

5. $L_1 \neq \varnothing \wedge L_2 \neq \varnothing \wedge L_3 = \varnothing$ L'instanciation globale du module est totale

L'instanciation du module générique $\widehat{\mathcal{E}}\left[\widehat{\mathcal{F}}_1, \widehat{\mathcal{F}}_2, ...\widehat{\mathcal{F}}_k, \widehat{\mathcal{F}}_{k+1}....\widehat{\mathcal{F}}_p\right]$ par les modules $\left[\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k, \widehat{\mathcal{E}}_{k+1}, ...\widehat{\mathcal{E}}_p\right]$ via les morphismes $\left[h_1, h_2, ..., h_k, h_{k+1}, ...h_p\right]$ est notée $\widehat{\mathcal{E}}[\, h_1, h_2, ..., h_k, h_{k+1}, ..., h_p]$ et consiste à générer le module $\widehat{\mathcal{E}}'$ défini par :

- $\widehat{\mathcal{E}}' = rename\left(\widehat{\mathcal{E}}, \left(\cup_{i=1..k} h_i\right) \cup \left(\cup_{i=k+1..p} h_i\right)\right) + combine\left(\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k, \widehat{\mathcal{E}}_{k+1}, \widehat{\mathcal{E}}_{k+2}, ...\widehat{\mathcal{E}}_p\right)$.
- La sémantique de l'instanciation $\widehat{\mathcal{E}}[\, h_1, h_2, ..., h_k, h_{k+1}, ..., h_p]$ est la sémantique du module générique $\widehat{\mathcal{E}}'\left[\widehat{\mathcal{E}}_{k+1}....\widehat{\mathcal{E}}_p\right]$.

6. $L_1 \neq \varnothing \wedge L_2 \neq \varnothing \wedge L_3 \neq \varnothing$ L'instanciation globale du module est partielle

L'instanciation du module générique $\widehat{\mathcal{E}}\left[\widehat{\mathcal{F}}_1, \widehat{\mathcal{F}}_2, ...\widehat{\mathcal{F}}_k, \widehat{\mathcal{F}}_{k+1}....\widehat{\mathcal{F}}_p, \widehat{\mathcal{F}}_{p+1}, ..., \widehat{\mathcal{F}}_n\right]$ par les modules $\left[\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k, \widehat{\mathcal{E}}_{k+1}, ...\widehat{\mathcal{E}}_p\right]$ via les morphismes $\left[h_1, h_2, ..., h_k, h_{k+1}, ...h_p\right]$ est notée $\widehat{\mathcal{E}}[\, h_1, h_2, ..., h_k, h_{k+1}, ..., h_p]$ et consiste à générer le module $\widehat{\mathcal{E}}'$ défini par :

- $\widehat{\mathcal{E}}' = rename\left(\widehat{\mathcal{E}}, \left(\cup_{i=1..k} h_i\right) \cup \left(\cup_{i=k+1..p} h_i\right)\right) + combine\left(\widehat{\mathcal{E}}_1, \widehat{\mathcal{E}}_2, ...\widehat{\mathcal{E}}_k, \widehat{\mathcal{E}}_{k+1}, \widehat{\mathcal{E}}_{k+2}, ...\widehat{\mathcal{E}}_p\right)$.
- La sémantique de l'instanciation $\widehat{\mathcal{E}}[\, h_1, h_2, ..., h_k, h_{k+1}, ..., h_p]$ est la sémantique du module générique $\widehat{\mathcal{E}}'\left[\widehat{\mathcal{E}}_{k+1}....\widehat{\mathcal{E}}_p, \widehat{\mathcal{F}}_{p+1}, ..., \widehat{\mathcal{F}}_n\right]$.

VI- Conclusion

Nous avons défini la sémantique du langage CIRTA au niveau classe de modèles. Dans une première étape, nous avons défini la théorie de réécriture d'un *module* de base en généralisant la théorie de réécriture des ECATNets temporels. Ceci permet de préciser la classe des modèles d'un module de base qui n'est autre que la catégorie des modèles de la théorie de réécriture correspondante. Il est clair que cette classe de modèles dépend de la version des ECATNets utilisée (simples, contextuels, temporels,...etc.).
Dans une seconde étape, nous avons précisé la sémantique des modules ordinaires, formels et génériques du langage CIRTA. Enfin nous avons décrit la sémantique des primitives du langage qui est, dans une très large mesure, indépendante de la version d'ECATNets utilisée et s'adapte sans grande difficulté à d'autres versions des ECATNets.

Chapitre 10

Etudes de Cas

Système de contrôle de voies ferrées

Sommaire

I- Introduction

Dans ce chapitre, nous allons décrire une étude de cas significative réalisée avec le langage de spécification CIRTA. Cette étude de cas a un double intérêt. En premier lieu, elle démontre comment spécifier de façon relativement aisée et progressive un exemple non trivial de système réel. En second lieu, elle nous permet d'évaluer les facilités offertes par le langage CIRTA, tant en ce qui concerne les primitives de structuration des spécifications que les facilités de nature syntaxique définies pour améliorer la lisibilité des spécifications.

Pour cette étude de cas, nous présentons le système de façon informelle et nous détaillons les principaux modules de spécification correspondants.

II- Présentation générale du système

Dans ce chapitre, nous allons décrire comment un exemple de système de contrôle de voies ferrées est spécifié progressivement en utilisant le langage CIRTA.

Le modèle du système considéré est représenté par la Figure 10.1. Dans ce système, la voie ferrée est composée de 16 tronçons (B1, B2,…, B16) et de deux voies secondaires (V1 et V2) liées par 4 commutateurs (C1, C2, C3, C4) et une intersection (ou croisement) CROSS. La manière suivant laquelle les trains peuvent traverser les commutateurs et l'intersection est indiquée par des arcs dans la Figure 10.1.
La circulation sur tous les tronçons peut être dans les deux sens. De plus on note que les commutateurs C1, C2, C3 et C4 ne sont pas identiques : chacun d'eux a une structure et une fonction propre.

La voie ferrée est connectée à un ordinateur via un port série qui permet de lire les informations à partir de capteurs. L'ordinateur envoie des ordres aux trains à travers les rails ou directement aux commutateurs. Chaque tronçon est équipé avec un capteur à chacune de ses extrémités pour détecter l'entrée ou la sortie d'un train. Les ordres envoyés aux trains peuvent être soit « STOP » ou « GO AVANT/ARRIERE» à une vitesse donnée.

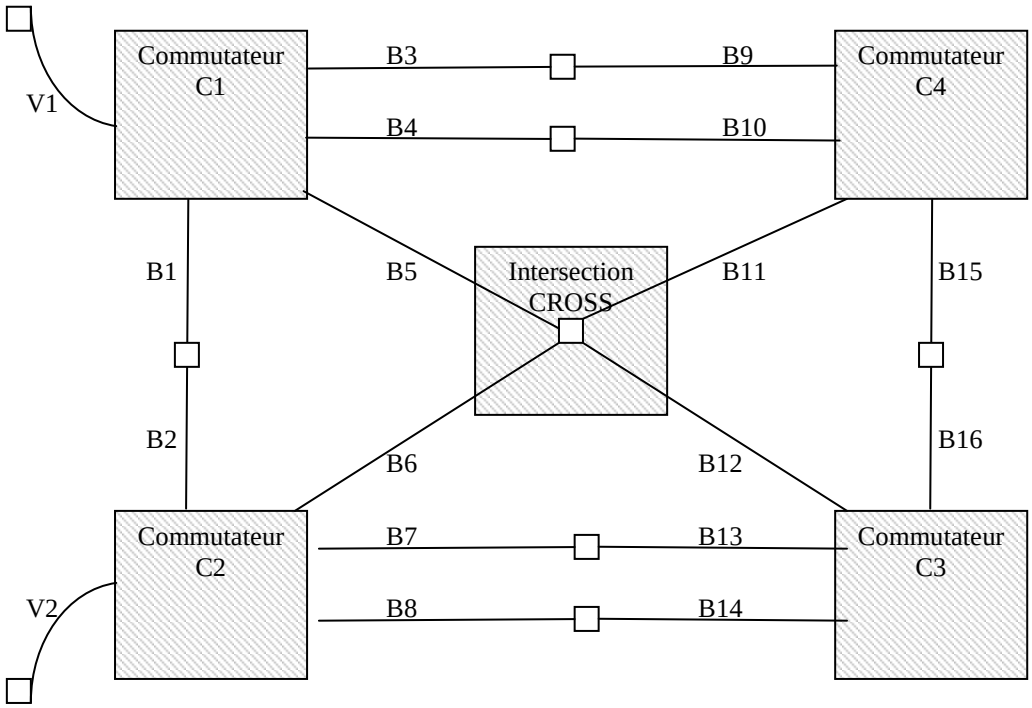


Figure 10.1 : Schéma des voies ferrées.

III- Spécification du module principal

Dans cette étude de cas, l'utilisation des transitions de substitution permet de faire abstraction dans un premier temps du traitement effectué par certains composants du système.

Le module principal présente la voie ferrée sans aucune considération de la stratégie utilisée pour gérer le déplacement des trains d'un tronçon vers un autre. Cette stratégie est décrite par des sous-modules correspondants aux différents commutateurs C_i (i=1..4), à l'intersection Cross et aux différentes transitions ($Move_i$) entre les tronçons adjacents.

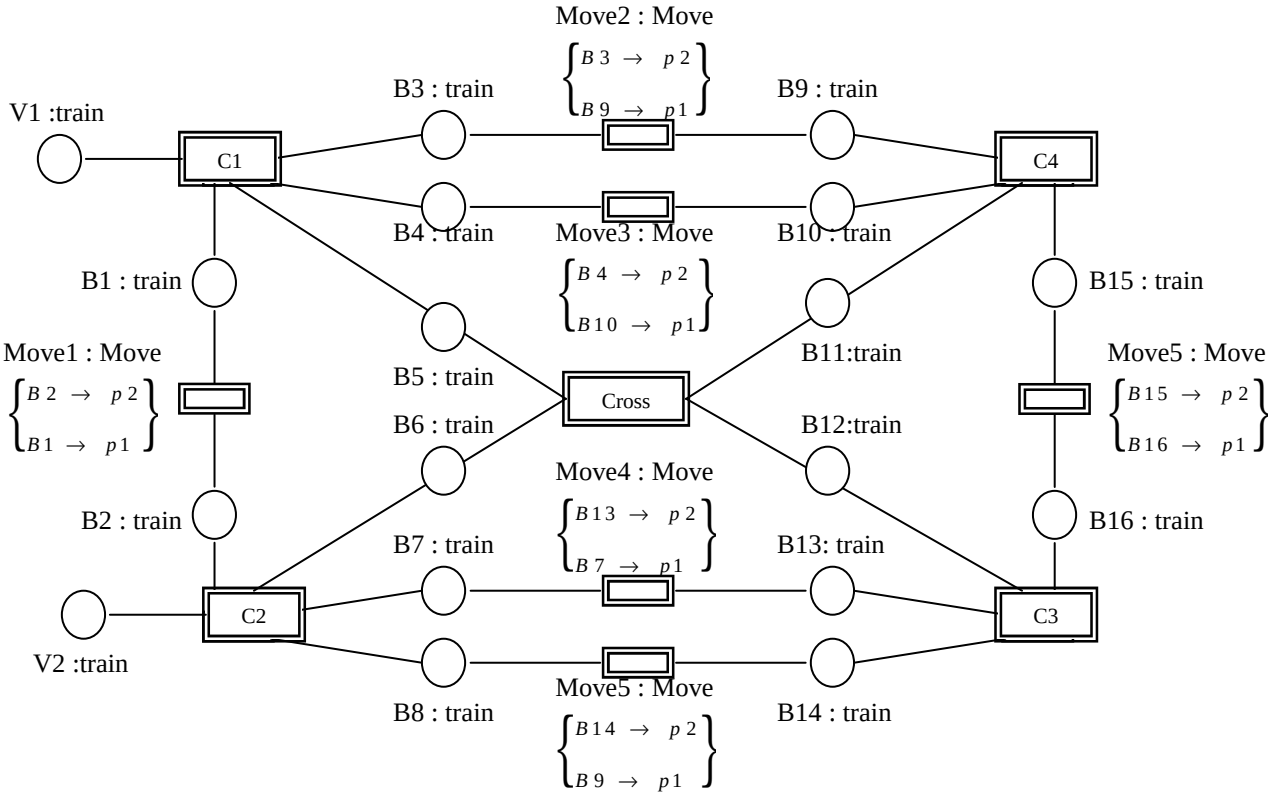


Figure 10.2: Module principal du système de contrôle de voies ferrées.

A première vue, il y a une similarité entre le système physique de la voie ferrée (Figure 10.1) et le module de spécification principal (Figure 10.2).

- Les places représentent les tronçons (ils ont les mêmes identificateurs dans les deux figures).
- Les transitions indiquent les différents mouvements possibles.

La présence d'un jeton *tr* dans la place B_i indique que le train *tr* circule sur le tronçon *Bi.*
La présence d'un jeton *tr* dans la place V_i indique que le train *tr* circule sur la voie secondaire *Vi*

Le déplacement d'un train d'un tronçon *Bi* vers le tronçon *Bj* (ou l'inverse) est décrit par la transition de substitution *Move*.

Les transitions de substitution *Ci* sont associées aux modules qui spécifient le comportement des commutateurs.

La transition de substitution Cross décrit la gestion de l'intersection.

IV- Spécification des trains

Dans ce système de contrôle de voies ferrées, on suppose que chaque train est identifié par un nom unique. L'ensemble de ces identificateurs (t_1, t_2,…t_n) est spécifié par le module TRAIN-ID ci-dessous.

Spec TRAIN-ID **is**
  Sort id
  Ops $t_1 :\rightarrow$ id
    $t_2 :\rightarrow$ id
    $t_3 :\rightarrow$ id
    ….
    $t_n :\rightarrow$ id
end

Figure 10.3: Module de spécification des identificateurs des trains.

La circulation des trains sur les tronçons peut être dans les deux directions. A un instant donné, un train roule sur le circuit dans le sens des aiguilles d'une montre ou dans le sens inverse des aiguilles d'une montre. On définit le module SENS pour spécifier les sens de circulation autorisés sur les tronçons.

Spec SENS **is**
  Sort sens
  Ops + :→ sens
    -:→ sens
end

Figure 10.4: Module de spécification des sens de circulation des trains.

Les opérations « + » et « - » désignent respectivement le sens des aiguilles d'une montre et le sens inverse.

Un train circulant sur un tronçon est défini par son nom et sa direction.

Spec TRAIN **is**
  Use TRAIN-ID SENS
  Sort train
  Ops <_,_> : id sens→ train
end

Figure 10.5: Module de spécification des trains.

Un terme < *t*,+> indique que le train *t* roule sur le circuit dans le sens des aiguilles d'une montre.

Un terme < *t*,-> indique que le train *t* roule sur le circuit dans le sens inverse des aiguilles d'une montre.

V- Spécification des mouvements entre deux tronçons

Plusieurs mouvements sont possibles entre deux tronçons. Les transitions de substitution de type *Move* permettent des mouvements entre tronçons adjacents sans sélection (ce qui n'est pas le cas pour les commutateurs). Toutes les transitions de ce type ont le même comportement. Par conséquent, toutes ces transitions de substitution font référence au même module.

La spécification du module *Move* est présentée de façon modulaire où chaque module décrit un mouvement élémentaire possible entre deux tronçons.

1- Module Move-positif

Un train peut passer d'un tronçon P2 vers un tronçon P1 en circulant dans le sens « + »

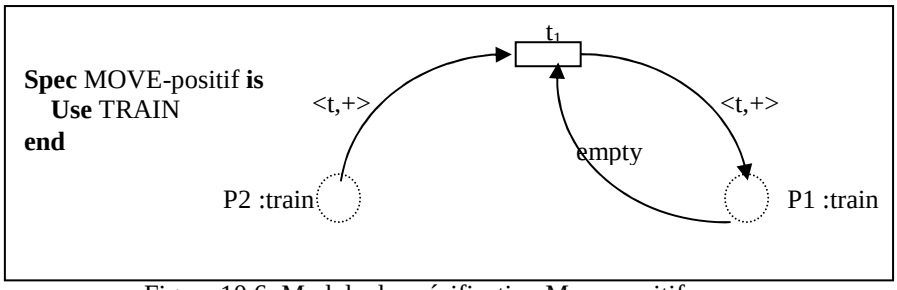


Figure 10.6: Module de spécification Move-positif.

Dans la Figure 10.6 la transition t_1 est franchissable s'il y a un train *t* allant dans le sens « + » dans la place P2 et il n'y a aucun train ($IC(P1,t_1)$=*« empty »*) dans la place P1. Le franchissement de cette transition correspond au déplacement du train du tronçon P2 vers le tronçon P1.

2- Module Move-negatif

Par analogie au module précédent, un train peut passer d'un tronçon P1 vers un tronçon P2 en circulant dans le sens « - ».

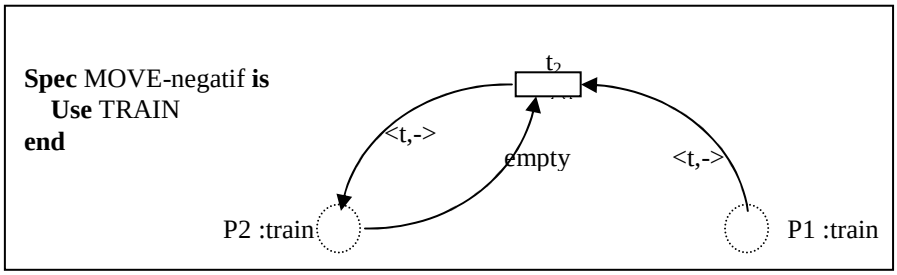


Figure 10.7: Module de spécification Move-negatif.

3- Module Move-back

Pour éviter certaines situations de blocage, les trains peuvent faire marche arrière (changer le sens de leur parcours) comme spécifié par le module de la Figure 10.8.

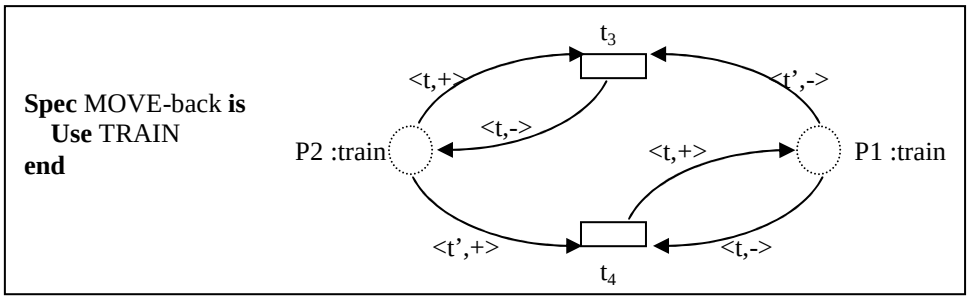


Figure 10.8: Module de spécification Move-back.

Un train *t* effectue une marche arrière dans l'une des situations suivantes :

- *t* circule sur un tronçon P2 dans le sens positif, et il y a un autre train *t'* circulant sur le tronçon P1 (P2 adjacent à P1) dans le sens négatif.
- *t* circule sur un tronçon P1 dans le sens négatif, et il y a un autre train *t'* circulant sur le tronçon P2 (P2 adjacent à P1) dans le sens positif.

Sachant que P2 est adjacent à P1, le retour en arrière d'au moins l'un des trains sert à éviter une collision.

Notons que tous les trains sont traités de la même manière. En effet, le choix du train qui doit faire marche arrière (dans une situation de blocage), est aléatoire et correspond à l'indéterminisme dans le franchissement de la transition t_3 et/ou t_4.

4- Module Move

Le module Move qui spécifie tous le déplacements possibles d'un train sur les tronçons est la composition des modules Move-positif, Move-negatif et Move-back décrivant chacun un type de déplacement sur les tronçons.

```
Spec Move is
   Combine Move-positif  Move-negatif  Move-back
end
```

Figure 10.9: Module de spécification Move.

VI- Spécification des commutateurs

Les commutateurs permettent de faire des mouvements soit d'un tronçon unique vers deux autres tronçons différents ou vice versa. Les commutateurs C1 C2 C3 et C4, ont des comportements différents. Pour chacun d'eux on précisera les tronçons concernés par son traitement, et on donnera le module de spécification correspondant.

1- Commutateur C1

Le commutateur C1 permet les mouvements entre les tronçons suivants :

- V1 et B3
- B1 et B3
- B1 et B4
- B1 et B5

Ceci peut être schématisé par la Figure 10.10 ci-dessous.

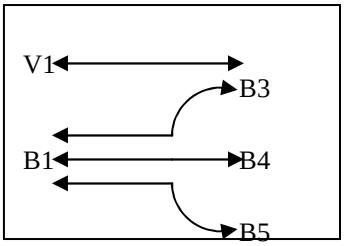


Figure 10.10: Schéma du commutateur C1.

Chaque mouvement « élémentaire » dans C1 correspond à un passage entre deux tronçons particuliers. Ces différents mouvements sont spécifiés par les modules CIRTA de la Figure10.11.

```
Spec C1-V1B3 is
  Rename Move by
        Place   p2→V1
        Place   p1 →B3
end
```

```
Spec C1-B1B3 is
  Rename Move by
        Place   p2→B1
        Place   p1 →B3
end
```

```
Spec C1-B1B4 is
  Rename Move by
        Place   p2→B1
        Place   p1 →B4
end
```

```
Spec C1-B1B5 is
  Rename Move by
        Place   p2→B1
        Place   p1 →B5
end
```

Figure 10.11: Modules de spécification
des mouvements possibles dans le commutateur C1.

```
Spec C1 is
  Combine C1-B1B3   C1-B1B4   C1-B1B5   C1-V1B3
end
```

Figure 10.12: Module de spécification du commutateur C1.

2- Commutateur C2

Le commutateur C2 permet les mouvements entre les tronçons suivants :

- B9 et B15
- B10 et B15
- B11 et B15

Ceci peut être schématisé par la Figure 10.13 ci-dessous.

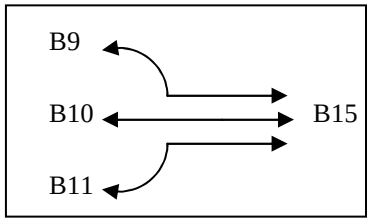


Figure 10.13: Schéma du commutateur C2.

Les différents mouvements élémentaires de C2 sont spécifiés par les modules de la Figure10.14.

```
Spec C2-B9B15 is
  Rename Move by
        Place   p2→B9
        Place   p1 →B15
end
```

```
Spec C2-B10B15 is
  Rename Move by
        Place   p2→B10
        Place   p1 →B15
end
```

```
Spec C2-B11B15 is
  Rename Move by
        Place   p2→B11
        Place   p1 →B15
end
```

Figure 10.14: Modules de spécification
des mouvements possibles dans le commutateur C2.

```
Spec C2 is
  Combine C2-B9B15    C2-B10B15    C2-B11B15
end
```

Figure 10.15: Module de spécification du commutateur C2.

3- Commutateur C3

Le commutateur C3 permet les mouvements suivants :

- B12 et B16
- B13 et B16
- B14 et B16

Ceci peut être schématisé par la Figure 10.16 ci-dessous.

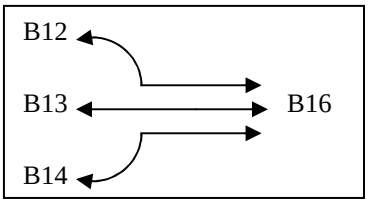


Figure 10.16: Schéma du commutateur C3.

Les modules CIRTA spécifiant les mouvements de C3 sont décrits par la Figure 10.17.

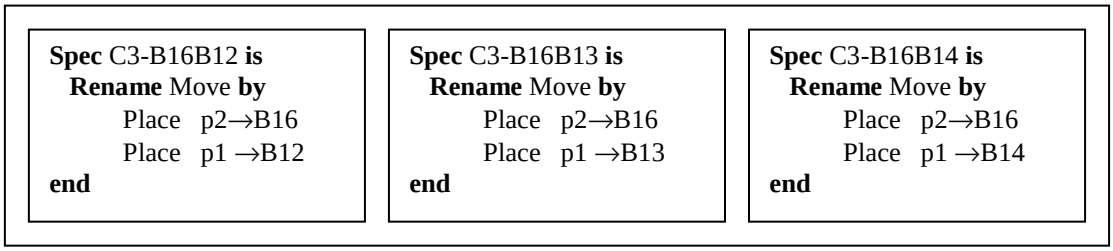


Figure 10.17: Modules de spécification
des mouvements possibles dans le commutateur C3.

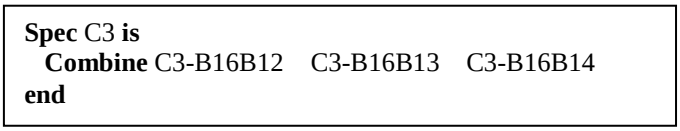


Figure 10.18: Module de spécification du commutateur C3.

4- Commutateur C4

Le commutateur C4 permet les mouvements suivants :

- B6 et B2
- B7 et B2
- B8 et B2
- V2 et B2

Ceci peut être schématisé par la Figure 10.19 ci-dessous.

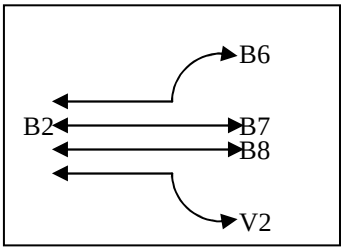


Figure 10.19: Schéma du commutateur C4.

Les mouvements possibles dans le commutateur C4 sont spécifiés par les modules CIRTA de la Figure 10.20.

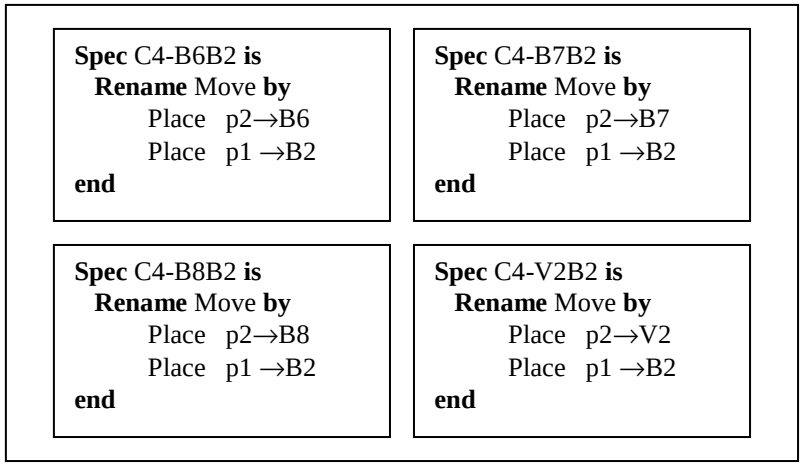


Figure 10.20: Modules de spécification
des mouvements possibles dans le commutateur C4.

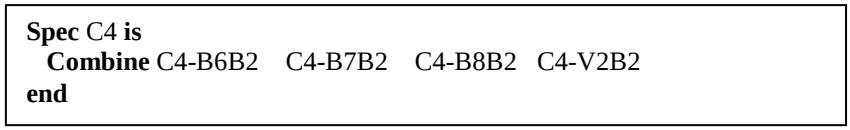


Figure 10.21: Module de spécification du commutateur C4.

VII- Spécification de l'intersection des voies ferrées

Etant donné un train qui circule sur l'un des tronçons B5 B6 B11 B12, l'intersection permet le passage de ce train vers l'un ou l'autres de ces tronçons.

Nous avons donc le passages (dans les deux sens de circulation) suivants :

- B5 et B11
- B5 et B12
- B11 et B6
- B12 et B6

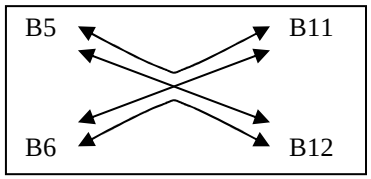


Figure 10.22: Schéma de l'intersection Cross.

Spec Cross-B5B11 **is**
Rename Move **by**
Place p2→B5
Place p1 →B11
end

Spec Cross-B5B12 **is**
Rename Move **by**
Place p2→B5
Place p1 →B12
end

Spec Cross-B11B6 **is**
Rename Move **by**
Place p2→B11
Place p1 →B6
end

Spec Cross-B12B6 **is**
Rename Move **by**
Place p2→B12
Place p1 →B6
end

Figure 10.23: Modules de spécification
des mouvements possibles dans l'intersection Cross.

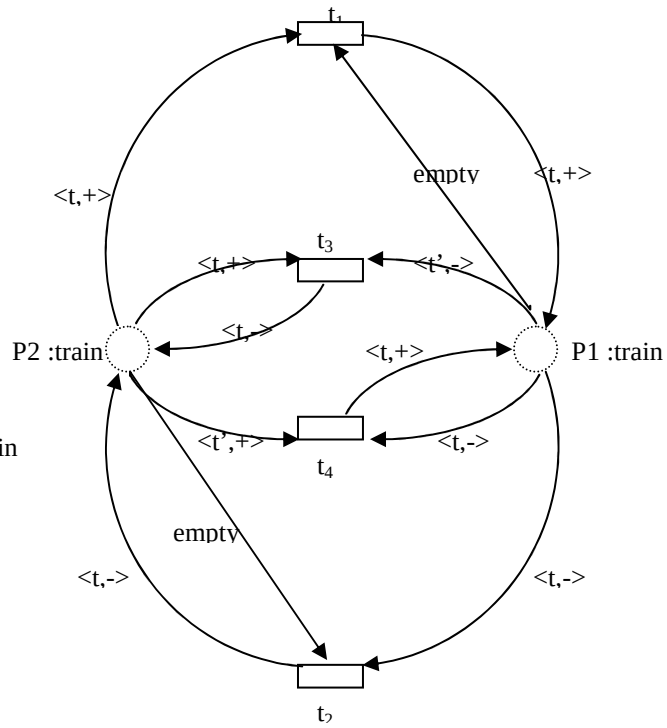
Spec Cross **is**
Combine Cross-B5B11 Cross-B5B12 Cross-B11B6 Cross-B12B6
end

Figure 10.24: Modules de spécification de l'intersection Cross.

VIII- Spécification plate associée au système global

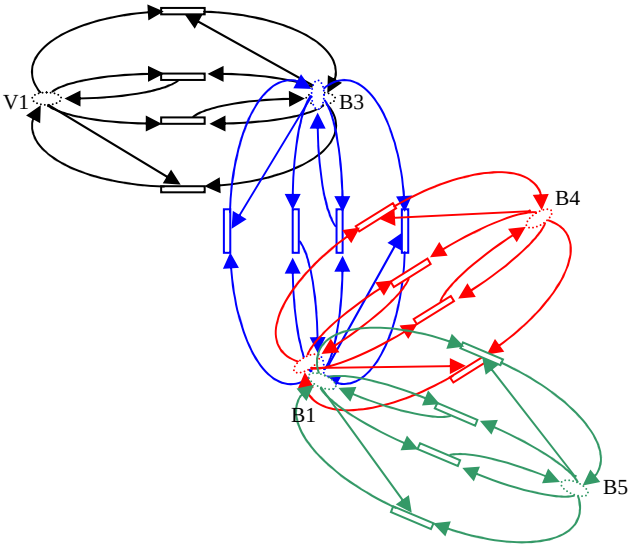
1- Module plat associé au module *Move*

Spec TRAIN **is**
Sorts id sens train
Ops $t_1 :\rightarrow$ id
$t_2 :\rightarrow$ id
$t_3 :\rightarrow$ id
….
$t_n :\rightarrow$ id
+ :→ sens
-:→ sens
<_,_> : id sens → train
end

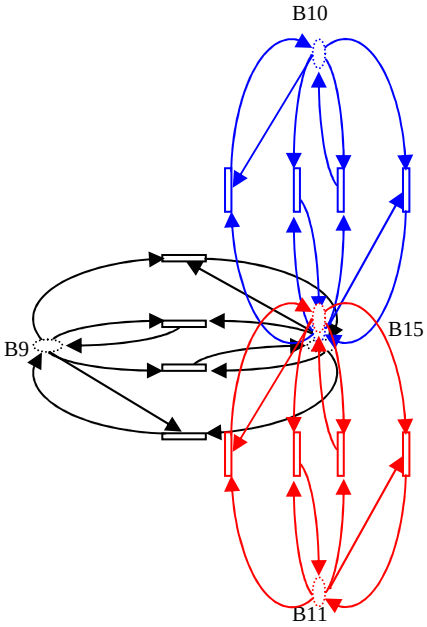


2- Modules plats associés aux commutateurs

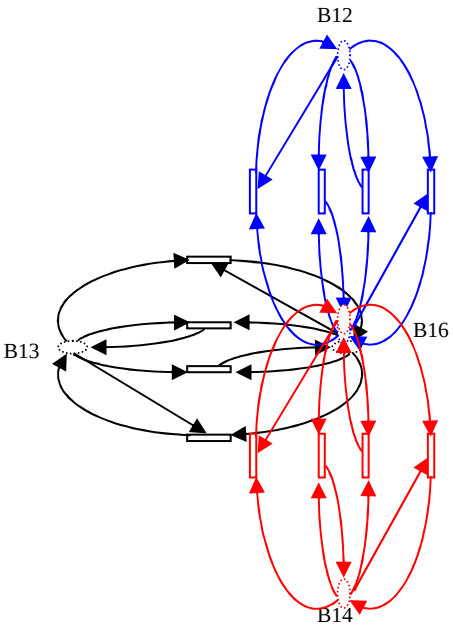
En faisant abstraction des spécifications algébriques et des inscriptions des arcs, le réseau du module plat associé à chaque commutateur est représenté par les figures suivantes.



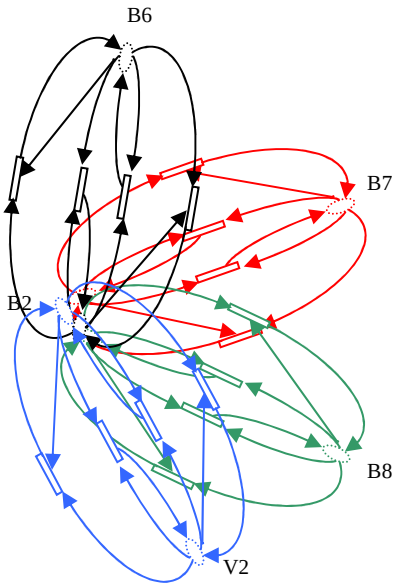
Commutateur C1



Commutateur C2

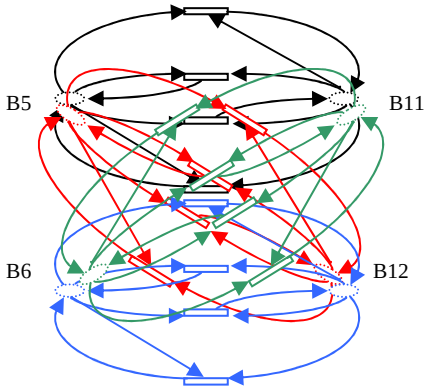


Commutateur C3

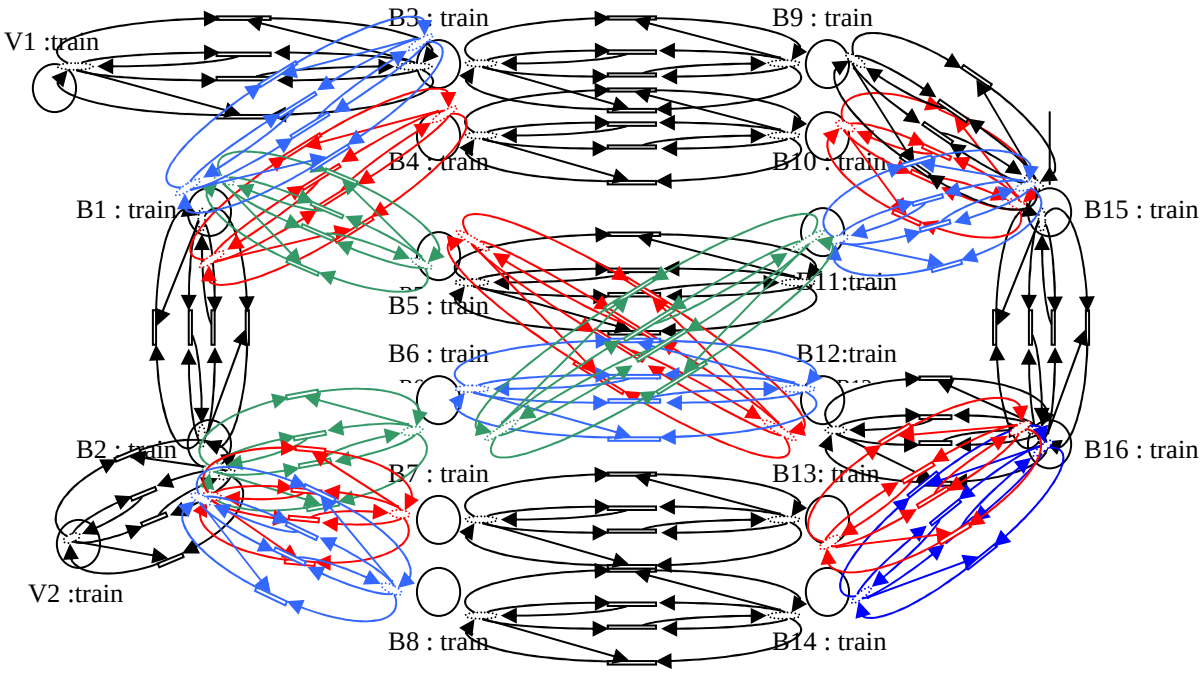


Commutateur C4

3- Module plat associé à l'intersection CROSS



4- Module plat associé au système global



IX- Conclusion

L'étude de cas décrite dans ce chapitre, démontre bien que le langage de spécification CIRTA est tout à fait utilisable pour spécifier des systèmes réalistes non triviaux. Par ailleurs, le développement modulaire des spécifications aide à une meilleure compréhension du fonctionnement du système global.
D'autre études de cas mettant en relief d'autres aspects du langage CIRTA (notion de contraintes temporelles, contraintes contextuelles, paramétrisation) sont présentées dans [ZM-79][ZBB-05][ZBB-06].

Conclusion

Conclusion

Quoique bien ambitieux comme travail, la définition d'un langage de spécification basé sur les ECATNets a atteint un stade assez avancé dans cette thèse. Le travail réalisé ici est l'aboutissement de longues réflexions et le fruit de la convergence de plusieurs domaines (spécifications algébriques, réseaux de Petri, langages de spécification…) en informatique.

En définissant un langage de spécification basé sur les ECATNets, ce travail contribue à améliorer la productivité et l'efficacité dans l'ensemble des opérations entrant dans le cycle de vie du logiciel de la spécification à la maintenance; et favorise ainsi l'émergence d'une industrie de produits de génie logiciel.

• Contributions

Le travail réalisé dans cette thèse présente un apport essentiel qui peut être vu à deux niveaux : d'une part au niveau des ECATNets en tant que formalisme de base du langage CIRTA ; et d'autre part au niveau de la définition même du langage de spécification CIRTA.

➢ Extension du Formalisme de base

Une attention particulière a été accordée à l'extension des ECATNets par de nouveaux concepts permettant d'une part la spécification d'une plus grande variété de systèmes, et offrant d'autre part une plateforme plus puissante qui favorise l'évolution future du langage de spécification CIRTA. Ces différentes extensions peuvent se résumer aux points suivants :

* Définition de la notion de *contraintes contextuelles positives* qui permet de tester la *présence* de certains jetons dans une place :
Dans les travaux antérieurs sur les ECATNets, le concept de *condition d'entrée* (IC) permet de tester la présence de jetons dans une place en procédant (sémantiquement) à une *destruction puis une re-création* de ces jetons (interdisant ainsi certains tests parallèles de contraintes contextuelles). La sémantique des contraintes contextuelles positives présentée dans cette thèse reflète plus fidèlement l'aspect d'accès *concurrent* à un *même jeton* pour des tests contextuels.

* Définition de la notion de *contraintes contextuelles négatives* qui permet de tester *l'absence* de certains (ou de tous les) jetons dans une place :
Cette notion inspirée des réseaux de Petri à arcs inhibiteurs a une sémantique souvent présentée dans d'autres cadres sémantiques que la logique de réécriture, et l'approche proposée dans cette thèse constitue à notre connaissance l'une des premières tentatives de spécification des tels types de réseaux dans le cadre la logique de réécriture.

* Extension de la fonction de destruction des jetons *DT* en permettant de procéder à la destruction de *tous les jetons* d'une place (*DT(p,t)*= ∀ *)* par une action indivisible.

* Définition d'une forme générale *unique* pour les règles de réécriture associées aux transitions dans la définition de la sémantique d'un ECATNet :
  Dans le cadre de la sémantique des ECATNets, l'apport principal réside dans la définition d'une *forme générale unique pour toutes les règles de réécriture* associées aux transitions. En effet, dans les premiers travaux sur les ECATNets, la forme de la règle de réécriture associée à une transition dépend de la relation entre *IC*, *DT* et *CT*. De plus dans certains cas, on associe à une transition deux règles de réécriture. La définition dans cette thèse, d'une forme générale *unique et homogène* des règles de réécriture favorise une meilleure compréhension, une plus grande souplesse dans les processus de preuve, et un prototypage plus simple.

* Définition des *ECATNets temporels* qui généralisent les ECATNets en permettant la prise en compte du facteur temps comme représentant une composante explicite du système spécifié. La sémantique des ECATNets temporels a été définie dans le cadre de la logique de réécriture.

- **Définition du langage CIRTA**

Les extensions que nous avons apportées au ECATNets augmentent leur pouvoir d'expression mais ne permettent pas un développement progressif et incrémental des spécifications. La définition du langage CIRTA contribue à favoriser une démarche de développement modulaire des spécifications. Dans ce contexte, l'apport de cette thèse se résume aux points suivants :

* Définition du concept de *module d'ECATNet :*
  L'avantage de ce concept est que l'on peut spécifier tous les composants d'un système de façon indépendante et exprimer leurs interactions. Plusieurs opérations de compositions peuvent être ensuite appliquées pour permettre l'obtention d'un module décrivant le système dans son ensemble; et le module ainsi obtenu pourra être réutilisé comme composant d'un autre système.
  Les modules d'ECATNets sont caractérisés par une *structure hiérarchique* et une *communication variée* entre eux. Cette communication définie par une interface, peut être *synchrone, asynchrone* ou les *deux à la fois*. Ce qui offre une certaine liberté lors du processus de spécification où l'utilisateur n'est pas limité dans la définition des modules à un seul type d'interface.
  La définition du concept de *module d'ECATNet* n'est pas une extension purement syntaxique. La sémantique formelle d'un module est définie par une théorie de réécriture.

* Définition d'un ensemble de *primitives de structuration* des spécifications :
Ces primitives sont déduites du domaine des spécifications algébriques et du domaine des réseaux de Petri, et permettent l'élaboration progressive des spécifications complexes:

- La primitive *d'enrichissement* est basée sur la substitution et la duplication et supporte ainsi à la fois le développement ascendant et descendant des spécifications. Nous pouvons noter en particulier l'introduction de la notion *noeud de substitution* qui généralise les notions de places de substitution et transition de substitution connues dans le domaine des réseaux de Petri.

- La primitive *de composition* permet de composer deux ou plusieurs modules. Cette composition se distingue de l'opération de composition des réseaux de Petri par le fait que les réseaux des modules d'ECATNet composés peuvent ne pas avoir de nœuds (places et/ou transitions) communs.

- La primitive de *renommage* contribue à faciliter la création de modules isomorphes quand des changements de notations sont nécessaires.

- La primitive de *contrôle de visibilité* permet de faire abstraction de certains éléments de la spécification en les rendant non visibles et contribue à faciliter le mécanisme d'encapsulation des modules.

- Le mécanisme de *paramétrisation* qui offre la possibilité de spécifier des systèmes génériques. Deux approches de paramétrisation ont été introduites. La première approche dite *paramétrisation statique* est inspirée principalement des travaux sur la paramétrisation des spécifications algébriques et permet de spécifier des systèmes ayant la même structure de contrôle mais qui manipulent des données "différentes". La seconde approche dite *paramétrisation dynamique* permet de décrire des systèmes ayant des comportements visibles (ou observationnels) comparables. Le mécanisme de paramétrisation proposé dans cette thèse est une généralisation de la méthode de paramétrisation d'un réseau de Petri par un autre réseau que nous avons présenté dans [ZEG-96][ZM-97].

- Le mécanisme d'*instanciation* des modules paramétrés qui permet de générer des modules de spécifications. Cette instanciation présente la caractéristique de pouvoir être de différents types : elle peut être d'une part *partielle ou total*, et d'autre part elle peut être *formelle ou effective.*

* Définition de *la syntaxe du langage* qui présente une souplesse d'utilisation caractérisée par:

- Une combinaison de la représentation *textuelle* et une représentation *graphique* (ou le graphe peut être décrit sous forme de texte)

-Une *liberté dans le choix des formes des opérations* définies par l'utilisateur: Chacune de ces opérations peut être préfixée, postfixée, mixfixée, distribuée, anonyme, surchargée,..etc.

- La possibilité de la *représentation multiple* d'une même place ou une même transition dans un même module. Ceci permet de ne pas avoir des arcs qui vont "d'un bout du réseau à l'autre".

- La possibilité de définition *d'abstractions syntaxiques* qui permettent d'éviter la manipulation de réseaux ayant des inscriptions (IC,DT,CT, marquage) complexes. A chaque inscription est alors associé un nom (l'abstraction syntaxique de cette inscription) qui rappelle la déclaration de constantes dans les langages de programmation.

* Définition de la *sémantique formelle du langage* CIRTA :
A chaque spécification CIRTA est associée une classe de modèles. La sémantique des différentes primitives est définie par des foncteurs ou des classes de foncteurs entre ces classes de modèles. Cette sémantique a été décrite au niveau modèle pour être la plus générale possible et est destinée à supporter l'évolution des ECATNets. En effet, une extension future des ECATNets modifie en général sa théorie de réécriture et par conséquent sa classe des modèles mais ne change pas la sémantique des primitives qui est définies sur les classes de modèles.

* *Application du langage* à la spécification d'un système non trivial :
Une étude de cas présentée dans cette thèse montre bien que le langage de spécification CIRTA est tout à fait utilisable pour spécifier des systèmes réalistes non triviaux. Par ailleurs, le développement modulaire des spécifications aide à une meilleure compréhension du fonctionnement du système global

- **Perspectives**

Le langage ainsi défini, constitue une première version d'un langage qui ne demande qu'à être étoffé aussi bien sur le plan théorique que pratique. Plusieurs voies différentes paraissent dignes d'intérêt:

En premier lieu, on pourrait étendre l'ensemble des primitives du langage: les primitives présentées dans cette thèse permettent de construire une spécification relativement complexe à partir de modules plus simples. Il peut s'avérer utile de définir des opérations qui *dérivent* un module de spécification simple à partir d'une spécification plus complexe.

Dans le futur, il nous semble indispensable de disposer d'un environnement de spécification pour les ECATNets. Cet environnement doit intégrer entre autres, un éditeur graphique, une bibliothèque de spécifications, un prouveur de théorèmes et un générateur d'analyseurs syntaxiques. Notons que d'autres travaux au niveau du laboratoire LIRE contribuent (ou ont contribué) au développement de certains de ces outils.

Il serait intéressant d'élaborer des outils d'analyse des propriétés (statiques et dynamiques) des spécifications permettant de vérifier qu'un module est un paramètre effectif valide (vérifie les contraintes statiques et dynamiques décrites par un module paramètre formel). Des travaux sur la réduction des réseaux de Petri et leur abstraction [BER-83][DES-91] peuvent faciliter de telles vérifications.

Il serait également intéressent que la modularité des spécifications CIRTA se répercute au niveau vérification. Grâce à la définition de morphismes de spécifications, les propriétés prouvées sur chaque module peuvent être transférées le long des morphismes jusqu'à la spécification globale, ce qui permet d'effectuer des preuves modulaires des propriétés du système spécifié.

Annexe-1

Preuves de Théorèmes

Théorème 8.1

Si $\mathcal{E}$ est un ECATNet simple alors :

$\forall [s][s'] \in State_{\mathcal{T}_{simp}}(\mathcal{E})$ *on a* $\mathcal{T}_{simp} \vdash [s] \rightarrow [s'] \Rightarrow \mathcal{T}_{cap} \vdash [s] \rightarrow [s']$

où : $State_{\mathcal{T}_{simp}}(\mathcal{E})$ est l'ensemble des états de $\mathcal{E}$ dans la théorie $\mathcal{T}_{simp}$

Preuve

La preuve consiste à démontrer que si un séquent $[s] \rightarrow [s']$ est prouvable dans $\mathcal{T}_{simp}$ alors il est prouvable dans $\mathcal{T}_{cap}$. Sachant que toute preuve dans $\mathcal{T}_{simp}$ se base sur l'utilisation des équations, des règles de déduction et des règles de réécriture de $\mathcal{T}_{simp}$, on montre que chaque équations, chaque règle de déduction et chaque règle de réécriture de $\mathcal{T}_{simp}$ est prouvable dans $\mathcal{T}_{cap}$.

(a) **Les équations** : $\mathcal{E}_{simp} \subseteq \mathcal{E}_{cap}$ par définition.

(b) **Les règles de déduction** : les règles de déduction (i) (ii) (iii) et (iv) de $\mathcal{T}_{simp}$ sont identiques à celles de $\mathcal{T}_{cap}$. Il suffit de prouver dans $\mathcal{T}_{cap}$ la validité des règles de composition et de décomposition (v) et (vi) de $\mathcal{T}_{simp}$.

* Règle de décomposition de $\mathcal{T}_{simp}$ $\dfrac{[m_1 \oplus m_2 \oplus ... \oplus m_n] \rightarrow [m] \quad [s] \rightarrow [< p, m >]}{[s] \rightarrow [< p, m_1 > \otimes < p, m_2 > \otimes ... \otimes < p, m_n >]}$

(1) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m >]$ hypothèse

(2) $\mathcal{T}_{cap} \vdash [m_1 \oplus m_2 \oplus ... \oplus m_n] \rightarrow [m]$ hypothèse

(3) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m, \infty >]$ règle de déduction *Ajout-capacité* de $\mathcal{T}_{cap}$ sur (1)

(4) $\mathcal{T}_{cap} \vdash [\infty + \infty + ... + \infty] \rightarrow [\infty]$ équation sur ∞ dans $\mathcal{T}_{cap}$

(5) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1, \infty > \otimes < p, m_2, \infty > \otimes ... \otimes < p, m_n, \infty >]$
règle de déduction *Décomposition* de $\mathcal{T}_{cap}$ sur (2,3,4)

(6) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1 > \otimes < p, m_2 > \otimes ... \otimes < p, m_n >]$
règle de déduction *Suppression-capacité* de $\mathcal{T}_{cap}$ sur (5)

* Règle de composition de $\mathcal{T}_{simp}$ $\dfrac{[s] \rightarrow [< p, m_1 > \otimes < p, m_2 > \otimes ... \otimes < p, m_n >]}{[s] \rightarrow [< p, m_1 \oplus m_2 \oplus ... \oplus m_n >]}$

(1) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1 > \otimes < p, m_2 > \otimes ... \otimes < p, m_n >]$ hypothèse

(2) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1, \infty > \otimes < p, m_2, \infty > \otimes ... \otimes < p, m_n, \infty >]$
règle de déduction *Ajout-capacité* de $\mathcal{T}_{simp}$ sur (1) *n* fois

(3) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1 \oplus m_2 \oplus .. \oplus m_n, \infty + \infty + .. + \infty >]$ règle de déduction *Composition* de $\mathcal{T}_{cap}$ sur (2)

(4) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1 \oplus m_2 \oplus .. \oplus m_n, \infty >]$ équation de ∞ dans $\mathcal{T}_{cap}$ sur (3)

(5) $\mathcal{T}_{cap} \vdash [s] \rightarrow [< p, m_1 \oplus m_2 \oplus ... \oplus m_n >]$ règle de déduction *Suppression-capacité* de $\mathcal{T}_{cap}$ sur (4)

(c) **Règle de réécriture**

Soit $L \to R$ une règle de réécriture de $\mathcal{T}_{simp}$ où :

$$L = \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t) >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t) >) \end{pmatrix} \quad \text{et} \quad R = \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t) >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t) >) \end{pmatrix}$$

On montre que $\mathcal{T}_{cap} \vdash [L] \to [R]$ en faisant abstraction de la condition de transition *TC(t)* qui est identique dans $\mathcal{T}_{simp}$ et $\mathcal{T}_{cap}$

(1). $\mathcal{T}_{cap} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t) >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t) >) \end{pmatrix} \to \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t) >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t) >) \end{pmatrix}$

règle de déduction *Réflexivité* de $\mathcal{T}_{cap}$.

(2). $\mathcal{T}_{cap} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), \infty >) \end{pmatrix} \to \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), \infty >) \end{pmatrix}$

règle de déduction *Ajout-capacité* de $\mathcal{T}_{cap}$ *2n* fois sur (1).

(3). $\mathcal{T}_{cap} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), \infty >) \end{pmatrix} \to \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \lvert CT(p_i,t) \rvert >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), \lvert CT(p_i,t) \rvert - \lvert DT(p_i,t) \rvert >) \end{pmatrix} \otimes \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, \varnothing, \infty >) \end{pmatrix}$

règle de déduction *Décomposition* de $\mathcal{T}_{cap}$ *n* fois sur (2)

(4). $\mathcal{T}_{cap} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), \infty >) \end{pmatrix} \to \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, \lvert DT(p_i,t) \rvert >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t), 0 >) \end{pmatrix} \otimes \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, \varnothing, \infty >) \end{pmatrix}$

règle de réécriture de $\mathcal{T}_{cap}$ sur (3)

(5). $\mathcal{T}_{cap} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), \infty >) \end{pmatrix} \to \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, \infty >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t), \infty >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t), \infty >) \end{pmatrix}$

règle de déduction *Composition* de $\mathcal{T}_{cap}$ sur (4)

(6). $\mathcal{T}_{cap} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t) >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t) >) \end{pmatrix} \to \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t) >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t) >) \end{pmatrix}$

règle de déduction *Suppression-capacité* de $\mathcal{T}_{cap}$ *2n* fois sur (5).

Théorème 8.2

Soit $\mathcal{E}$ un ECATNet à capacité tel que $dom(IC) = \varnothing$ *et* $\forall (p,t) \in dom(DT)\ DT(p,t) \neq \forall$ alors :

$$\forall [s][s'] \in State_{\mathcal{T}_{cap}}(\mathcal{E}) \quad on\ a \quad \mathcal{T}_{cap} \vdash [s] \rightarrow [s'] \ \Rightarrow\ \mathcal{T}_{cont} \vdash [s] \rightarrow [s']$$

où : $State_{\mathcal{T}_{cap}}(\mathcal{E})$ est l'ensemble des états de $\mathcal{E}$ dans la théorie $\mathcal{T}_{cap}$

Preuve

La preuve consiste à démontrer que si un séquent $[s] \rightarrow [s']$ est prouvable dans $\mathcal{T}_{cap}$ alors il est prouvable dans $\mathcal{T}_{cont}$ (sachant que $[s]$ et $[s']$ sont des états d'un ECATNet à capacité sans contraintes contextuelles).
Comme dans le cas du théorème précédent, la preuve revient à prouver la validité des équations, des règles de déduction et des règles de réécriture de la théorie $\mathcal{T}_{cap}$ dans la théorie $\mathcal{T}_{cont}$. Pour cela nous avons besoin de définir les opérations suivantes.

*Définition-1 : Opération fc («**f**orget **c**ontext »)*

La fonction *fc* est définie par :

op $fc(\_) : state \rightarrow state$

vars $s_1\ s_2 : state\ \ c : nat_\infty$

$m : m_{\sigma(p)}\ \ M : gm_{\sigma(p)}$ {pour chaque place p}

eqs

$$fc(\varnothing) = \varnothing$$

$$fc(s_1 \otimes s_2) = fc(s_1) \otimes fc(s_2)$$

$$\left.\begin{array}{l} fc(< p,m >) =< p,m > \\ fc(< p,m,c >) =< p,m,c > \\ fc(< p,m,c,M >) =< p,m,c > \\ fc([\![p|n]\!]) = \varnothing \end{array}\right\} \{ pour\ chaque\ place\ p \}$$

L'opération *fc* fait abstraction des informations relatives au contexte. Elle associe à chaque état de $State_{\mathcal{T}_{cont}}$ l'état correspondant en faisant abstraction d'une part, du marquage global de chaque place ($fc(< p,m,c,M >) =< p,m,c >$), et d'autre part du nombre de copies du marquage globale d'un place ($fc([\![p|n]\!]) = \varnothing$).

*Définition-2 : Opération gm (« **g**et global **m**arquing »)*

On définit l'opération *gm* comme suit :

op $gm(\_) : state \rightarrow state$

vars $s_1 \;\; s_2 : state \;\; c : nat_\infty$

$m : m_{\sigma(p)} \;\; M : gm_{\sigma(p)}$ {pour chaque place p}

eqs

$gm(\varnothing) = \varnothing$

$gm(s_1 \otimes s_2) = gm(s_1) \otimes gm(s_2)$

$gm(< p, m >) =< p, m, \infty, ? >$

$gm(< p, m, c >) =< p, m, c, ? >$

$gm(< p, m, c, M >) =< p, m, c, M >$

$gm([\![p \,|\, n]\!]) = [\![p \,|\, n]\!]$

{ pour chaque place p }

Le rôle de l'opération *gm* est d'associer à chaque état un état dans lequel le marquage global des places est spécifié.

*Définition-3 : Opération gc (« **g**et **c**ounters »)*

L'opération gc_p ajoute à un état donné s le nombre d'occurrences des termes de la forme $< p_i, m, c, M >$ dans s. Elle est définie par :

op $gc_p(\_) : state \rightarrow state$ $\{$*pour chaque place p*$\}$

$compter_p(\_) : state \rightarrow nat$ $\{$*pour chaque place p*$\}$

vars $s \;\; s_1 \;\; s_2 : state \;\; n : nat \;\; c : nat_\infty$

$m : m_{\sigma(p)} \;\; M : gm_{\sigma(p)}$ {pour chaque place p}

eqs

$gc_p(s) = s \;\; if \;\; compter_p(s) = 0$

$gc_p(s) = s \otimes [\![p \,|\, n]\!] \;\; if \;\; (compter_p(s) = n) \wedge (n \neq 0)$

$compter_p(\varnothing) = 0$

$compter_p(s_1 \otimes s_2) = compter_p(s_1) + compter_p(s_2)$

$compter_p(< p', m >) = 0$

$compter_p(< p', m, c >) = 0$

$compter_p([\![p' \,|\, n]\!]) = 0$

$compter_p(< p, m, c, M >) = 1$

$compter_p(< p', m, c, M >) = 0 \;\; if \;\; p \neq p'$

(a) **Les équations** : $\mathcal{E}_{cap} \subseteq \mathcal{E}_{cont}$ par définition.

(b) **Les règles de déduction** : les règles de déduction (i) (ii) (iii) et (iv) de $\mathcal{T}_{cap}$ sont identiques à celles de $\mathcal{T}_{cont}$. Il suffit de prouver dans $\mathcal{T}_{cont}$ la validité des règles de composition et de décomposition (v) et (vi) de $\mathcal{T}_{cap}$.

* Règle de décomposition de $\mathcal{T}_{cap}$ $\dfrac{[m_1 \oplus m_2 \oplus ... \oplus m_n] \to [m] \quad [c_1 + c_2 + ... + c_n] \to [c] \quad [s] \to [< p, m, c >]}{[s] \to [\otimes_{i=1..n} < p, m_i, c_i >]}$

(1) $\mathcal{T}_{cont} \vdash [m_1 \oplus m_2 \oplus ... \oplus m_n] \to [m]$ hypothèse

(2) $\mathcal{T}_{cont} \vdash [c_1 + c_2 + ... + c_n] \to [c]$ hypothèse

(3) $\mathcal{T}_{cont} \vdash [s] \to [< p, m, c >]$ hypothèse

(4) $\mathcal{T}_{cont} \vdash [s] \to [< p, m_i, c, m_i > \otimes [\![p \,|1]\!]]$ règle de déduction *Ajout-marquage* de $\mathcal{T}_{cont}$ sur (3)

(5) $\mathcal{T}_{cont} \vdash [s] \to [< p, m_1, c_1, m > \otimes < p, m_2, c_2, m > \otimes ....\otimes < p, m_n, c_n, m > \otimes [\![p \,|n]\!]]$
règle de déduction *Décomposition* de $\mathcal{T}_{cont}$ sur (1,2,4)

(6) $\mathcal{T}_{cont} \vdash fc([s]) \to fc([< p, m_1, c_1, m > \otimes < p, m_2, c_2, m > \otimes ....\otimes < p, m_n, c_n, m > \otimes [\![p \,|n]\!]])$
règle de déduction *Congruence* de $\mathcal{T}_{cont}$ sur (5)

(7) $\mathcal{T}_{cont} \vdash fc([s]) \to [fc(< p, m_1, c_1, m >) \otimes fc(< p, m_2, c_2, m >) \otimes .... \otimes fc(< p, m_n, c_n, m >) \otimes fc([\![p \,|n]\!])]$
distributivité de fc par rapport à $\otimes$ sur (6)

(8) $\mathcal{T}_{cont} \vdash fc([s]) \to [< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ...\otimes < p, m_n, c_n > \otimes \varnothing]$ par définition de fc sur (7)

(9) $\mathcal{T}_{cont} \vdash [s] \to [< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ...\otimes < p, m_n, c_n >]$ $\quad fc([s]) = [s]$ car $[s] \in State_{cap}(\mathcal{E})$

* Règle de composition de $\mathcal{T}_{cap}$ $\dfrac{[s] \to [< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ...\otimes < p, m_n, c_n >]}{[s] \to [< p, m_1 \oplus m_2 \oplus ... \oplus m_n, c_1 + c_2 + ... + c_n >]}$

(1) $\mathcal{T}_{cont} \vdash [s] \to [< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ...\otimes < p, m_n, c_n >]$ hypothèse

(2) $\mathcal{T}_{cont} \vdash gm([s]) \to gm([< p, m_1, c_1 > \otimes < p, m_2, c_2 > \otimes ...\otimes < p, m_n, c_n >])$
règle de déduction *Congruence* de $\mathcal{T}_{cont}$ sur (1)

(3) $\mathcal{T}_{cont} \vdash gm([s]) \to [gm(< p, m_1, c_1 >) \otimes gm(< p, m_2, c_2 >) \otimes ... \otimes gm(< p, m_n, c_n >)]$
distributivité de *gm* sur $\otimes$

(4) $\mathcal{T}_{cont} \vdash gm([s]) \to [< p, m_1, c_1, ? > \otimes < p, m_2, c_2, ? > \otimes ...\otimes < p, m_n, c_n, ? >]$ par définition de *gm* sur (3)

(5) $\mathcal{T}_{cont} \vdash gc_p(gm([s])) \to gc_p([< p, m_1, c_1, ? > \otimes < p, m_2, c_2, ? > \otimes ...\otimes < p, m_n, c_n, ? >])$
règle de déduction *Congruence* de $\mathcal{T}_{cont}$ sur (4)

(6) $\mathcal{T}_{cont} \vdash gc_p(gm([s])) \to [< p, m_1, c_1, ? > \otimes < p, m_2, c_2, ? > \otimes ...\otimes < p, m_n, c_n, ? > \otimes [\![p \,|n]\!]]$
par définition de gc_p sur (5)

(7) $\mathcal{T}_{cont} \vdash gc_p(gm([s])) \to [< p, m_1 \oplus m_2 \oplus ... \oplus m_n, c_1 + c_2 + ... + c_n, ? > \otimes [\![p \,|1]\!]]$
règle de déduction *Composition* de $\mathcal{T}_{cont}$ sur (6)

(8) $\mathcal{T}_{cont} \vdash fc(gc_p(gm([s]))) \to fc([< p, m_1 \oplus m_2 \oplus ... \oplus m_n, c_1 + c_2 + ... + c_n, ? > \otimes [\![p \,|1]\!]])$
règle de déduction *Congruence* de $\mathcal{T}_{cont}$ sur (7)

(9) $\mathcal{T}_{cont} \vdash [s] \to [< p, m_1 \oplus m_2 \oplus ... \oplus m_n, c_1 + c_2 + ... + c_n >]$ par définition de *fc* sur (8)

(c) **Règle de réécriture**

Soit $L \rightarrow R$ une règle de réécriture de $\mathcal{T}_{cap}$ où

$$L = \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix} \quad \text{et} \quad R = \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, |DT(p_i,t)| >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t), 0 >) \end{pmatrix}$$

On montre que $\mathcal{T}_{cont} \vdash [L] \rightarrow [R]$ en faisant abstraction de la condition de transition *TC(t)* qui est identique dans $\mathcal{T}_{cap}$ et $\mathcal{T}_{cont}$

1) $\mathcal{T}_{cont} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}$

règle de déduction *Réflexivité* dans $\mathcal{T}_{cont}$

2) $\mathcal{T}_{cont} \vdash gm\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}\right) \rightarrow gm\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}\right)$

règle de déduction *Congruence* de $\mathcal{T}_{cont}$ sur (1)

3) $\mathcal{T}_{cont} \vdash gm\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}\right) \rightarrow \begin{pmatrix} (\otimes_{i=1..n1} gm < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} gm < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} gm < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}$

distributivité de *gm* par rapport à $\otimes$ sur (2)

4) $\mathcal{T}_{cont} \vdash gm\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}\right) \rightarrow \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0, ? >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)|, ? >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)|, ? >) \end{pmatrix}$

par définition de *gm* sur (3)

5) $\mathcal{T}_{cont} \vdash gm\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}\right) \rightarrow \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, |DT(p_i,t)|, ? >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t), 0, ? >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t), 0, ? >) \end{pmatrix}$

règle de réécriture de $\mathcal{T}_{cont}$ sur (4)

6) $\mathcal{T}_{cont} \vdash fc\left(gm\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix}\right)\right) \rightarrow fc\left(\begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, |DT(p_i,t)|, ? >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t), 0, ? >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t), 0, ? >) \end{pmatrix}\right)$

règle de déduction *Congruence* de $\mathcal{T}_{cont}$ sur (5)

7) $\mathrm{T}_{cont} \vdash \begin{pmatrix} (\otimes_{i=1..n1} < p_i, DT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, \varnothing, |CT(p_i,t)| >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, DT(p_i,t), |CT(p_i,t)| - |DT(p_i,t)| >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n1} < p_i, \varnothing, |DT(p_i,t)|_{\sigma(p_i)} >) \otimes \\ (\otimes_{i=n1+1..n2} < p_i, CT(p_i,t), 0 >) \otimes \\ (\otimes_{i=n2+1..n} < p_i, CT(p_i,t), 0 >) \end{pmatrix}$

par définition de *fc* sur (6).

Théorème 8.3

Si $\mathcal{E}$ un ECATNet contextuel sans contraintes temporelles alors :

$\forall [s][s'] \in State_{\mathcal{T}_{cont}}(\mathcal{E})$ *on a* $\quad \mathcal{T}_{cont} \vdash [s] \rightarrow [s'] \quad \Rightarrow \quad \mathcal{T}_{temp} \vdash [s] \rightarrow [s']$

où : $State_{\mathcal{T}_{cont}}(\mathcal{E})$ est l'ensemble des états de $\mathcal{E}$ dans la théorie $\mathcal{T}_{cont}$

Preuve :

La preuve consiste à démontrer que si un séquent $[s] \rightarrow [s']$ est prouvable dans $\mathcal{T}_{cont}$ alors il est prouvable dans $\mathcal{T}_{temp}$ (sachant que $[s]$ et $[s']$ sont des états d'un ECATNet contextuel sans contraintes temporelles).
Comme dans le cas du théorème précédent, la preuve revient à prouver la validité des équations, des règles de déduction et des règles de réécriture de la théorie $\mathcal{T}_{cont}$ dans la théorie $\mathcal{T}_{temp}$. Pour cela nous avons besoin de définir les opérations suivantes :

Définition-4 : Opération *gh* (« ***g**et **h**orloge* »)

op $gh(\_)$ **:** *state* $\rightarrow$ *state*
var s **:** *state*

eqs $gh(s) = s \otimes < H = 0 >$

Définition-5 : Opération *fh* (« ***f**orget **h**orloge* »)

op $fh(\_)$ **:** *state* $\rightarrow$ *state*
vars s_1 s_2 **:** *state* n **:** *nat* c **:** nat_∞
m **:** $m_{\sigma(p)}$ M **:** $gm_{\sigma(p)}$ {pour chaque place p}
eqs
$fh(\varnothing) = \varnothing$
$fh(s_1 \otimes s_2) = fh(s_1) \otimes fh(s_2)$
$fh(< H = n >) = \varnothing$
$fh(< p, m >) =< p, m >$
$fh(< p, m, c >) =< p, m, c >$
$fh(< p, m, c, M >) =< p, m, c >$
$fh([\![p|n]\!]) = [\![p|n]\!]$
} *{ pour chaque place p }*

(a) **Les équations** : $\mathcal{E}_{cont} \subseteq \mathcal{E}_{temp}$ par définition.

(b) **Les règles de déduction** : les règles de déduction de $\mathcal{T}_{cont}$ sont incluses dans l'ensemble des règles de déduction de $\mathcal{T}_{temp}$ par définition.

(c) <u>Les règles de réécriture</u> :

Soit $L \rightarrow R$ une règle de réécriture de $\mathcal{T}_{cont}$ où :

$$L = \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \text{ et } R = \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i, t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i, t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i, t), 0, ? >) \end{pmatrix}$$

On montre que $\mathcal{T}_{temp} \vdash [L] \rightarrow [R]$ en faisant abstraction de la condition $TC(t) \wedge (\wedge_{i=1..n_1} ic_i) \wedge (\wedge_{i=n_2+1..n} ic_i)$ qui est identique dans $\mathcal{T}_{cont}$ et $\mathcal{T}_{temp}$.

1)

$$\mathcal{T}_{temp} \vdash \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix}$$

règle de déduction *Réflexivité* de $\mathcal{T}_{temp}$

2)

$$\mathcal{T}_{temp} \vdash gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix}$$

règle de déduction *Congruence* de $\mathcal{T}_{temp}$ sur (1).

3)

$$\mathcal{T}_{temp} \vdash gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \otimes \\ < H = 0 > \end{pmatrix}$$

par définition de gh sur (2).

4)

$$\mathcal{T}_{temp} \vdash gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \otimes \\ [H = 0] \otimes [\![H | 1]\!] \end{pmatrix}$$

règle de déduction *Copier-horloge* de $\mathcal{T}_{temp}$ sur (3).

5)

$$\mathcal{T}_{temp} \vdash gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i, t, h >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i, t, h >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i, t, h >) \otimes \\ [H = 0] \otimes [\![H | 1]\!] \end{pmatrix}$$

règle de réécriture *sens.t* de $\mathcal{T}_{temp}$ sur (4).

6) $$\mathcal{T}_{temp} \vdash gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i, t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i, t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i, t), 0, ? >) \otimes \\ [H = 0] \otimes [\![H | 1]\!] \end{pmatrix}$$

règle de réécriture *t* de $\mathcal{T}_{temp}$ sur (5).

7) $$\mathcal{T}_{temp} \vdash gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i, t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i, t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i, t), 0, ? >) \otimes \\ < H = 0 > \end{pmatrix}$$

règle de déduction *Fusionner-copies-horloge* de $\mathcal{T}_{temp}$ sur (6).

8) $$\mathcal{T}_{temp} \vdash fh \left(gh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \right) \rightarrow fh \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i, t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i, t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i, t), 0, ? >) \otimes \\ < H = 0 > \end{pmatrix}$$

règle de déduction *Congruence* de $\mathcal{T}_{temp}$ sur (7).

9) $$\mathcal{T}_{temp} \vdash \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, dt_i, 0, M_i >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, \varnothing, |CT(p_i, t)|, M_i >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, dt_i, |CT(p_i, t)| - |DT(p_i, t)|, M_i >) \end{pmatrix} \rightarrow \begin{pmatrix} (\otimes_{i=1..n_1} < p_i, \varnothing, |DT(p_i, t)|, ? >) \otimes \\ (\otimes_{i=n_1+1..n_2} < p_i, CT(p_i, t), 0, ? >) \otimes \\ (\otimes_{i=n_2+1..n} < p_i, CT(p_i, t), 0, ? >) \otimes \end{pmatrix}$$

par définition de fh sur (8).

Références Bibliographiques

[ABK-02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Br¨uckner, P.D. Mosses, D. Sannella, and A. Tarlecki. «*Casl: The common algebraic specification language*», Theoretical Computer Science, 286:153–196, 2002.

[ABR-96a] J.R. ABRIAL, «*The B Book*», Cambridge University Press, ISBN 0521-496195, 1996.

[ABR-96b] J.R. Abrial, «*Extending B without Changing it (for Developping Distributed Systems)*», Proceeding of 1st Conference on the B method, pp. 169-190, Nantes, November 1996.

[ABR-97a] J-R. Abrial and L. Mussat, «*Specification and Design of a transmission protocol by successive refinements using B*», In M. Broy and B. Schieder (Eds), Mathematical Methods in Program Development, volume 158 of NATO ASI Series F: Computer and Systems Sciences, pp 129-200. Springer, 1997.

[ABR-97b] J.R. Abrial, «*Constructions d'Automatismes Industriels avec B*», Congrès AFADL, ONERA-CERT, Toulouse, Mai 1997.

[AEM-95] M.A. Alberti, P. Evi, and D. Marini, «*Modeling Geometric Objects with OBJSA Nets* », Proceeding of the Int. Workshop on Object-Oriented Programming and Models of Concurrency, Turin, Italy, June 1995 http://wrcm.dsi.unimi.it/PetriLab/ws95/home.html.

[AND-82] C. Andre , « *Use of the Behavior equivalence in Place-Transition Net Analysis*», In Application and Theory of Petri Nets, IFB 52, Springer, 1982.

[BB-87] T. Bolognesi and E. Brinksma, «*Introduction to the ISO Specification Language LOTOS*», Computer Networks and ISDN Systems 14, pp. 25-59, 1987.

[BDE-87] D. Bert, P. Drabik and R. Echahed, «*Manuel de référence de LPG* », Technical Report 17, IMAG-LIFIA,1987.

[BDK-01] E. Best, R. Devillers and M. Koutny, « *Petri Net Algebra*», EATCS Monographs on Theoretical Computer Science, ISBN 3-540-67398-9, Springer-Verlag, 2001.

[BDKP-91] E.Best, R.Devillers, A.Kiehn and L.Pomello, « *Concurrent bissimulations in Petri nets* », Acta Informatica Volume 28 , Issue 3 (1991) pp: 231 – 264, ISSN:0001-5903, 1991.

[BBG-97] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi «*Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 formalism*» In G. Agha and F. De Cindio, editors, Advances in Petri Nets on Object-Orientation, 1997.

[BDM-88] E. Battiston, F. De Cindio, and G. Mauri. « *OBJSA nets: a class of high level nets having objects as domains* » In G. Rozenberg, editor, Advances in Petri nets, LNCS Vol. 340, pages 20-43. Springer Verlag Berlin, 1988.

[BEL-02] F. Belala, «*Un cadre Formel pour l'Analyse des ECATNets*» Phd thesis, Université de Constantine, 2002.

[BER-87] G. Bernot, « *Good functors… are those preserving philosophy*», In Proc. of Summer Conference on Category theory and Computer Science, LNCS Vol.283, pp182-195, Springer-Verlag, 1987.

[BET-90] M.Bettaz « *An association of algebraic term nets and abstract data types for specifying real communication protocols* », 7th ADT-Workshop, Wusterhausen, LNCS Vol.534, 1991, p11-30.

[BG-77] R.M. Burstall and J.A. Goguen, « *Putting theories together to make specifications* », in Proc. Of the 5th International Joint Conference on Artificial Intelligence IJCAI, pp 1045-1058, 1977.

[BG-91a] Buchs D and Guelfi N., «*The CO-OPN Formalism* », in Workshop on Algebraic Nets, C3 Journal, January. 1991.

[BG-91b] D. Buchs and N. Guelfi, «*The CO-OPN Language*», in DEMON newsletter 10, 1991.

[BGL-01] S. Blom, W.J. Fokkink, J.F. Groote, I. Langevelde, B. Lisser and J.C. Pol, « *CRL: A toolset for analysing algebraic specifications*», In G.Berry, H. Comon, A. Finkel, eds.: Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings, LNCS Vol. 2102, pp 250–254.

[BGV-90] W.Brauer, R.Gold and W.Vogler, **«** *A survey of behaviour and equivalence preserving refinements of Petri Nets* », Advances in Petri Nets 1990 LNCS Vol.483, pp1-47.

[BID-89] M. Bidoit « *Pluss , un langage pour le développement de spécifications algébriques modulaires* », PhD Thesis , N°3541, Université de Paris-Sud , Centre d'Orsay, 1989.

[BJ-78] D. Bjørner and C. B. Jones, « *The Vienna Development Method: The Meta-Language*», LNCS, Vol. 61, Springer-Verlag,1978. ISBN 0-387-08766-4.

[BM-91] M. Bettaz and M. Maouche « *How to specify non determinism and true concurrency with algebraic term nets*» joints 8th WADT-3rd , Compass Workshop, Dourdan, France 8/1991, LNCS Vol.655, 1993, pp164-180.

[BM-95] M. Bettaz, and M. Maouche, «*Modelling of Object Based Systems with hidden sorted ECATNets*», MASCOTS' 95, Durham, North-Carolina, IEEE, 1995, pp. 307-311.

[BMSB-93]M. Bettaz, M. Maouche, M. Soualmi, and M Boukebeche, «*Compact modelling and rapid prototyping of communication software with ECATNets: a case study*». Simulation Series Vol.25 N.1, SCS 1993, pp.149-154.

[BOW-07] J. Bowen « *Formal Methods*», http://vl.fmnet.info/

[BR-94] M. Bettaz and G. Reggio «*A SMoLCS Based Kit for Defining the Semantics of High Level Algebraic Petri Nets*», LNCS Vol.785, pp.98-112, Springer-Verlag, 1994.

[BRA-82]G.W.Brams *« Réseaux de Petri: Théorie et Pratique*», Tome 1&2 , Eds Masson, 1982, ISBN 2-903-60712-5.

[BWW-88] J.Billington, G.R.Weeler and M.C.Wilbur-ham «*Protean: a high-level petri net tool for the specification and verification of communication protocols*», IEEE Transaction on Software Engineering, March 1988, Vol.14 N°3, pp301-316.

[CDE-07] M.Clavel, F. Duran, S. Eker, P. Lincoln, M. Marti-oliet, J.Meseguer and C. Talcott « *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic* », LNCS Vol. 4350, 2007.

[CDEMS-99] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M.-O. Stehr « *Maude as a Formal meta-tool*» Proc. FM'99, LNCS Vol.1709 , pp 1684-1703,Springer,1999.

[CER-92] K.Cerans, «*Decidability of bissimulation equivalences for parallel timer processes*», Proc. 4th Ann. Workshop on Computer-Aided Verification. LNCS Vol. 663, pp269-300, Springer Verlag, 1992.

[COL-88] D. Coleman et al., « *The AXIS specification language*», Technical report HPL-BRC-TM-88-020, Hewlett-Packard Laboratories (Bristol), 1988.

[CP-00] S.Christensen, and L.Petrucci, « *Modular Analysis of Petri Nets*», Computer Journal 43(3), 2000, pp.224-242.

[DAW-91] J. Dawes, «*The VDM-SL Reference Guide*», Pitman 1991. ISBN 0-273-03151-1

[DGMB-98] K. Djemame, D.G. Gilles, L.M. Mackenzie and M. Bettaz, « *Performance comparison of high-level algebraic nets distributed simulation protocols*», Journal of Systems Architecture 44, 1998, pp. 457-472.

[DW-96] J. Davies and J.Woodcock. «*Using Z*», Prentice Hall, ISBN 0-13-948472-8, 1996.

[EF- 83] H.Ehrig, W.Fey « Algebraic concepts applied to software development using parameterized specifications with requirements », Berlin. TU, Fachber. 20. 1983. 59 Bl.. Bericht. Technische Universität Berlin. Fachbereich 20, Informatik ; 83-13.

[EFHL-88] H. Ehrig, W.Fey, H. Hansen and M. Lowe « *algebraic theory of modular specification development* », Tech.Rep. 87-06 Fachbereich Informatik, Tech.Univ. Berlin, Sept 1987. Workshop on Algebraic Development Techniques, 1988.

[EFHLJ-88]] H.Ehrig, W.Fey, H.Hansen, M.Lowe and D.Jacobs « *Algebraic concepts for software configuration management* », Technische Report,. TU Berlin N° 88-19, August 88.

[EHH-89] H.Ehrig, A.Heis and U.Hummert « *Algebraic high level nets with capacities*», Technische Report, TU Berlin, 1989.

[EM-85] H. Ehrig and B. Mahr « *Fundamentals of algebraic specification: equations and initial semantics*», EATCS Vol.6, Springer-Verlag, 1985.

[ETL-84] H. Ehrig, J.W. Thatcher, P. Luca and S.N. Zilles, « *Denotational and initial algebra semantics of the algebraic specification language LOOK*», Technical Report 84-22, TU Berlin FB 20,1984.

[FG-06] D.Frutos-Escrig and C. Gregorio-Rodríguez, «*Process Equivalences as Global Bisimulations*». J. UCS 12(11): 1521-1550 , 2006.

[FUT-86] K. Futatsugi, « *An overview of OBJ2*», In Proc. Of the France-Japan Artificial Intelligence and Computer Science Symposium, pp29-48, 1986.

[GB-84] J.A. Goguen and R.M. Burstall « *Introducing institutions* », in Proc. of the Workshop on Logics of Programming, Sringer-Verlag, LNCS Vol.164, pp 221-256, 1984

[GH-83] J.V. Guttag and J. Horning, « *An introduction to the LARCH shared language*», In Proc. Of the IFIP'83 9th World Computer Congress, pp809-815, North-Holland, 1983.

[GT-79] J.A. Goguen and J. Tardo, «*An introduction to OBJ-T*», in Specification of Reliable Software, pp170-189, IEEE, 1979.

[GT-79] J.Goguen and .J Tardo «*An introduction to OBJ: a language for writing and testing formal algebraic program specifications*», In Proc. of IEEE Conference on Specification of Reliable Software, pages 170-188, 1979.

[GTW-77] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, « *Initial algebra semantics and continuous algebras*», Journal of the ACM, 24(1):68-95,1977.

[GTW-78] J.A. Goguen, J.W. Thatcher and E.G. Wagner, « *An initial approach to the specification, correctness, and implementation of abstract data types*», Vol.4 of current Trends in Programming Methodology, Prentice Hall, 1978.

[GUT-75] J.V. Guttag , « *The specification and application to programming of abstract data types* », PhD thesis, University of Toronto,1975.

[GV-03]Girault C., Valk R. and al., «*Petri Nets for Systems Engineering*», ISBN 3-540-41217-4 Editors Girault, C.and Valk, R., Springer-Verlag, 2003.

[HOA-85] C.A.R. Hoare «*Communicating Sequential Processes* », Prentice Hall International Series in Computer Science, ISBN 0-13-153271-5 (0-13-153289-8 PBK), 1985.

[ISO-88] ISO, «*LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*», International Standard 8807, ISO/OSI, Genève, September 1988.

[ISO-89] ISO/IEC. Information Processing Systems - Open Systems Interconnection: «*LOTOS, a Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*», IS 8807, February 1989.

[ISO-98] Committee Draft on Enhancements to LOTOS (E-LOTOS). Final Committee Draft, Juan Quemada, editor, ISO/IEC FCD 15437, avril 1998.

[JEN-92] K. Jensen «*Coloured Petri nets. Basic concepts, analysis methods and practical use*», *EATCS* Vol.1, monographs on Theoretical Computer Science, Springer 1992.

[JON-90] C. B. Jones, « *Systematic Software Development Using VDM*», Prentice Hall International, 1990. ISBN 0-13-880725-6.

[KKM-88] C. Kirchner, H. Kirchner and J. Meseguer, «*Operational semantics of OBJ3*", In Proc. Of the 15th International Colloquium on Automata, Languages and Programming ICALP, pp287-301, Springer-Verlag LNCS Vol.317, 1988.

[KL-83].B. Kutzler and F. Lichtenberger, « *Bibliography on abstract data types* », Springer-Verlag I.F.B. 68,1983.

[KRA-87] B. Krämer « *SEGRAS: a formal and semigraphical language combining Petri nets and abstract data types for the specification of distributed systems*», Proceedings of the 9th international conference on Software Engineering, Monterey, California, United States, 1987, pp: 116 - 125, ISBN:0-89791-216-0.

[LL-87] T. Lehmann and J. Loeckx, « *The specification language of OBSCURE*», In Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specification of Abstract Data Types, pp131-153, LNCS Vol. 332, Springer-Verlag, 1987.

[MAK-03] M. Mäkelä, « *Model checking safety properties in modular high-level nets*», In 24th Int. Conf. ICATPN 2003, The Netherlands, June 2003. LNCS Vol.2679, Springer-Verlag, pp. 201-220.

[MCD-91] Mc. Dermid (Edit.) «*Software Engineer's Reference Book*», Butterworth-Heinemann, Oxford, U. K., 1991.

[Mer-74] P. Merlin. «*A Study of the Recoverability of Communication Protocols*». Ph.D. Thesis, Computer Science Dep., University of California, Irvine, 1974.

[MES-96] J.Meseguer. «*Rewriting Logic as a Semantic Framework for Concurrency*» In U. Montanari and V. Sassone, editors, Proc. Concur'96, LNCS Vol. 1119, pp.331-372, Springer, 1996.

[MIL-89] R. Milner. «*Communication and Concurrency* », Prentice Hall International Series in Computer Science, C.A.R Hoare serie eds., 1989.

[MOS-00] T. Mossakowski, « *Casl: From semantics to tools*». In S. Graf and M. chwartzbach, editors, TACAS 2000, LNCS Vol. 1785, pp93–108, Springer-Verlag, 2000.

[MUR-89] T. Murata, « *Petri Nets: Properties, Analysis and Applications* », Poc. Of the IEEE, Vol. 77 N°4, April 1989, pp541-579.

[NMV-90] R.Denicola, U.Montanari and F.W.Vaangrager « *Back and Forth bissimulations*», Report CS-R9021, centre for mathematics and computer science PO.Box 4079, 1009 AB, Amsterdam, The Netherlands.

[NW-95] M. Nielsen and G. Winskel, «*Petri nets and bisimulation* ». TCS, Vol.153 pp:211-244, 1995.

[PET-79] C.A. Petri « *Introduction to general Net theory* » LNCS Vol.84, Net theory and applications, 1979.

[PF-03] E. Pelz, and H. Fleishhack, «*Compositional high level Petri nets with timing constraints*». Proc. ACSD' 03, June 2003, Guimaraes, Portugal.

[POM-85] L. Pomello, « *Some equivalence notions for concurrent systems. an overview*», in Advances in Petri Nets 1985, Proc. 6th European Workshop on Applications and Theory in Petri Nets, LNCS Vol. 222, pp: 381 – 400.

[REI-91] W.Reisig « *Petri nets and algebraic specifications*» Theoretical Computer Science 80(1991) p1-34.

[RES-99] P.Resende « *Modular specification of concurrent systems with observational logic*», LNCS Vol. 1589, pp 307-321. Spinger, 1999.

[RES-01] P. Resende « *Quantales, finite observations and strong bissimulations* » , TCS, 254, pp 95-149, 2001.

[SEN-96] Y. Souissi, A. Ezzati, and K. Nafil, «*Composition asynchrone de réseaux de Petri et conception modulaire de systèmes distribués*», CFIP'96, Rabat, 14-17 Oct. 1996, pp. 67-81.

[SPI-98] Spivey and John M. « *The Z notation: a reference manual*», Prentice Hall, International Series in Computer Science,1998, ISBN 0139785299.

[VOI-86] F. Voisin « *CIGALE : a tool for interactive grammar construction and expression parsing* », Science of Computer Programming, Vol.7 (1), pp61-86, 1986.

[WIL-87] M.C. Wilbur-Ham « *Numerical Petri nets*: A guide--version 2»," Telecom Australia Res. Labs., Switching and Signalling Branch Paper 111, Mar. 1987.

[WIN-87] J.M. Wing, « *Writing LARCH interface language specifications*», ACM Transactions on Programming Languages and Systems, 9(2): 1-25, 1987.

[Win-89] J.M. Wing, « *What is a Formal Method?* »,Carnegie Mellon University, Computer Science Dept CMU-CS-89-200, 1989.

[WIR-83] M. Wirsing, «*Structured Algebraic Specifications: A Kernel Language*», PhD thesis, Techn. Univ. Munchen, 1983.

[WIR-86] M. Wirsing, «Structured *Algebraic Specifications: A Kernel Language*», Theoretical Computer Science, 42(2) : 124-249,1986.

[WM-85] P.T.Ward and S.J.Mellor, «*Structured Development for Real-time Systems*», Yourdon Press computing series, ISBN 0-917072-51-0, 1985.

[WOR-96] J.B.Wordsworth « *Software Development with Z*», Addison-Wesley, 1996.

[YC-78] E.Yourdon and L.L.Constantine « *Structured Design*», Fundamentals of a Discipline of Computer Program and Systems Design, Prentice Hall, Englewood Cliffs, NJ, Yourdon Press, 1978.

[ZB-04] N.Zeghib and M. Bettaz , « *On Synchronous and Asynchronous Communication in Modular High-level Nets: The case of ECATNets*» in Proceedings of the 1st International Conference on Information & Communication Technologies: from Theory to Application- ICTTA'04, Damascus, Syria, 19-23 April, 2004, IEEE Catalog Number 04EX852C, ISBN 0-7803-8483-0

[ZB-05] N.Zeghib and M. Bettaz « CIRTA: a Formal Language For Modular ECATNets Specification », Revue Science&Technologie ISSN-1111-5041 N°24 Décembre 2005 pp47-56 .

[ZB-06] N.Zeghib and M. Bettaz «Embedded Systems Specification Using CIRTA : Application to an Acoustic Echo Canceller GMDGα » Proceedings of the 4th International Multiconference on Computer Science and Information Technology CSIT'06, April 5-7,Amman, Jordan, 2006, ISBN:9957-8592-0-X.

[ZBB-05] N.Zeghib, M. Bettaz and K.Barkaoui, « *CIRTA : An ECATNets Based Model for Embedded Systems Specification*», Proceeding of the 2005 International Conference on Embedded Systems and Applications ESA'05, Las Vegas, Nevada, USA, June 27-30, 2005, pp261-267, CSREA Press ISBN: 1-932415-53-X

[ZBB-06]N.Zeghib, Barkaoui and M. Bettaz « *Contextual ECATNets semantics in conditionnal rewriting logic* », Proceedings of the 4th ACS/IEEE AICCSA-06 International Conference on Computer Systems and Applications, Dubai/Sharjah UAE, March 8-11, 2006, pp936-943, ISBN 1-4244-0212-3.

[ZEG-90] N. Zeghib « *Conception et implémentation des outils de base d'un environnement de spécification algébrique*», thèse de magister, Institut d'informatique, Université de Constantine, 1990.

[ZEG-92] N. Zeghib, «*ASTRE: un environnement de spécification algébrique*», Proceedings of the 2nd Maghrebine Conference on Software Engineering and Artificial Intelligence MCSEAI'92, Tunis, April 13-16, 1992, pp02-16.

[ZEG-96] N.Zeghib « *Proposition d'une approche pour la paramétrisation des réseaux de Petri*» Proceeding du Séminaire National d'Informatique SNITO'96 Tizi-Ouzou 11-13/11/1996 p159-176.

[ZM-97] N.Zeghib and M. Maouche «*Proposition d'une approche pour la paramétrisation des ECATNets: application à un système de production*». Actes de MOSIM'97 modélisation et simulation des systèmes de production et de logistique, Rouen, France, 5-6 juin 1997, pp.425-433, ISBN :9782866016234, ed.Hermès.