N° d'ordre : 26/2012-M/inf

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ETPOPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE
LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE
HOUARI BOUMEDIENE

FACULTÉ D'ÉLECTRONIQUE ET D'INFORMATIQUE

MEMOIRE
Présenté pour l'obtention du diplôme de Magister
En : Informatique

Spécialité : Ingénierie des Logiciels et Génie des Procédés

Par : BOUBEZARI ABDERRAHIM

Thème

Transcription des Réseaux de Petri dans le système FoCaLiZe

Soutenu publiquement le 30/06/2012, devant le Jury composé de :

Mr - Belkhir Abdelkader, Professeur à U.S.T.H.B Président
Mr - Ben-Yelles Choukri-Bey, Professeur à U.P.M.F (Grenoble 2) Directeur de mémoire
Mlle – Selmi Nabila, Maître de conférence A à U.S.T.H.B Examinatrice
Mme – Zaouche Djaouida, Maître de conférence B à U.S.T.H.B Invitée

Remerciements

Je remercie tous les membres du jury pour avoir accepté d'examiner ma thèse.

Je tiens à remercier mon directeur de thèse, le Professeur BEN-YELLES Choukri-Bey, pour m'avoir fait confiance en me proposant ce sujet.

J'exprime ma profonde reconnaissance et mes vifs remerciements à Mme DAHMANI Djaouida et Mr ABBAS Messaoud pour le temps précieux consacré à mon encadrement, et leur apport considérable dans l'aboutissement de ce travail.

Je remercie Mr KADDOURI Lies, avec qui ce travail a été mis sur les rails.

J'adresse mes sincères remerciements aux membres du forum FoCaliZe, Messieurs Renaud Rioboo, David Delahaye, Guillaume Yziquel, François Pessaux, Lionel Habib et Mathieu Jaume, pour leurs remarques et leurs conseils.

Merci aux membres de ma famille pour leur soutien indéfectible, à mon ami et collègue DJERBI Rachid et à toutes les personnes qui ont participé de près ou de loin à la réalisation de cette thèse.

Enfin, une pensée à Mr BENABADJI Karim, ancien membre de l'équipe FoCAL de l'USTHB, et qui nous a tragiquement quitté depuis quelque temps.

Résumé

Les méthodes formelles regroupent l'ensemble des outils qui permettent de s'assurer de la fiabilité des logiciels par des raisonnements mathématiques rigoureux. Nous avons d'un côté des méthodes s'appuyant sur la représentation graphique telles que les réseaux de Petri, un outil formel de spécification et de modélisation de systèmes réels. D'un autre côté, nous avons les méthodes basées sur la preuve formelle comme le système FoCaLiZe, un environnement de développement orienté objet de programmation certifiée. Dans ce document, nous proposons une démarche de transcription des Réseaux de Petri ordinaires dans le système FoCaLiZe. Cette démarche permet d'obtenir une spécification FoCaLiZe exprimée en une hiérarchie d'espèces, à partir d'un modèle en réseau de Petri, et d'utiliser Zenon (outil de preuve automatique de FoCaLiZe), pour vérifier des propriétés des réseaux de Petri ordinaires.

Mots clés : FoCaLiZe, Réseaux de Petri, formel, preuve, Zenon, Coq.

.

Sommaire

Introduction

Les méthodes formelles sont une classe d'outils de spécification de systèmes, qui permettent d'atteindre un haut niveau de fiabilité en se basant sur des modélisations mathématiques, et en procédant à des démonstrations rigoureuses de l'absence de défauts de conception.
Les méthodes formelles se divisent en plusieurs catégories. Nous avons d'un côté, des outils basés sur la représentation graphique comme les réseaux de Petri [PET62], [DIA01], ils permettent de décrire un système réel et de l'analyser à travers des propriétés mathématiques afin de prévoir son comportement futur. De l'autre côté, nous avons des méthodes qui s'appuient sur la notion de démonstration mathématique, telles que FoCaLiZe [FCL09.a], [FCL09.b], un atelier de programmation certifiée [BOU00], qui permet de spécifier et de développer des logiciels, et de valider leur modèle grâce à des techniques de preuve formelle [JAU05], [ETI06].
FoCaLiZe est la dernière version d'un système appelé FoCAL [PRE03], [FOC06.a], [FEC05], acronyme de « Formal Ocaml Coq ALgebraic Library », et qui a été développé au sein du LIP6, le CNAM et l'INRIA.

Dans ce document, nous décrivons une démarche de transcription des réseaux de Petri dans le système FoCaLiZe, qui nous permettra d'exploiter ses techniques de preuve formelle pour vérifier des propriétés des réseaux de Petri ordinaires. Afin d'atteindre cet objectif, nous passerons par une phase de transcription pour obtenir une spécification FoCaLiZe à partir d'un modèle en réseaux de Petri. Nous commencerons par dériver les composants du réseau de Petri en une hiérarchie d'espèces FoCaLiZe, la relation entre les espèces définira le lien entre les composants. Les attributs des composants seront représentés par des types FoCaLiZe, qui seront manipulés par des méthodes introduites dans les espèces. Par la suite, d'autres espèces seront définies pour regrouper les composants dans des ensembles, et, finalement, une espèce globale spécifiera la structure du réseau de Petri en général.
Les propriétés des réseaux de Petri seront dérivées en propriétés et théorèmes FoCaLiZe, et seront prouvées grâce aux outils de preuve qui lui sont associés.
Dans les premières versions de FoCaLiZe, la preuve était déléguée au système d'aide à la preuve Coq [COQ10.a], [COQ10.b], depuis, un autre outil a été introduit, le prouveur Zenon [DOL04], [BON07], qui a permis d'améliorer le processus de preuve en le rendant automatique.

Plusieurs travaux se sont intéressés à l'association des réseaux de Petri à d'autres outils formels. Le plus connu d'entre eux est le model checking [CLA99.a], qui a été utilisé pour vérifier des propriétés dynamiques des réseaux de Petri de haut niveau [EVA05], [BER03]. Cette vérification passe par la construction de l'espace des états, ce qui s'avère peu pratique pour des systèmes de grandes tailles. Ceci a ouvert le champ à l'utilisation d'autres techniques, comme celles basées sur la preuve formelle. [BON00] a proposé une méthodologie globale de développement qui s'appuie sur les réseaux de Petri colorés [JEN92] et la méthode B [ABR96]. Cette démarche englobe une phase de traduction du réseau de Petri en B, permet de prouver des invariants du réseau de Petri grâce aux outils de preuve de B, et se termine par la production d'un code exécutable. L'outil de preuve Coq a également été utilisé pour la vérification des propriétés des réseaux de Petri, notamment la propriété de raffinement, sur les réseaux de Petri ordinaires [MAY08], et les réseaux colorés [CHO10]. Ces derniers travaux ont largement inspiré notre démarche de transcription vers FoCaLiZe.

Cette thèse est composée de quatre chapitres :

Dans le premier chapitre, nous présenterons les concepts de base du système FoCALiZe à travers un exemple complet, ainsi que les outils d'aide à la preuve COQ et Zenon.

Le deuxième chapitre introduit les éléments de base des réseaux de Petri ordinaires et leurs propriétés ainsi qu'une brève présentation de la technique de raffinement et de ses différentes variantes.

Dans le troisième chapitre, nous présenterons un état de l'art de méthodes et techniques formelles qui ont été associées aux réseaux de Petri.

Enfin, le quatrième chapitre est une présentation de notre démarche de transcription des réseaux de Petri ordinaires dans le système FoCALiZe. Il se termine avec la vérification de la propriété de raffinement grâce au prouveur Zenon.

Chapitre I

Le système FoCaLiZe

FoCaLiZe [FCL09.a] [FCL09.b], acronyme de « Formal Ocaml Coq ALgebraic Library », est un atelier de développement et de programmation certifiée [BOU00], développé dans les laboratoires du LIP6, du CNAM et de l'INRIA. Il permet de spécifier, programmer et s'assurer de la fiabilité d'un logiciel en intégrant des preuves de correction de la programmation vis-à-vis de la spécification dans le développement même de celui-ci. Ce développement est réalisé par une conception incrémentale, partant de spécifications abstraites vers des implantations concrètes, où à chaque étape des preuves sont apportées afin de préserver des propriétés et contraintes du système modélisé.

Les propriétés exprimées dans un code FoCaLiZe sont démontrées grâce aux outils associés à l'atelier que sont l'outil d'aide à la preuve Coq [COQ10.a], [COQ10.b], et le prouveur automatique Zenon [DOL04], [BON07]. Pour cela, FoCaLiZe produit, lors de la compilation, un fichier Coq, et un autre Zenon, ce dernier agissant comme un outil intermédiaire entre FoCaLiZe et Coq, pour rechercher automatiquement une preuve, qui est ensuite validée par Coq.

Au début du projet, l'atelier FoCaLiZe a permis de développer une librairie certifiée efficace destinée au calcul formel [RIO04]. Parmi les applications majeures, on peut citer l'implantation des politiques de contrôle d'accès [JAU05], la formalisation de la politique de sécurité d'un aéroport [DEL06], ou encore la modélisation de la réglementation de l'aviation civile [ETI06]. L'équipe Focal de l'USTHB pour sa part, a étudiée la traduction de spécifications UML en Focalize [ABB07].

Durant son évolution, le système FoCaLiZe a connu différentes versions. Dans la première d'entre elles, appelée *Foc* [PRE03], la preuve était uniquement déléguée à l'outil Coq. Sur la version suivante, FoCAL [FOC06.a] [FOC06.b], le prouveur automatique Zenon a été introduit. Le compilateur FoCaLiZe a été complètement réécrit sous sa dernière version, et de nouvelles spécifications ont été rajoutées à sa bibliothèque comme celle des ensembles finis.

Dans ce chapitre, nous avons opté pour une présentation des principales caractéristiques de FoCaLiZe, à travers un exemple concret complet.

1.1. Concepts de base de FoCaLiZe :

Le développement d'un programme FoCaLiZe repose sur la construction d'une hiérarchie de structures reliées logiquement entre elles, que sont les **espèces** et les **collections**. Une espèce (*species*) est une structure qui spécifie des opérations déclarées (*signature*) ou définies (*let*) et leurs propriétés (*property*, *theorem*). Ces opérations manipulent des éléments appelés **entités** d'un type appelé **type support** (*representation*). Les espèces subissent des raffinements successifs inspirés des techniques orientées objet (**héritage**, **paramétrage**…), pour obtenir en finalité des collections (*collections*) : des structures complètement définies destinées à être exploitées par l'utilisateur final.

Afin d'introduire les principaux concepts de FoCaLiZe, nous développerons, de manière informelle et pas à pas, un exemple concret : la spécification FoCaLiZe d'un système de contrôle de passage à niveau[1]. Le système est composé d'un contrôleur, d'un feu de signalisation et d'une barrière. Son comportement est le suivant :

- A l'état normal, le feu est au vert et la barrière est ouverte ;
- A l'approche du train, le contrôleur arrête la circulation, le feu passe à l'orange puis au rouge et la barrière est fermée ;
- Après le passage du train, le contrôleur débloque la circulation, la barrière s'ouvre, et le feu passe au vert.

Une spécification FoCaLiZe commence généralement par invoquer d'autres spécifications dont les définitions peuvent être importées et exploitées.

Dans notre exemple nous introduisons la bibliothèque **basics**[2], dans laquelle sont définis plusieurs types (int, bool, …) et fonctions (print…) de base.

1.1.1. Les espèces :

Les espèces sont les nœuds de la hiérarchie FoCaLiZe, elles correspondent au niveau le plus élevé d'abstraction dans une spécification. Une espèce peut être vue comme un enregistrement composé d'un ensemble de champs appelés **méthodes**, pour chacune d'entre elles, le programmeur a la possibilité de donner une **déclaration** abstraite (méthode déclarée) et /ou une **définition** concrète (méthode définie).

[1] Les détails du système modélisé ainsi qu'une présentation plus détaillée de la syntaxe, en FoCAL, peuvent être trouvés dans [ABB07].

[2] Ceci est réalisée en utilisant la directive ***use basics***.

Les méthodes inclues dans une espèce sont de trois types :

- Le **type support** : entités manipulées.
- Les **fonctions** : méthodes calculatoires.
- Les **propriétés** : méthodes non calculatoires.

La syntaxe générale d'une espèce est la suivante :

```
species <nom> =
      representation [= nom_type] ;      (*type de représentation*)
      signature <nom> : <type> ;         (*déclaration*)
      let <nom> = <corps> ;              (*definition*)
      property <nom> : <prop> ;          (*propriété*)
      theorem <nom> : <prop>             (*théorème*)
      proof  = <preuve> ;
end ;;
```

Notre spécification nécessitera l'introduction de quatre espèces, qui seront détaillées au fur et à mesure dans ce chapitre :

- *Barrier* : modélise la barrière dans le système ;
- *Light* : modélise le feu de signalisation ;
- *Control* : représente les commandes du contrôleur ;
- *Control_light_barrier* : regroupe l'ensemble des composants du système.

Type support :

Unique pour chaque espèce, le type support ou type de représentation définit la nature des entités manipulées par les autres méthodes de l'espèce. Il peut être abstrait ou concret, et est introduit par le mot clé **representation**.

Dans le cas des espèces *Barrier* et *Light*, la représentation est concrète. On utilise des types intégrés dans les définitions globales (au niveau top level).

```
type barrier_status = | Opened | Closed ;;
type light_status = | Green | Red | Orange ;;
species Barrier =
      representation = barrier_status ;
      ...
end ;;
species Light =
      representation = light_status ;
      ...
end ;;
```

Il n'est pas nécessaire de définir le type support de l'espèce *Control*, qui est encore abstrait à ce niveau de la spécification. Il pourra être défini par la suite à n'importe quel niveau d'héritage.

Méthodes calculatoires :

Les méthodes calculatoires correspondent aux opérations autorisées sur les entités du type support.

Ces fonctions sont introduites, soit comme des déclarations (seul le type est donné) grâce au mot clé **signature**, et dont les définitions seront données plus tard dans l'implémentation, soit comme des définitions complètes grâce au mot clé **let**.

Dans notre exemple, l'espèce *Barrier* intègre ces deux types de méthodes :

```
species Barrier =
      ...
      signature egale : Self -> Self -> bool ;
      signature close_barrier : Self -> Self ;
      signature open_barrier : Self -> Self ;
      let new_barrier(x : barrier_status) : Self = x ;
      ...
end ;;
```

1. Le mot clé **Self** dénote l'abstraction du type support de l'espèce courante.
2. La méthode *egale* représente l'égalité de deux éléments du type support,
3. Les méthodes *close_barrier* et *open_barrier* pour contrôler l'ouverture et la fermeture de la barrière.
4. La méthode *new_barrier* permet la création de nouvelles entités appartenant au type support.

Méthodes non calculatoires :

Les méthodes non calculatoires correspondent aux propriétés qui doivent être satisfaites par les éléments du type support et, donc, vérifiées par toute implémentation de l'espèce. Quand seul l'énoncé est donné, elles sont appelées **propriétés** et introduites par le mot clé **property**. Si la preuve est fournie, nous parlerons alors de **théorèmes**, introduits par le mot clé **theorem**. Dans ce dernier cas, la preuve elle-même, introduite par le mot clé **proof**, constitue la définition de la méthode.

Il est possible d'admettre sans preuve une propriété, il suffit alors de remplacer la preuve par le mot clé **assumed**. Cette technique permet au développeur de pouvoir continuer sa spécification et compiler son programme en retardant l'opération de preuve de la propriété concernée. Elle permet également de considérer la propriété comme un axiome dans la spécification.

Dans l'espèce *Barrier*, trois propriétés et un théorème sont introduits :

```
species Barrier =
      ...
      Property egale_transitive : all x y z : Self,
      egale (x, y)-> egale (y, z) -> egale (x, z) ;
      property open_to_close_prop : all x : Self,
      egale(x, ouverte) -> egale((close_barrier(x)), ferme) ;
      property close_to_open_prop : all x : Self,
      egale(x, ferme) -> egale((open_barrier(x)), ouverte) ;

      theorem barrier_transition : all x : Self,
      egale(x, ferme) -> egale((open_barrier(x)), ouverte)
      -> egale((close_barrier((open_barrier(x)))), ferme)
      proof =
      assumed ;
end ;;
```

1. *egale_transitive* : propriété qui définit la transitivité de la relation d'égalité.
2. *open_to_close_prop* et *close_to_open_prop*: modélisent le changement d'état de la barrière.
3. La méthode *barrier_transition :* théorème permettant de s'assurer de la cohérence du cycle de transitions des états de la barrière.

Une espèce dont toutes les méthodes sont définies (type support concret, toutes les fonctions définies et toutes les propriétés prouvées) est appelée **espèce complète** ou **terminale**.

A contrario, une **interface** est une espèce dont toutes les méthodes ne sont que déclarées. Toute espèce doit pouvoir être associée à une interface, elle permet à l'utilisateur final de connaître les fonctions utilisables et les propriétés sur ces fonctions sans accéder à leur implémentation.

Avant de présenter l'héritage et le paramétrage, nous complétons l'espèce *Light* et l'espèce *Control*.

```
species Light =
      representation = light_status ;
      signature egale : Self -> Self -> bool;
      signature lightRed : Self -> Self;
      signature lightGreen : Self -> Self;
      signature lightOrange : Self -> Self;
      let new_light(x : light_status) : Self = x;
      ...
      property lightRed_prop : all x : Self,
      egale(x, orange) -> egale((lightRed(x)), rouge) ;
      property lightOrange_prop : all x : Self,
      egale(x, vert) -> egale((lightOrange(x)), orange) ;
      property lightGreen_prop : all x : Self,
      egale(x, rouge) -> egale((lightGreen(x)), vert) ;

      theorem light_transition : all x : Self,
      egale(x, vert) -> egale(lightOrange(x), orange)->
      egale(lightRed (lightOrange(x)), rouge)->
      egale(lightGreen(lightRed (lightOrange(x))), vert) ->
      egale(x, lightGreen(lightRed (lightOrange(x))))
      proof =
      ... ;
end ;;
species Control =
      signature egale : Self -> Self -> bool;
      signature remettre_circulation : Self -> Self;
      signature arreter_circulation : Self -> Self;
      ...
end ;;
```

1. L'espèce *Light* introduit les fonctions *lightRed*, *lightGreen* et *lightOrange* pour contrôler le changement du feu de signalisation, et les propriétés correspondantes.
2. La fonction *new_light* permet de créer de nouvelles entités du type support.
3. Le théorème *light_transition* permet de prouver que tout élément *x* de l'espèce *Light* se trouve toujours dans un état sûr.

4. L'espèce *Control* est introduite pour modéliser les commandes du contrôleur du système : l'arrêt et la reprise de la circulation sont spécifiés par les fonctions *arreter_circulation* et *remettre_circulation*.

1.1.2. L'héritage et le paramétrage :

Une espèce peut aussi être définie à partir d'autres espèces en utilisant le mécanisme d'**héritage**. Dans ce cas, la nouvelle espèce hérite, de ses parents, toutes les méthodes, déclarées comme définies. Il est possible aussi de déclarer de nouvelles méthodes, de définir des méthodes déjà déclarées ou encore de redéfinir des méthodes définies par ailleurs.
Si l'espèce hérite de plusieurs autres espèces on parlera d'**héritage multiple**. Dans ce cas, l'ordre d'héritage est important. Si une même méthode est définie différemment dans deux espèces parentes, FoCaLiZe choisit la définition de l'espèce parente figurant dans la position la plus à droite dans l'ordre des espèces héritées, ordre établi dans l'entête de l'espèce héritière. De plus, le type support défini doit être le même dans toutes les espèces parentes, et ne doit en aucun cas être modifié lors de l'héritage afin de sauvegarder la propriété de sûreté de typage. FoCaLiZe vérifie également que les déclarations (types des fonctions ou énoncés des propriétés) sont cohérentes dans chacune des espèces parentes.

Une autre manière de combiner les espèces est d'utiliser le mécanisme de **paramétrage**. En effet, FoCaLiZe offre la possibilité de définir une espèce **paramétrée** soit par des collections soit par des entités appartenant à des espèces. Dans le premier cas, l'espèce peut faire appel à toutes les méthodes de ses paramètres, mais uniquement à travers leurs interfaces, sans pouvoir accéder aux définitions de leurs méthodes. A l'implémentation d'une espèce paramétrée, ses paramètres doivent être instanciés par les collections correspondantes.
La syntaxe de l'héritage et du paramétrage dans une espèce est la suivante :

```
species <nom> (<nom> is <nom>[(<parametres>)],
               <nom> in <nom>, . . .) =
inherit <nom>, <nom> (<parametres>), . . . ;
end ; ;
```

On utilise le mot clé **is** dans le cas d'un paramètre de collection, et le mot clé **in** si c'est un paramètre d'entité.

La dernière espèce spécifiée dans notre exemple, l'espèce *Control_light_barrier*, intègre héritage et paramétrage :

```
species Control_light_barrier (L is Light, B is Barrier) = inherit
Control ;
      representation = (L*B) ;
      let first (x : Self ) : L = fst(x) ;
      let scnd (x : Self ) : B = snd(x) ;
      signature arreter_circulation : Self -> Self;
      signature remettre_circulation : Self -> Self;
      let new_control(x : L, y : B) : Self = (x, y);

      property arreter_prop : all x : self ,
      egale(x, (L!vert, B!ouverte))-> egale(arreter_circulation(x),
      ((L!lightRed(L!lightOrange(first(x)))),
      (B!close_barrier(scnd(x))))) ;
      . . .
end ;;
```

1. L'espèce *Control_light_barrier* est définie par héritage de l'espèce *Control*, et paramétrée par les collections *L* et *B* implémentant les espèces *Light* et *Barrier*.
2. Le type support *L*B* dénote le produit cartésien *L!representation*B!representation* les types support de *L* et de *B*.
3. Les méthodes définies dans l'espèce *Control_light_barrier* utilisent des fonctions standards définies dans la bibliothèque FoCaLiZe, *fst* et *snd* permettent de récupérer le premier et le second élément d'une paire.
4. Un paramètre peut être utilisé comme un type FoCaLiZe dans l'espèce paramétrée comme *L* et *B* dans la définition de la méthode *new_control*.
5. La propriété *arreter_prop* fait appel aux différentes méthodes de ses paramètres : *L!vert*, *B!ouverte*, *B!close_barrier*…

Utiliser une méthode extérieure à l'espèce par héritage peut poser un problème particulier, c'est celui de la **dépendance**.

Une méthode m_1 dépend d'une méthode m_2, si elle contient un appel à m_2.

Deux types de dépendances existent :

- **Decl-dépendance** : Si m_1 a seulement besoin de connaître le nom et le type de m_2 ;
- **Def-dépendance** : Si, en plus du nom et du type, m_1 a besoin de connaître la définition de m_2.

Dans ce dernier cas, si m_2 est redéfinie au cours de l'héritage, la définition de m_1 devient invalide, et il faudra alors l'effacer et en fournir une nouvelle, en particulier s'il s'agit d'un théorème, sa preuve devra donc être refaite.

1.1.3. Les collections :

Une collection est une instance d'une espèce complète, qui permet d'utiliser les méthodes de cette espèce. Elle n'est vue qu'à travers son interface correspondante.

Cette abstraction, notamment celle du type support, permet d'assurer que les méthodes de la collection manipulent bien les entités pour lesquelles elles sont prévues, et d'empêcher l'utilisateur de la collection d'accéder à la définition de ses méthodes.

Les collections correspondent au dernier niveau de la spécification : le niveau implémentation. Ce sont des structures figées qui ne peuvent plus être raffinées par héritage.

La syntaxe d'une collection est la suivante :

```
collection <nom> = implement <nom> (<parametres>) end ;;
```

Les collections implémentant les espèces *Barrier* et *Light* sont les suivantes :

```
collection Barrier_collection = implement Barrier ; end ;;
collection Light_collection = implement Light ; end ;;
```

Pour la collection implémentant l'espèce *Control_light_barrier* il est nécessaire d'instancier les paramètres :

```
collection Control_light_barrier_collection = implement
Control_light_barrier (Light_collection, Barrier_collection) ;
end ;;
```

Nous pouvons utiliser les collections précitées pour faire appel à la commande d'arrêt de la circulation :

```
let vert = Light_collection!new_light (#Green) ;
let ouverte = Barrier_collection!new_barrier (#Opened) ;
let etat_initial =
Control_light_barrier_collection!new_control (#vert, #ouverte) ;
let  arret =
Control_light_barrier_collection!arreter_circulation (#etat_initial);
```

NB : **C!m** désigne la méthode *m* de la collection *C* qui implémente l'espèce en cours. **f#m** désigne la méthode *m* définie dans un fichier *f* d'une spécification référencée au niveau top-level, tandis que **#m** désigne une méthode *m* définie dans la spécification courante.

Nous obtenons la hiérarchie des espèces de la spécification suivante (figure1.1) :

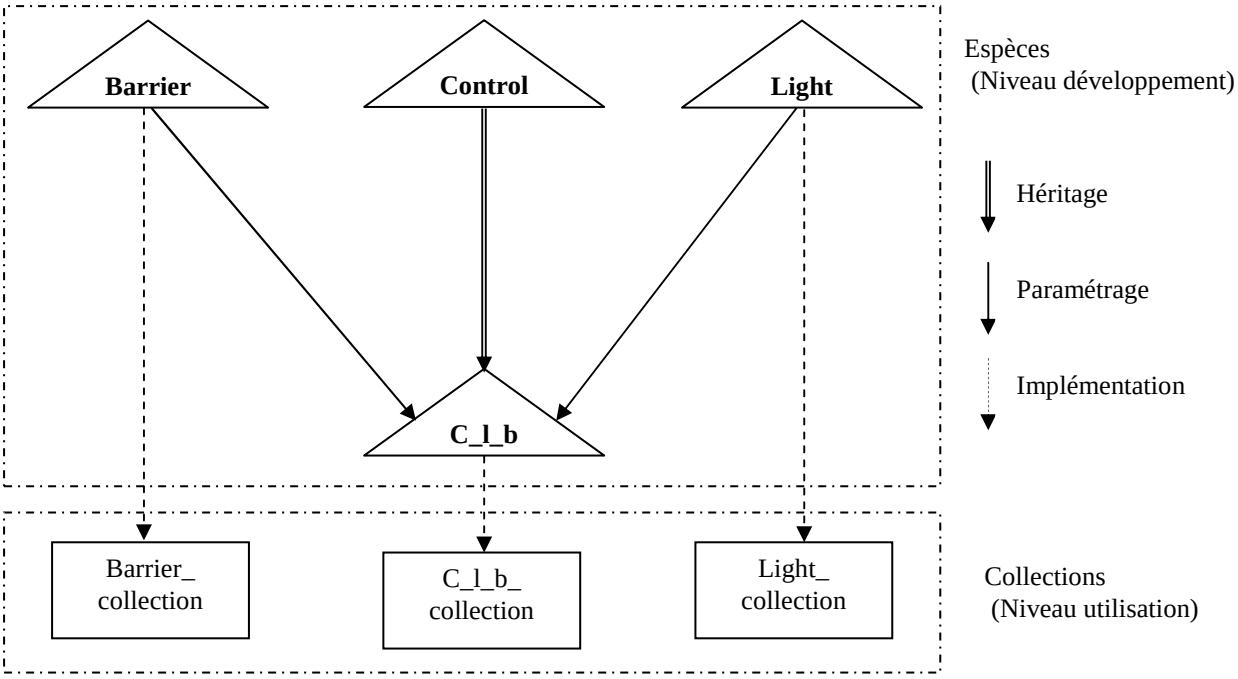

Fig 1.1 Hiérarchie des espèces de la spécification du système de contrôle de trains.

1.2. Compilation d'un programme FoCaLiZe : [FCL09.a]

La compilation d'un programme FoCaLiZe est constituée de deux passes. Dans la première, le système vérifie que les spécifications et les preuves sont bien formulées, et que le code est bien typé. Dans la seconde, le système vérifie la correction des preuves données par l'utilisateur.

1. **Première passe** : Typage et analyse statique.

Cette étape permet de vérifier certaines règles :

- Les arguments donnés aux espèces paramétrées doivent avoir le type ou l'interface attendus.
- Le type support ne doit pas changer au cours de l'héritage.
- La redéfinition d'une méthode au cours de l'héritage ne doit pas en changer le type.
- Pour créer une collection à partir d'une espèce, les méthodes de celle-ci doivent toutes être définies.
- Eviter les cycles de dépendances et résoudre les conflits en cas d'héritage multiple.

2. **Seconde passe** : Génération et preuve

Elle produit quatre fichiers en sortie (figure1.2) :

- Pour l'exécution : le code Ocaml [LER05].
- Pour la documentation : le source pour un langage intermédiaire (FoCaLiZeDoc) permettant de créer des fichiers de documentation automatique sous différents formats (HTML, XML, LATEX).
- Pour l'automatisation des preuves : le source pour Zenon, outil d'aide à la preuve automatique, qui construit une preuve et le terme Coq correspondant.
- Pour la certification : le code Coq permettant de valider la preuve et certifier le logiciel.

Les étapes de compilation d'un programme FoCaLiZe et les fichiers produits à chaque étape sont illustrés dans la figure 1.2 [PRE05].

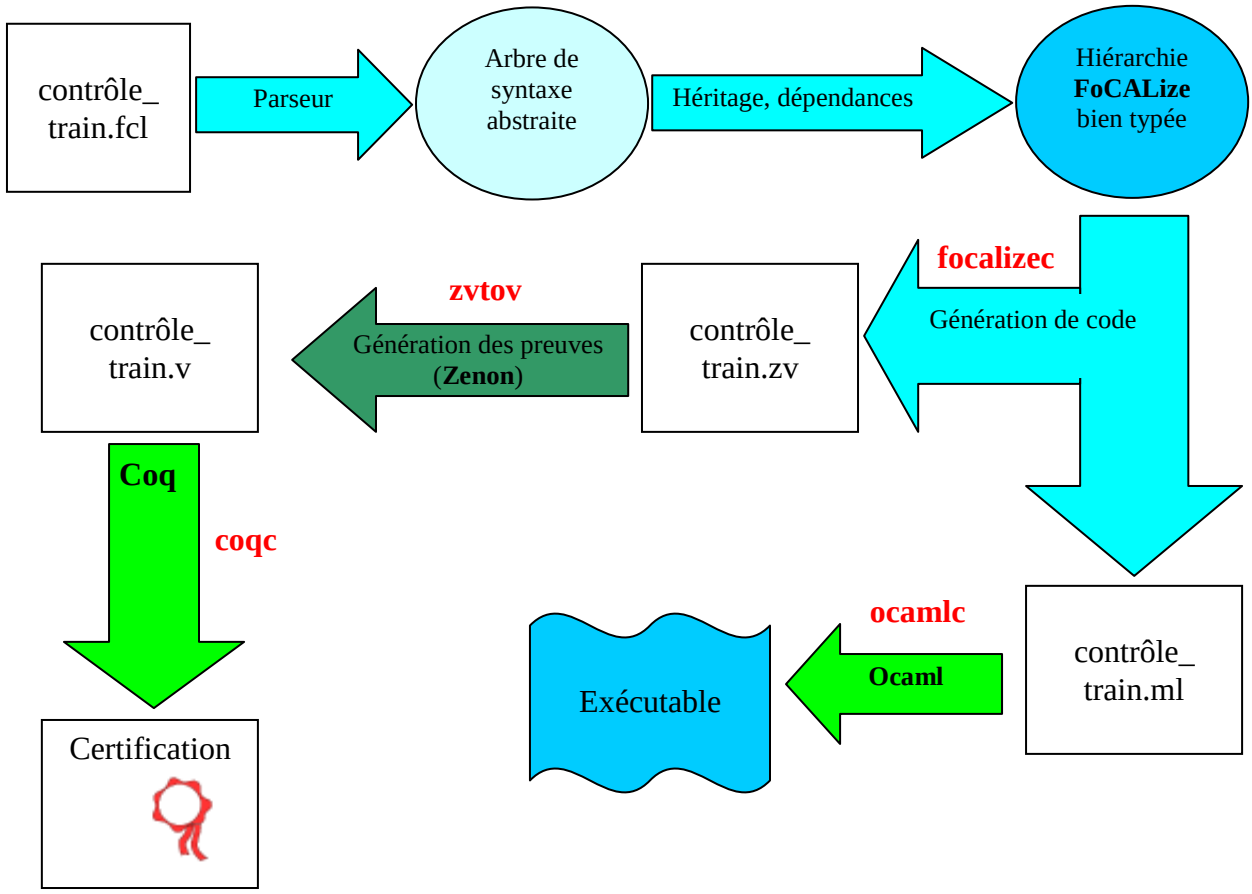

Fig 1.2 Etapes de compilation dans le langage FoCaLize.

1.3. Faire la preuve avec FoCaLiZe :

Dans les anciennes versions de FoCaLiZe, c'est au système d'aide à la preuve Coq que revenait la tâche de réaliser les preuves des propriétés fournies dans les différentes espèces de la spécification. Par la suite, le prouveur Zenon a été développé afin de faciliter cette phase. En effet, ce dernier s'occupe de rechercher la preuve de façon automatique, preuve qui nécessitera tout de même une validation finale par Coq.

Dans ce qui suit, nous allons présenter ces deux outils.

1.3.1. Coq :

Coq [COQ10.a], [COQ10.b] est un système d'aide à la preuve basé sur la théorie des types (le calcul des constructions inductives plus précisément), et utilise une logique de base intuitionniste. Il permet d'écrire des spécifications, des définitions et des théorèmes et de réaliser des preuves formelles, son but étant le développement et la certification de logiciels sans bugs.

L'association de FoCaLiZe et Coq pour la construction des preuves se fait selon les étapes suivantes :

1. La compilation du code source FoCaLiZe permet de traduire les spécifications en un fichier de scripts Coq : une espèce est encodée en un **enregistrement** Coq (appelé également chapitre) dont les champs (ou variables) représentent les signatures, les méthodes définies sont traduites en **termes** Coq. Quant aux propriétés, chacune d'entre elles est représentée par une **hypothèse**.
2. Les preuves sont ensuite complétées manuellement en Coq. Ceci nécessite de savoir comment un source FoCaLiZe est traduit en Coq.
3. Les scripts de preuve Coq sont finalement copiés à leurs places (à la suite du mot clé *proof*) dans le code source FoCaLiZe.

Afin d'illustrer l'association entre FoCaLiZe (spécification) et Coq (preuve), nous allons prouver un théorème à partir de la fonction d'égalité qui figure dans l'espèce *Light*, et ce en utilisant Coq.

```
species Light =
      representation = light_status ;
      . . .
      signature egale : Self -> Self -> bool ;
      let different (x, y) = ~(egale (x, y)) ;
```

```
        theorem difference : all x y : Self,
        (egale (x, y)) -> ~(different (x, y))

    (* preuve Coq du théorème, intégrée dans l'espèce Light *)
        proof =
        coq proof
        {* unfold abst_different. unfold Light_different.
        intros x y; case (abst_egale x y);simpl.
        unfold not;intros HH1 HH2;elim HH2.
        intro HH;elim HH. *};
        . . .
    end ;;
```

Pour réaliser une preuve, Coq utilise des **tactiques**, des fonctions qui construisent une preuve d'un but donné à partir de preuves élémentaires de sous-buts (un but étant la propriété principale que nous voulons démontrer, un sous-but est la propriété à démontrer à un certain niveau de preuve).

Dans l'exemple proposé, nous avons les tactiques suivantes :

- *Unfold idf* : permet remplacer toutes les occurrences de l'identifiant *idf* par sa définition dans la spécification courante (*unfold abs_different*, remplace cet identifiant par la définition de la méthode *different* présente dans le corps de l'espèce *Light*).
- *Intro* : permet d'introduire une hypothèse (*intros x y*, introduit les éléments *x* et *y*, de type *Self* dans la propriété *difference*).
- *Case* : permet de prouver la propriété en procédant par cas (dans l'exemple, deux cas possibles, selon la valeur de la proposition *abst_egale x y*, vrai ou faux).
- *Elim* : permet d'appliquer un schéma d'induction (en prenant le second cas où *abst_egale x y* prend la valeur faux, la propriété est prouvée par *elim HH*, nous pouvons déduire n'importe quoi d'une proposition fausse).

Une présentation exhaustive des tactiques utilisées dans coq est disponible dans [BER11].

1.3.2. Zenon :

Zenon [DOL04] est un prouveur automatique de théorèmes pour la logique du premier ordre, basé sur la méthode des tableaux.

Zenon est destiné à être le prouveur de l'environnement FoCaLiZe, il produit des preuves directement vérifiables en Coq, qui servira, dans ce cas la, uniquement pour la validation du code final.

Nous donnons un exemple de la preuve du théorème *barrier_transition* de l'espèce *Barrier* en utilisant Zenon :

```
species Barrier =
      . . .

      theorem barrier_transition : all x : Self,
      egale(x, ferme) -> egale((open_barrier(x)) , ouverte) ->
      egale((close_barrier((open_barrier(x)))) , ferme)

      proof =
      <1>1 assume x : Self
           assume H1: egale(x, ferme)
            prove egale(x, ferme) -> egale((open_barrier(x)),
            ouverte) -> egale((close_barrier((open_barrier(x)))),
            ferme)
      <2>1 prove egale(x, ferme) -> egale((open_barrier(x)), ouverte)
            by hypothesis H1 property close_to_open_prop
      <2>2 prove egale((open_barrier(x)), ouverte) ->
            egale((close_barrier((open_barrier(x)))), ferme)
            by step <2>1 property open_to_close_prop
      <2>3 prove egale(x, ferme) -> egale((open_barrier(x)), ouverte)
            -> egale((close_barrier((open_barrier(x)))), ferme)
            by step <2>1 , <2>2
      <2>4 conclude
      <1>2 conclude ;

. . .
end ;;
```

Certaines preuves ne peuvent être générées en temps limité, ou sont interminables pour un générateur de preuves automatique. Afin d'aider Zenon à trouver la preuve le plus rapidement possible, le développeur peut le guider par une preuve structurée composée d'**indications de preuve** écrites dans un langage appelé **FoCaLiZe proof language** (suivant un ordre précis, <x>y : x, dénote le niveau de preuve, y, le cheminement de la preuve en cours). Elles représentent le chemin à suivre pour arriver au but.

Les indications de preuve sont de trois types :

1. **Par hypothèse** (by hypothesis, exemple <2>1), où une hypothèse est utilisée comme axiome pour prouver le niveau de preuve courant.

2. **Par propriété** (by property, exemple <2>1), la preuve est réalisé grâce à une propriété prouvée par ailleurs dans la spécification.
3. **Par étape** (by step, exemple <2>3), un niveau de preuve précédent est utilisé pour apporter la preuve du niveau courant.

Ces types d'indication peuvent être combinés les unes aux autres pour la réalisation de la preuve. Il est à noter que si Zenon échoue, le développeur doit revoir et modifier ces indications puis refaire le même processus.

Réaliser des preuves de propriétés dans une spécification FoCaLiZe directement en Coq, nécessite une connaissance préalable de la manière dont les constructions et éléments de base de FoCaLiZe sont traduits en Coq. L'introduction de Zenon s'avère donc intéressante pour le concepteur, car elle limite sa participation dans l'élaboration de la preuve en automatisant le processus, tout en utilisant des mécanismes similaires à Coq comme :

- L'introduction d'hypothèses.
- L'utilisation de propriétés déjà prouvées dans la preuve de nouvelles propriétés.
- La réalisation de la preuve par étapes, en décomposant la propriété en sous-propriétés plus facile à prouver en utilisant les indications de preuve. Ceci rend la preuve plus compréhensible car plus proche d'un raisonnement mathématique.

L'usage de Coq reste essentiel pour la vérification du code final, mais ceci donne un atout supplémentaire à Zenon, car il apporte une garantie certaine à la fiabilité des preuves apportées par Zenon.

Conclusion :

La présentation de l'atelier FoCaLiZe à travers l'exemple du système de contrôle de passage à niveau, a permis d'illustrer une démarche permettant d'obtenir un modèle formel à partir d'un cas réel. Grâce aux éléments de base de FoCaLiZe, il a été possible de :

- Capturer les composants du système de contrôle, en utilisant les constructions d'espèces et de collections ;
- Etablir les liens entre les composants grâce au mécanisme de paramétrage ;
- Modéliser les différentes fonctionnalités et commandes du système par les fonctions définies dans les espèces ;
- S'assurer du bon fonctionnement du système de manière rigoureuse, en exprimant des propriétés sur celui-ci, et en les prouvant grâce aux outils Zenon et Coq.

Chapitre II

Présentation des réseaux de Petri

Les *réseaux de Petri* ont été créés au début des années 60 par le mathématicien allemand C.A Petri [PET62], dans le but de résoudre le problème du parallélisme des processus dans un système. Ils ont été développés par la suite, notamment dans le cadre du projet MAC [HOL70] [HAC72], au MIT (Massachusetts Institute of Technology) dans les années 70.

Les réseaux de Petri [DIA01] sont un outil mathématique formel de modélisation basé sur la représentation graphique. Un modèle en réseau de Petri permet, non seulement, d'exprimer des caractéristiques telles que la concurrence, le non déterminisme ou la synchronisation, mais également d'analyser le système modélisé afin de l'évaluer et l'améliorer.

Grâce à la simplicité du modèle graphique et aux résultats des recherches obtenus, les réseaux de Petri ont été largement exploités dans de nombreux domaines comme les protocoles de communication [BIL99] ou le génie logiciel.

Cependant, et malgré leurs avantages, l'évaluation en réseaux de Petri simples n'est que qualitative, et ne prend pas en charge des contraintes telles que le temps moyen d'exécution d'une tache, le taux de perte de message sur un réseau ou la différenciation des types de ressources. Ceci s'avère insuffisant pour s'assurer du bon fonctionnement de systèmes complexes comme les systèmes distribués ou à temps réel. Afin d'intégrer ce type de paramètres dans l'analyse, une nouvelle classe de réseaux a été introduite, les réseaux de Petri de haut niveau, parmi lesquels nous pouvons citer : les réseaux temporisés [RAM74], les réseaux colorés [JEN92] ou les réseaux stochastiques [MOL82], [NAT80].

Obtenir un modèle satisfaisant en réseau de Petri n'est pas uniquement en rapport avec le type du réseau, mais passe aussi par une conception incrémentale, durant laquelle des transformations sont apportées sur un modèle initial pour aboutir au modèle final validé. Ces transformations s'appuient sur des techniques comme le raffinement [LAK01], qui consiste à modifier la structure du réseau pour en faciliter l'analyse.

Nous commençons notre chapitre par la description des notions de base de la structure et du comportement des réseaux de Petri, et leurs principales propriétés. Ensuite, nous donnons une brève présentation des réseaux de Petri colorés, et nous terminons ce chapitre en énonçant une définition de raffinement des réseaux de Petri.

Notre présentation des principaux concepts des réseaux de Petri s'appuie principalement sur les ouvrages de G.W. Brams [BRA83] et de M. Diaz [DIA01], les définitions utilisées dans cette section en sont extraites. Par ailleurs, pour illustrer notre propos, nous utiliserons l'exemple d'un réseau de Petri modélisant un atelier de coupe de bois, développé dans le cours de G. Scorletti et G. Binet [SCO06].

2.1. Structure et marquage d'un réseau de Petri :

Un réseau de Petri se représente par un graphe orienté comprenant deux types de sommets : les places et les transitions qui sont reliées par des arcs. Deux sommets de même type ne peuvent pas être reliés entre eux.

Définition 1 : Réseau de Petri [BRA83], [DIA01]

Un **réseau de Petri** est un quadruplet $R = (P, T, Pre, Post)$ où

- $P = \{p_1, p_2, \ldots, p_n\}$ est un ensemble fini de places.
- $T = \{t_1, t_2, \ldots, t_m\}$ est un ensemble fini de transitions.
- P et T sont non vides et disjoints.
- $Pre, Post : P \times T \rightarrow \mathbb{N}$ sont les applications d'incidence avant et arrière, respectivement, et qui attribuent à chaque arc du réseau un entier appelé **poids.** $Pre(p, t)$ donne le poids de l'arc reliant la place p à la transition t, et $Post(p, t)$ donne le poids de l'arc reliant la transition t à la place p. Un poids de 0 indique l'absence d'arc.

Remarque :

- On note $°t$ l'ensemble des places en entrée de la transition $t \in T$, et $t°$ l'ensemble de ses places en sortie.
- Dans la suite du chapitre, nous utiliserons les notations introduites sans redéfinition systématique.

Pour définir l'état d'un système modélisé par un réseau de Petri, il est nécessaire de le compléter par un marquage, qui associe un nombre de jetons à chaque place du réseau.

Définition 2 : Réseau de Petri marqué [DIA01]

Un réseau de Petri **marqué** est un couple (R, M) dans lequel R est un réseau de Petri et $M : P \rightarrow \mathbb{N}$ est une application appelée **marquage.**

Le marquage d'un réseau de Petri est représenté par un vecteur de $\mathbb{N}^P$ regroupant les marquages de toutes les places du réseau.

Le **marquage initial**, noté M_0, est le marquage du réseau à l'instant initial t = 0.

Exemple :

Modélisation d'un atelier de coupe de bois (figure 2.1) :

L'atelier est constitué d'une machine de coupe et d'un stock. Quand une commande arrive et que la machine est disponible, la commande peut être traitée. Une fois le traitement terminé, la commande est stockée, sinon, la commande reste en attente de la libération de la machine.

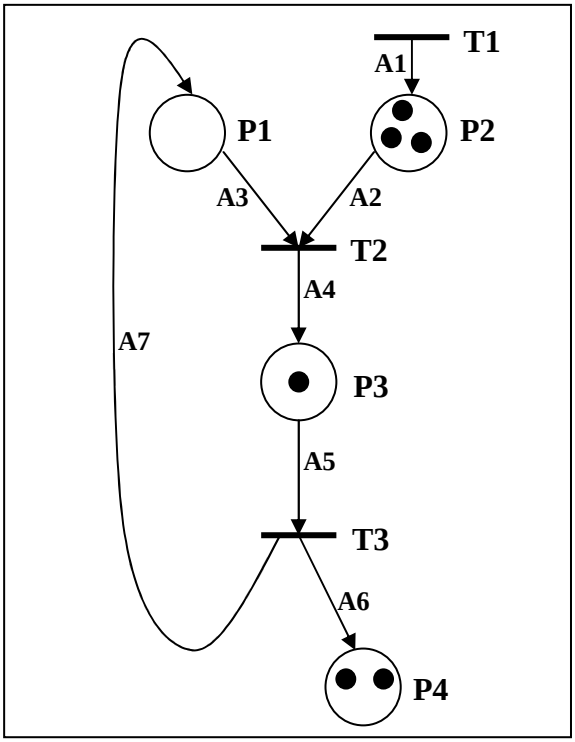

Fig 2.1 Réseau de Petri modélisant l'atelier de coupe.

1. $P = \{P_1, P_2, P_3, P_4\}$:

   Les places du réseau représentent les variables d'état du système à modéliser. A un instant donné, une place contient un nombre de **jetons** ou de **marques**, destiné à évoluer dans le temps, et qui indique la valeur de la variable d'état à cet instant :

   $M(P_1)$ représente le nombre de machines disponibles, $M(P_2)$ le nombre de commandes en attente de traitement, $M(P_3)$ le nombre de commandes en instance de traitement et $M(P_4)$ le nombre de commandes déjà traitées en stock.

   Le marquage du réseau de la figure 2.1 est donné par :

$$M_0 = \begin{bmatrix} 0 \\ 3 \\ 1 \\ 2 \end{bmatrix}$$

2. $T = \{T_1, T_2, T_3\}$:

   Les transitions du réseau représentent les actions ou les événements qui entraînent l'évolution du marquage du réseau, et donc de l'état du système :

   T_1 indique l'arrivée d'une nouvelle commande, T_2 le traitement d'une commande si une machine est disponible, et T_3 le stockage dès la fin du traitement d'une commande.

3. *Pre* et *Post* : le poids d'un arc indique, le nombre de ressources nécessaires à consommer s'il est entrant (d'une place vers une transition), ou le nombre de ressources produites s'il est sortant (d'une transition vers une place), lors du changement d'état du système.

   Dans notre exemple, tous les arcs ont le même poids égal à 1.

2.2. Comportement d'un réseau de Petri :

Le comportement d'un réseau de Petri se traduit par l'évolution de son marquage. La transition d'un marquage à un autre ne peut se faire que si toutes les conditions relatives à cette transition sont satisfaites, ce qui définit la règle de franchissement d'une transition.

Définition 3 : Franchissement d'une transition [BRA83], [DIA01]

Une transition *t* est dite **franchissable** (**sensibilisée** ou **tirable**) pour un marquage *M* si et seulement si :

$$\forall p \in {}^\circ t, M(p) \geq Pre(p, t)$$

Le **franchissement** d'une transition conduit à un nouveau marquage *M'* défini par :

$$\forall p \in P, M'(p) = M(p) - Pre(p, t) + Post(p, t)$$

Le franchissement ou le tir d'une transition *t* sera noté : $M [t > M'$.

Le marquage d'un réseau peut évoluer si le nombre de jetons dans chacune des places en entrée d'une transition est supérieur ou égal au poids des arcs reliant ces places à la transition.

La règle de franchissement signifie que le nouveau marquage *M'* est obtenu à partir du marquage *M* en retirant de chacune des places en entrée de la transition *t* un nombre de jetons

égal au poids de l'arc reliant la place à la transition, et en ajoutant à chacune de ses places en sortie un nombre de jetons égal au poids de l'arc reliant la transition à la place.

Dans l'exemple de la figure 2.1, à partir de M_0, nous avons :

1. La transition T_3 est franchissable.
2. La transition T_2 n'est pas franchissable.
3. La transition T_1 ne possède pas de place en entrée, c'est une **transition source**. Elle est par définition toujours franchissable.

   Une transition sans place en sortie est appelée **transition puits**, elle ne peut être franchie.

Exemple :

Évolution du marquage du réseau de coupe de bois (figure 2.2) :

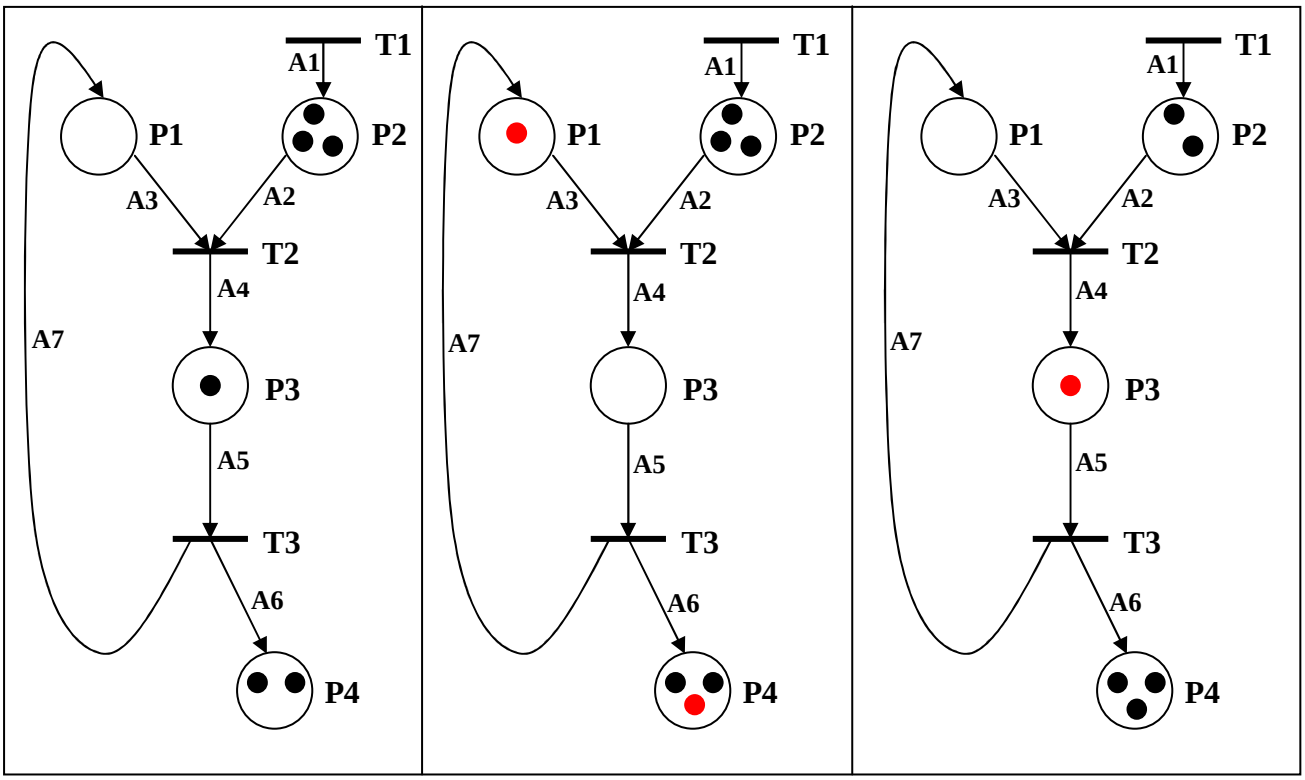

Fig 2.2 Evolution du réseau de l'atelier de coupe.

A partir d'un marquage donné, il peut être possible de franchir plusieurs transitions l'une après l'autre menant à des marquages différents.

Définition 4 : Séquence de franchissement [DIA01]

Soit (R, M_0) un réseau de Petri marqué. Une **séquence de franchissement** $S \in T^*$ est une séquence ordonnée de transitions t_1, t_2, …, t_n telle qu'il existe n marquages M_1, M_2, …, M_n avec :

$$M_{i-1}\,[t_i > M_i \text{ pour } i = 1,\ldots, n$$

Le franchissement de la séquence S sera noté : $M_0\,[S > M_n$.

Un marquage obtenu à partir du marquage initial M_0 par la suite d'une séquence de franchissement de transitions du réseau est appelé marquage accessible.

Définition 5 : Marquage accessible [DIA01]

Soit (R, M_0) un réseau de Petri marqué. Un marquage M est **accessible** si et seulement si :

$$\exists\ S \in T^*,\ M_0\,[S > M.$$

L'idée la plus naturelle pour étudier le comportement d'un réseau de Petri est d'énumérer tous les états possibles. Ceci est réalisé par la construction d'un graphe appelé graphe des marquages accessibles.

Définition 6 : Ensemble des marquages accessibles [DIA01]

Soit (R, M_0) un réseau de Petri marqué. L'**ensemble des marquages accessibles** ou **ensemble d'accessibilité** d'un réseau, noté $A(R, M_0)$, est l'ensemble des marquages atteints par une séquence de franchissement :

$$A(R, M_0) = \{M \in \mathbb{N}^P \mid \exists S \in T^*,\ M_0[S>M\}$$

Définition 7 : Graphe d'accessibilité [DIA01]

Soit (R, M_0) un réseau de Petri marqué. Le **graphe d'accessibilité** (ou **graphe des marquages accessibles**) d'un réseau, noté $G(R, M_0)$, est un graphe constitué de nœuds représentant les marquages accessibles de $A(R, M_0)$, et d'arcs étiquetés par les noms des transitions franchies : un arc étiqueté par t joint M à M' si et seulement si $M\,[t> M'$.

Exemple :

Graphe des marquages accessibles de l'atelier coupe de bois, où pour simplifier, nous avons supprimé la transition source T_1 (figure 2.3).

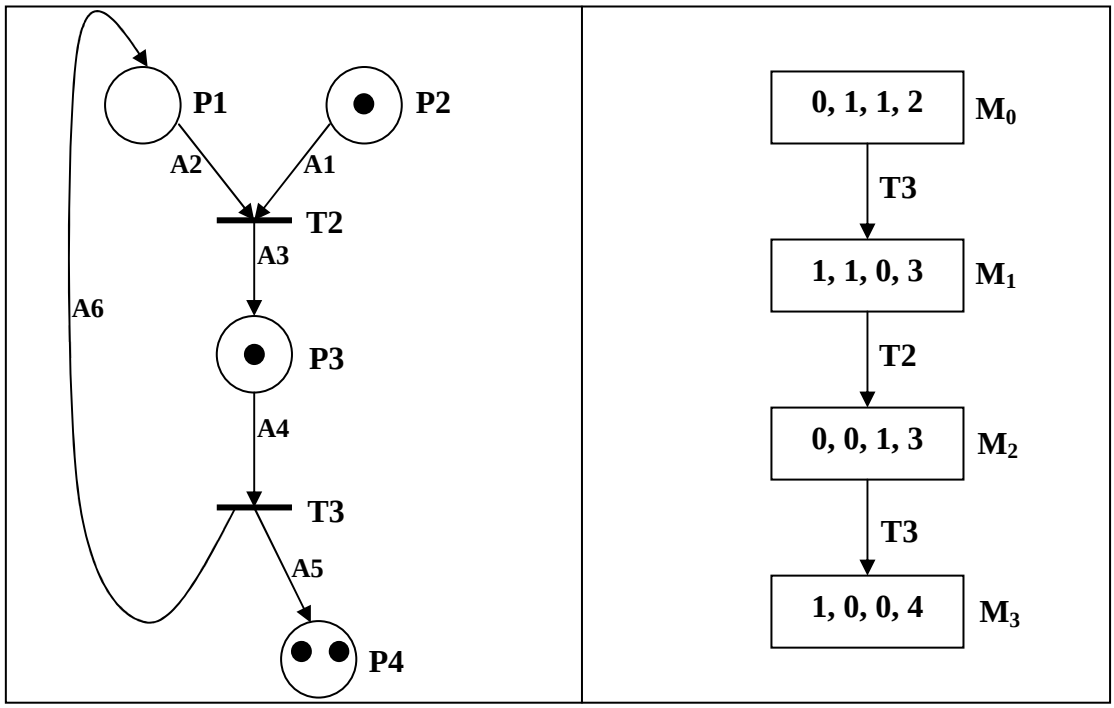

Fig 2.3 Atelier coupe de bois : le graphe des marquages accessibles.

2.3. Propriétés des réseaux de Petri :

Nous nous intéressons aux deux types de propriétés des réseaux de Petri, que sont le caractère borné d'un réseau et son degré d'activité. Le nombre de jetons circulant dans le réseau reste-t-il borné ? Le réseau peut-il évoluer ?

2.3.1. Bornitude :

Cette propriété caractérise le nombre limité ou fini de jetons qu'une place peut accumuler. Elle dépend du marquage initial du réseau.

Définition 8 : K-bornitude [BRA83], [DIA01]

Etant donné une valeur entière *k*.

Une place *p* est dite **k-bornée** si et seulement si :

$$\forall M \in A(R, M_0), M(p) \leq k$$

Définition 9 : Bornitude [BRA83], [DIA01]

Un réseau marqué (R, M_0) est dit **borné** si et seulement si il existe un entier *k* tel que toutes les places de *R* sont k-bornées :

$$\exists k \in \mathbb{N}, \forall p \in P, \forall M \in A(R, M_0), M(p) \leq k$$

Dans l'exemple de l'atelier de coupe de la figure 2.1, la place P_2 possède en entrée une transition source qui peut produire des jetons indéfiniment, par conséquent cette place n'est pas k-bornée, et le réseau de Petri de la figure 2.1 n'est pas borné.

2.3.2. Blocage :

L'une des propriétés plus importantes à vérifier sur le comportement d'un réseau de Petri est l'absence de blocage, impliquant que le réseau a toujours la possibilité d'évoluer.

Définition 10 : Réseau sans blocage [BRA83]

Soit un réseau de Petri (R, M_0). Un marquage accessible M est dit **blocage** (**état puit** ou **marquage mort**) si et seulement si aucune transition n'est franchissable pour ce marquage :

$$\forall t \in T, \exists p \in {}^\circ t, M(p) < Pre(p, t)$$

Un réseau de Petri (R, M_0) est dit **sans blocage** pour un marquage initial M_0 si aucun marquage accessible n'est un blocage :

$$\forall M \in A(R, M_0), \exists t \in T, \forall p \in {}^\circ t, M(p) \geq Pre(p, t)$$

Le réseau de la figure 2.1 est sans blocage pour le marquage initial $M_0 = \begin{bmatrix} 0 \\ 3 \\ 1 \\ 2 \end{bmatrix}$, car la transition T_1 est une transition source et toujours franchissable.

Par contre, le marquage $M_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 4 \end{bmatrix}$ est un blocage pour le réseau de Petri de la figure 2.3.

2.3.3. Vivacité :

Cette propriété caractérise la capacité d'évolution du système à partir de n'importe quel état accessible.

Définition 11 : Vivacité [DIA01]

Une transition t d'un réseau de Petri (R, M_0) est dite **vivante** si et seulement si :

$$\forall M \in A(R, M_0), \exists S \in T^*, \exists M' \in A(R, M), M [S t > M'$$

Un réseau de Petri (R, M_0) est **vivant** si toutes ses transitions sont vivantes :

$$\forall t \in T, \forall M \in A(R, M_0), \exists S \in T^*, \exists M' \in A(R, M), M [S t > M'$$

La propriété de vivacité est plus forte que le non blocage :

- Un réseau vivant modélise un système qui fonctionne sans aucun blocage, et dont toutes les actions, modélisées par le franchissement des transitions, peuvent être exécutées en permanence.
- Un réseau sans blocage modélise un système au fonctionnement permanent, sans garantir la permanence de la totalité de ses actions.

2.3.4. Etat d'accueil :

Il est possible de vérifier sur un réseau de Petri, la possibilité d'accéder à un marquage donné durant son évolution.

Définition 12 : Existence d'un état d'accueil [DIA01]

Un réseau de Petri (R, M_0) admet un **état d'accueil** M_a pour un marquage initial M_0 si pour tout marquage accessible à partir de M_0, il existe une séquence de franchissements permettant d'atteindre le marquage M_a.

$$\forall M \in A(R, M_0), \exists S \in T^*, M\ [S> M_a.$$

Autrement dit, un état d'accueil est un état accessible quelle que soit l'évolution du réseau.

La présence d'un état d'accueil sur un réseau de Petri signifie que le système modélisé peut être réinitialisé à un état donné quelle que soit son évolution.

2.3.5. Invariants :

Analyser un réseau de Petri en utilisant le graphe des marquages accessibles possède l'avantage de la simplicité, mais peut s'avérer complexe lorsque le nombre de marquages accessibles est important, et même inadapté si ce nombre est infini.

Afin de faciliter l'opération d'analyse, des **invariants** peuvent être recherchés sur un réseau de Petri. Ils permettent de caractériser certaines propriétés des marquages accessibles et des transitions franchissables et ce, quelle que soit l'évolution du marquage d'un réseau de Petri. Les invariants des réseaux de Petri sont de deux types :

Définition 13 : Composante conservative [SCO06]

Soit (R, M_0) un réseau de Petri. Un **invariant de marquage** est vérifié sur le réseau s'il existe un ensemble de place $P' \subseteq P$, un vecteur d'entiers naturels $q \in \mathbb{N}^{P'}$ appelé **vecteur de pondération**, et une constante $k \in \mathbb{N}$, tels que :

$$\forall M \in A(R, M_0), \quad \sum_{p \in P'} q_p M(p) = k$$

P' est appelée **composante conservative**, ou encore ***p*-semiflot** ou ***p*-invariant**.

Cette propriété exprime la conservation du nombre de jetons dans un réseau de Petri ou une partie de celui-ci.

Le réseau est **conservatif** si *P' = P*.

Définition 14 : Composante répétitive [SCO06]

Soit *S* une séquence de franchissements de transitions qui permet d'accéder au marquage *M'* à partir d'un marquage *M* *(M*[*S*>*M'*).

On dit que le marquage M couvre M', noté $M \geq M'$ si $\forall i, M(P_i) \geq M'(P_i)$,

et M > M' si de plus $\exists i, M(P_i) > M'(P_i)$)

S est dite **répétitive stationnaire** si : *M' = M*.

Elle est dite **répétitive croissante** si : *M' > M*.

Elle est dite **répétitive décroissante** si : *M' < M*.

Elle est dite **complète** si elle contient toutes les transitions du réseau.

Une **composante répétitive,** appelée également ***t*-invariant** ou ***t*-semiflot**, est l'ensemble *T'* des transitions de T qui composent la séquence *S*.

Cette propriété permet de repérer des comportements cycliques dans le réseau.

Le réseau est **répétitif** si *T' = T*.

L'avantage des composantes conservatives et répétitives pour les réseaux de Petri est de pouvoir étudier certains aspects du comportement d'un réseau sans avoir recours à la construction du graphe des marquages accessibles. La présence de la première donne une indication sur la bornitude du réseau, tandis que celle de la seconde est une condition nécessaire et suffisante pour que le réseau soit vivant.

2.4. Réseaux de Petri de haut niveau :

Certains systèmes réels possèdent des propriétés complexes qui ne peuvent être modélisées par des réseaux de Petri ordinaires (systèmes de production de grande taille, systèmes à contraintes temporelles. . .). Afin de remédier à ce problème, ont été introduits les **réseaux de Petri de haut niveau**. Ces réseaux permettent de raisonner sur la nature des jetons, et d'annoter le réseau à l'aide d'un langage du premier ordre.

Il existe de nombreux types de réseaux de Petri de haut niveau, parmi lesquels nous pouvons citer :

- Les réseaux de Petri Prédicats/Transitions [GEN87].
- Les réseaux de Petri algébriques [REI91]
- Les réseaux de Petri temporisés [RAM74]
- Les réseaux de Petri stochastiques [MOL82], [NAT80]

Dans le cadre de nos travaux, nous nous intéresserons aux **réseaux de Petri colorés** [JEN92].

Un réseau de Petri coloré est un réseau classique muni d'un ensemble de **couleurs**. Les couleurs sont des types qui regroupent, dans un même réseau, les jetons selon leurs natures, et permettent ainsi de les distinguer entre eux. Elles permettent de réduire considérablement la taille d'un réseau, et d'en faciliter l'expression.
Dans ce qui suit, nous présentons la définition formelle d'un réseau coloré, accompagnée d'un exemple afin d'expliquer son évolution.

Définition 15 : Réseau de Petri coloré [SCO06]

Un **réseau de Petri coloré marqué** est un uplet $<P, T, \Sigma, C, Pre, Post, M_0>$ où :

- $P = \{p_1, p_2, \ldots, p_n\}$ est un ensemble fini non vide de places.
- $T = \{t_1, t_2, \ldots, t_m\}$ est un ensemble fini non vide de transitions.
- P et T sont deux ensembles disjoints.
- $\Sigma = \{c_1, c_2, \ldots, c_r\}$ est un ensemble fini non vide de **couleurs**.
- $C : P \cup T \rightarrow \wp(\Sigma)$ est une fonction qui associe à chaque place et à chaque transition un sous-ensemble de couleurs de Σ.
- *Pre* : est la fonction d'incidence avant, définie sur $P \times T$, elle associe à chaque couple (p, t) une application $Pre(p, t) : C(t) \times C(p) \rightarrow \mathbb{N}$.

  $Pre(p, t)$ associe à chaque couple (c_t, c_p), couleurs de t et de p respectivement, un entier naturel qui représente le nombre de jetons de la couleur c_p dans la place p nécessaire pour franchir la transition t pour la couleur c_t.
- *Post* : est la fonction d'incidence arrière, définie sur $P \times T$, elle associe à chaque couple (p, t) une application $Post(p, t) : C(t) \times C(p) \rightarrow \mathbb{N}$.

  $Post(p, t)$ associe à chaque couple (c_t, c_p), couleurs de t et de p respectivement, un entier naturel qui représente le nombre de jetons de la couleur c_p à rajouter dans la place p après franchissement de la transition t pour la couleur c_t.

- M_0 : est une fonction qui associe à chaque place p une application $M_0(p) : C(p) \rightarrow \mathbb{N}$. $M_0(p)$ attribue à chacune des couleurs de la place p leur nombre de jetons initial.

Le franchissement des transitions dans les réseaux de Petri colorés consiste en une transformation des jetons des places en entrée de la transition concernée (s'ils sont compatibles avec les couleurs associées à cette dernière), afin d'obtenir d'autres jetons (de couleurs différentes si nécessaires) grâce aux fonctions d'incidence.

Définition 16 : Franchissement d'une transition [SCO06]

Une transition t est franchissable pour un marquage M et une couleur c si et seulement si :

$$\forall p \in {}^\circ t,\ \forall c_i \in C(p),\ M(p)(c_i) \geq Pre(p, t)(c, c_i)$$

C'est-à-dire que pour toutes les places p en entrée de t, pour toutes les couleurs c_i de chacune des places p, le nombre de jetons de c_i est supérieur ou égal à l'entier $Pre(p, t)(c, c_i)$.

Si une transition t est franchissable, son franchissement pour la couleur c se fait comme suit :

$$\forall p \in P,\ \forall c_i \in C(p),\ M'(p)(c_i) = M(p)(c_i) - Pre(p, t)(c, c_i) + Post(p, t)(c, c_i)$$

Exemple :

Franchissement de transition dans les réseaux de Pétri colorés (figure 2.4) :

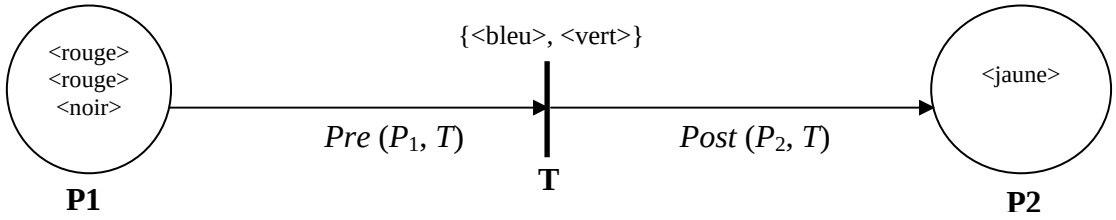

Fig 2.4 Processus de franchissement dans les réseaux de Petri colorés.

L'exemple exprime clairement

1. Les ensembles de couleurs : $C(P_1)$, $C(P_2)$ et $C(T)$, associés aux places et à la transition;
2. Et la fonction de marquage M.

Les arcs sont annotés par les fonctions d'incidence $Pre(P_1, T)$ et $Post(P_2, T)$ telles que

$Pre(P_1, T)$ (<*bleu*>, <*rouge*>) = 1

$Pre(P_1, T)$ (<*bleu*>, <*noir*>) = 0

$Pre(P_1, T)$ (<*vert*>, <*rouge*>) = 0

$Pre(P_1, T)$ (<*vert*>, <*noir*>) = 2

Post (P_2, T) (<*bleu*>, <*jaune*>) = 1

Post (P_2, T) (<*vert*>, <*jaune*>) = 1

Dans ce qui suit, nous montrons la sémantique opérationnelle du réseau coloré ci-dessus en étudiant le franchissement de la transition T par rapport aux différentes couleurs de son domaine de couleurs.

1. Il est possible de franchir la transition T pour la couleur <*bleu*> en consommant un jeton de la couleur <*rouge*> de la place P_1, et en produisant un jeton de couleur <*jaune*> dans la place P_2. En effet :
   - Pour franchir la transition T pour la couleur <*bleu*>, il faut consommer une couleur <*rouge*> de la place $P1$ (puisque *Pre* (P_1, T) (<*bleu*>, <*rouge*>) =1) ; ceci est possible à partir du marquage du réseau, car $M(P_1)$(<*rouge*>) = 2 .
   - Un tel franchissement produit un jeton de couleur <*jaune*> dans la place P_2, puisque *Post* (P_2, T) (<*bleu*>, <*jaune*>) = 1.
2. Il est impossible de franchir la transition T pour la couleur <*vert*> en consommant un jeton <*noir*>, car le nombre de jetons de couleur <*noir*> dans la place P_1 est insuffisant :

   $M(P_1)$(<*noir*>) = 1

   Pre (P_1, T) (<*vert*>, <*noir*>) = 2

Les domaines de couleurs sont généralement des produits cartésiens des classes de couleurs de base. Les fonctions de couleur de base les plus utilisées sont l'identité et la projection sur les tuples de couleurs.

La puissance d'abréviation des réseaux de Petri de haut niveau permet d'analyser des systèmes complexes. Malgré cela, une spécification pas à pas est nécessaire pour atteindre un modèle satisfaisant en suivant des techniques spécifiques, parmi lesquelles : le raffinement.

2.5. Raffinement des réseaux de Petri :

Le raffinement est une série de transformations, ajouts ou réductions de composants, appliquées sur la structure d'un modèle en réseau de Petri pour en obtenir un autre, selon certaines règles. Ceci afin d'obtenir un niveau de détails supplémentaire (en cas d'ajout), ou bien pour réduire sa taille (réduction). L'objectif de l'utilisation de cette technique est de faciliter la spécification et la vérification de propriétés de systèmes complexes.

[LAK01] introduit trois types de raffinement des réseaux de Petri :

1. **Raffinement de type :** Ce genre de raffinement se préoccupe de transformer le type de marquage du réseau sans en modifier la structure (en rajoutant, par exemple, un élément dans le tuple dans le cas des réseaux colorés). Le raffinement de type doit préserver les propriétés des types : si un type A est raffiné par un type B, les propriétés satisfaites par A doivent être satisfaites également par B.
2. **Raffinement de nœud :** Ce raffinement consiste à remplacer une place ou une transition du réseau d'origine par un sous-réseau, pour obtenir un réseau raffiné, et ce selon certaines contraintes notamment la préservation du comportement du réseau raffiné.
3. **Raffinement de sous-réseau :** Consiste à rajouter des composants (places, transitions et arcs) au réseau original afin d'obtenir un nouveau réseau. La définition de ce type de raffinement est détaillée ci-dessous.

Définition 17 : Raffinement de sous-réseau [MAY08]

Soient $R1 = <P1, T, \ldots>$ et $R2 = <P2, T2, \ldots>$ deux réseaux de Petri ordinaires. On dit que $R2$ est un raffinement de $R1$ si :

i. $P1 \subseteq P2$
ii. $T1 \subseteq T2$
iii. *{arcs de R1}* $\subseteq$ {arcs de $R2$}
iv. Aucun arc ne relie une place de $P1$ à une transition de $T2 \setminus T1$ (transitions appartenant à *T2* et n'appartenant pas à *T1*).
v. Aucun arc ne relie une transition de $T1$ à une place de $P2 \setminus P1$.
vi. Aucun arc ne relie une transition de $T2 \setminus T1$ à une place de $P1$.
vii. Le marquage initial des places du réseau $R1$ reste inchangé dans le réseau $R2$.

Exemple :

Pour illustrer la propriété de raffinement, nous reprenons l'exemple de l'atelier de coupe, simplifié (figure 2.5).

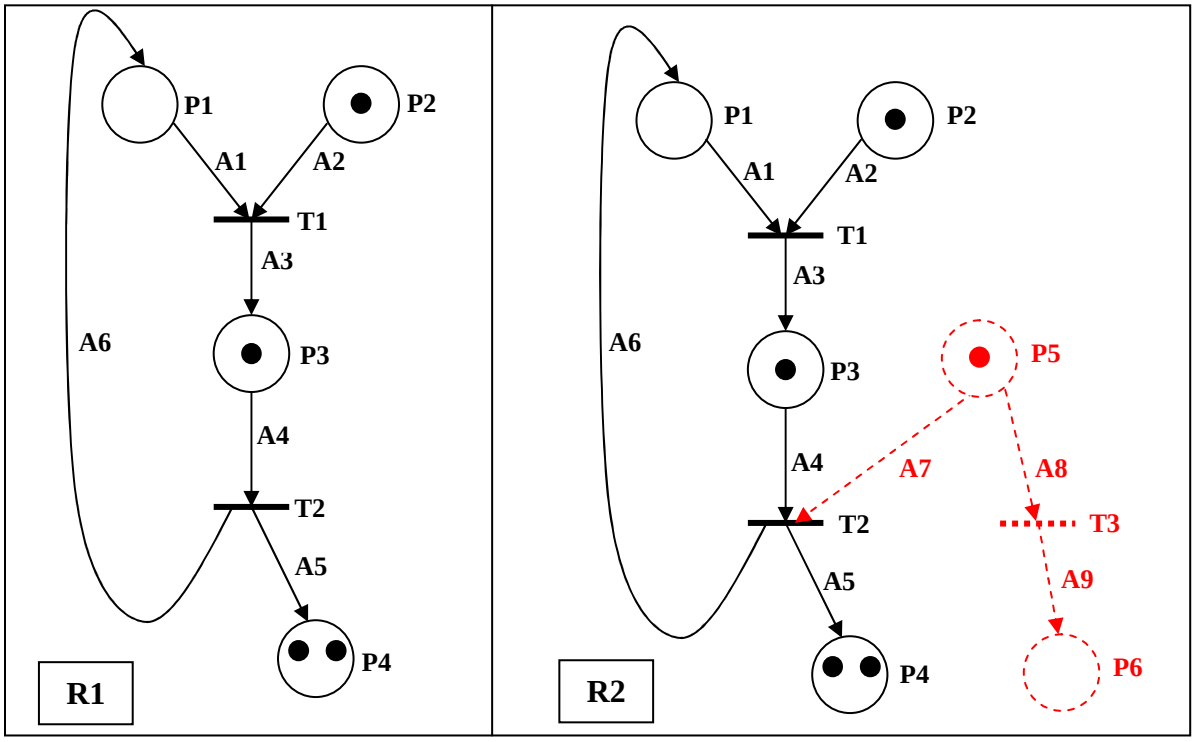

Fig 2.5 Réseau de l'atelier de coupe de bois simplifié et son raffinement.

Tout d'abord, un système complexe est modélisé dans sa forme abstraite. Une fois que ses propriétés sont vérifiées pour un prototype réduit, une étape de raffinement, qui introduit un niveau de détail supplémentaire, peut être effectuée. Le modèle ainsi raffiné est ensuite vérifié. Ces raffinements sont utilisés dans le cadre d'une approche *incrémentale* de la spécification de systèmes *complexes*. Outre les avantages du raffinement pour la spécification de systèmes, ils permettent de faciliter la vérification de propriétés.

Jusqu'à présent, le processus de raffinement de réseaux est complètement manuel et assez complexe. Toutefois, il existe certains travaux permettant de prouver automatiquement qu'un réseau est effectivement un raffinement d'un autre réseau (abstrait). A cet effet, on utilise des techniques de preuve de théorèmes, en particulier, l'assistant d'aide à la preuve Coq, pour une formalisation en Coq des définitions de raffinements ainsi que des réseaux auxquels ils s'appliquent [CHO10] [MAY08].

Conclusion :

Les réseaux de Petri possèdent deux avantages pratiques dans le domaine de la modélisation :

- Ils utilisent un formalisme graphique, ce qui contribue à la clarté du modèle, contrairement à bon nombre d'outils de conception, dont les modèles sont difficilement compréhensibles, notamment par des non-initiés.
- Ils s'appuient sur une base mathématique formelle, tant pour la définition des composant d'un réseau (notions d'ensembles, matrices, fonction…), que pour l'expression des propriétés (quantifieurs, appartenance, inclusion…).

Ce second avantage rapproche les réseaux de Petri des systèmes basés sur la preuve mathématique, ce qui permettrait leur formalisation dans ce type de système, et l'utilisation de la notion de preuve dans la vérification de leurs propriétés.

Chapitre III

Réseaux de Petri et méthodes formelles

Parallèlement aux nombreuses méthodes développées, les réseaux de Petri demeurent un outil formel largement utilisé dans le domaine de la modélisation et de la spécification de systèmes. Un de ses principaux atouts est, qu'en plus de permettre une spécification non ambiguë, ils sont facilement accessibles (même à un utilisateur peu ou non expérimenté). C'est donc tout naturellement que s'est imposée l'association entre le formalisme des réseaux de Petri et d'autres méthodes formelles de spécification.

Une première technique formelle concernée par cette association est le model checking [CLA99.a], qui a été exploité pour vérifier, de manière automatique, les propriétés dynamiques des réseaux de Petri comme dans [EVA05] ou [BER03]. Outre l'automaticité, en cas de propriété insatisfaite, un contre-exemple est exhibé. Ces avantages sont néanmoins contrebalancés par les problèmes d'explosion combinatoire des états du système.

Les autres méthodes que nous présenterons pour vérifier des propriétés des réseaux de Petri, sont des méthodes basées sur la démonstration mathématique. Nous citerons notamment, la propriété de raffinement [LAK01] sur les réseaux de Petri ordinaires [MAY08], ou les réseaux colorés [CHO10], dont la correction a été vérifiée grâce à l'outil de preuve Coq.

Une autre technique de preuve formelle, la méthode B [ABR96] a été exploitée, non seulement pour la vérification des propriétés des réseaux de Petri, mais plus globalement, pour combiner les deux outils dans une méthodologie de développement de logiciels. Cette dernière passe par une traduction des réseaux de Petri en B, comme ce fut le cas dans [BON00] et [ATT09].

Si l'approche preuve permet de dépasser les contraintes de taille, la production de preuve ne peut être entièrement automatisée et nécessite une expertise humaine non négligeable.

Il faut noter que notre choix, dans le chapitre suivant, de la preuve de théorème plutôt que le model checking, ne repose que sur notre objectif d'utiliser l'environnement Focalize que nous développons au sein du département. En effet les atouts et faiblesses des deux choix sont complémentaires :

- Le model checking est automatique, pas la preuve formelle.

- La preuve formelle peut gérer des systèmes complexes, pas le model checking.

3.1. Réseaux de Petri et model checking :

Toutes les applications exigent une certaine fiabilité dans leur conception et développement, ce qui nécessite de mettre en place des méthodes de vérification automatiques. Pour cela, des méthodes formelles de vérification sont utilisées parmi lesquelles le model checking, qui a été largement exploité dans les domaines industriels. Plusieurs algorithmes ont été implémentés donnant naissance à des Model-Checkers tels que SMV [MCM00], SPIN [HOL97], KRONOS [YOV97] et UPPAAL [LAR97].

Description informelle du Model checking :

Le model checking désigne une famille de techniques de vérification automatique de systèmes dynamiques [HAS02]. Il permet de vérifier si un système ou une abstraction de son comportement vérifie une certaine propriété, telle que l'absence d'inter-blocage (deadlock) ou la terminaison. Pour cela, il faut analyser tous les cas d'exécution possibles du système. A l'issue de cette vérification, si une propriété n'est pas satisfaite, un contre-exemple est produit permettant ainsi de fournir un cas d'exécution non conforme et ainsi localiser et corriger la source de l'erreur (voir la figure 3.1).

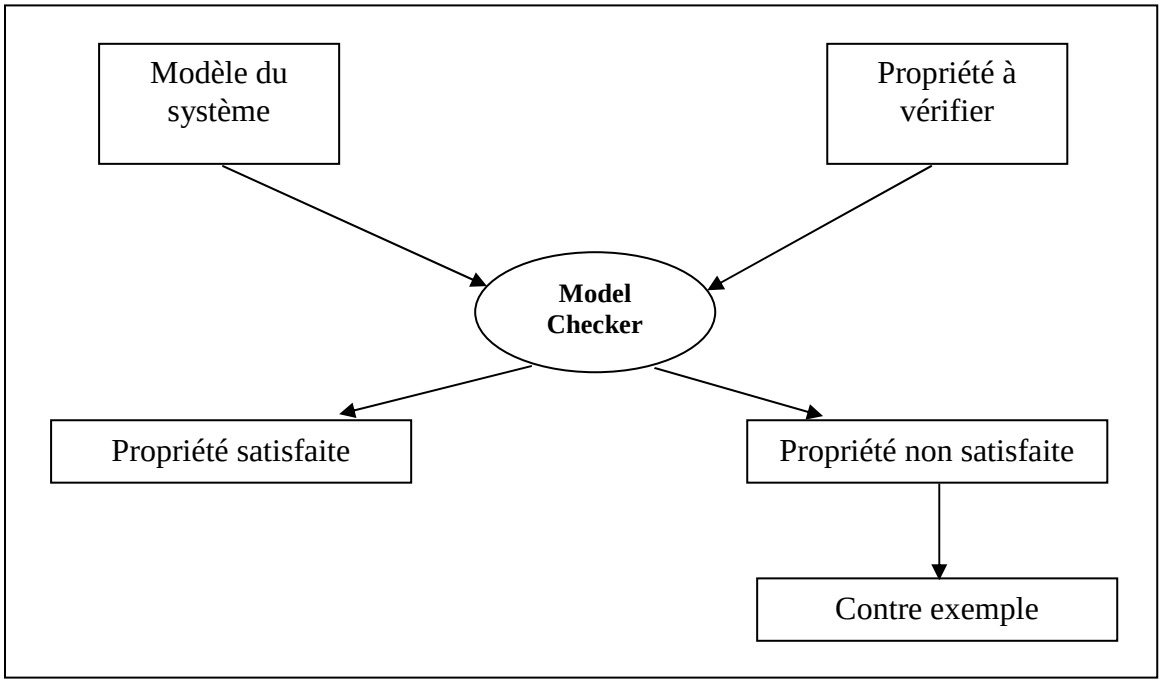

Fig 3.1 Principe du Model Checking.

Pour vérifier toutes les exécutions possibles du système, il est modélisé sous la forme d'un système de transitions à états, où chaque noeud représente un état (ou une configuration) du système, et une transition d'un état à un autre représente une évolution du système d'une configuration à une autre suite à l'exécution d'une action. Ce système de transition peut être représenté par un automate, une machine de Turing ou un réseau de Petri [DIA01]. Le model checking génère l'espace des états du système qui correspond à l'ensemble des états accessibles à partir de l'état initial.

D'autre part, pour formaliser, sans ambiguïté, les propriétés à vérifier, le model checking utilise des langages puissants tels que les logiques temporelles. Parmi ces logiques, on citera LTL (Linear Temporal Logic [ART10]) qui permet de considérer une propriété tout le long d'un chemin d'exécution de l'espace d'états du système, ou bien la logique CTL (Computation Tree Logic [ART10]) qui prend en charge des propriétés sur plusieurs chemins d'exécution.

Limitations du model Checking

En pratique, la limitation majeure du model checking est la taille gigantesque des systèmes de transitions, on parle, dans ce cas, d'*explosion combinatoire* du nombre d'états du système [VAL98]. Ce phénomène possède deux sources distinctes : la taille du système de transitions augmente exponentiellement, d'une part, avec la taille du système à considérer, d'autre part, avec le nombre de composants du système dans le cas où il est concurrent. Des techniques spécifiques ont été développées pour limiter chacune de ces sources potentielles d'explosion. En l'occurrence, le model checking *symbolique* [BUR90] qui propose une gestion et représentation efficaces pour les états, ou bien les *ordres partiels* [CLA99.b] qui utilisent des relations d'équivalence pour masquer les entrelacements des actions dues à l'indéterminisme réduisant ainsi l'espace d'états.

En conclusion, même s'il reste largement utilisé, le model checking souffre du problème d'explosion combinatoire de l'espace des états lorsqu'il s'agit de systèmes de grande taille. Dans notre cas, l'utilisation des systèmes de preuve formelle s'avère plus pertinente.

3.2. Réseaux de Petri et Coq :

La preuve formelle, plus précisément la preuve de théorèmes, nécessite d'abord d'énoncer les propositions à démontrer dans un système de déduction [MON03].

Bien qu'on parle souvent de « preuve automatique », on devrait plutôt parler de « preuve assistée par ordinateur » utilisant des « systèmes d'aide à la démonstration » largement inspirés de l'isomorphisme de Curry-Howard [HOW80], même s'il est indéniable que les preuves tendent à être automatisées (Zenon dans Focalize).

[CHO10] propose un travail qui vérifie une propriété de raffinement des réseaux de Petri colorés par le système d'aide à la preuve Coq. Le type de raffinement concerné est le raffinement de type [LAK01] (section 2.5). Ce travail fait suite à celui de [MAY08] qui prend en charge le raffinement de sous-réseau sur les réseaux de Petri ordinaires (section 2.5), en utilisant le même système Coq.

La démarche proposée concerne une sous classe des réseaux de Petri colorés appelée *réseaux symétriques* [JEN08], qui prennent en charge les types suivants pour représenter les couleurs : types énumérés, booléens, entiers, tuples et la combinaison de tous ces types. Ces derniers sont spécifiés en Coq de manière naturelle, ce qui facilite le passage des réseaux de Petri en un formalise Coq.

A partir d'un modèle initial abstrait, exprimé en réseau de Petri coloré, des modifications sont apportées au fur et à mesure, jusqu'à atteindre un niveau de description satisfaisant. La technique de raffinement s'inscrit donc dans une spécification incrémentale.

Le raffinement de type permet d'enrichir et de raffiner les domaines de couleurs des modèles intermédiaires (en rajoutant par exemple des couleurs à un tuple de couleurs). Les domaines de couleurs du réseau raffiné final doivent respecter des règles de sous-typage avec les domaines de couleurs du réseau initial. Ces règles de sous-typage sont inspirées de la programmation orientée objet [LIS93].

[CHO10] propose de vérifier le raffinement de type comme suit (figure 3.2) :

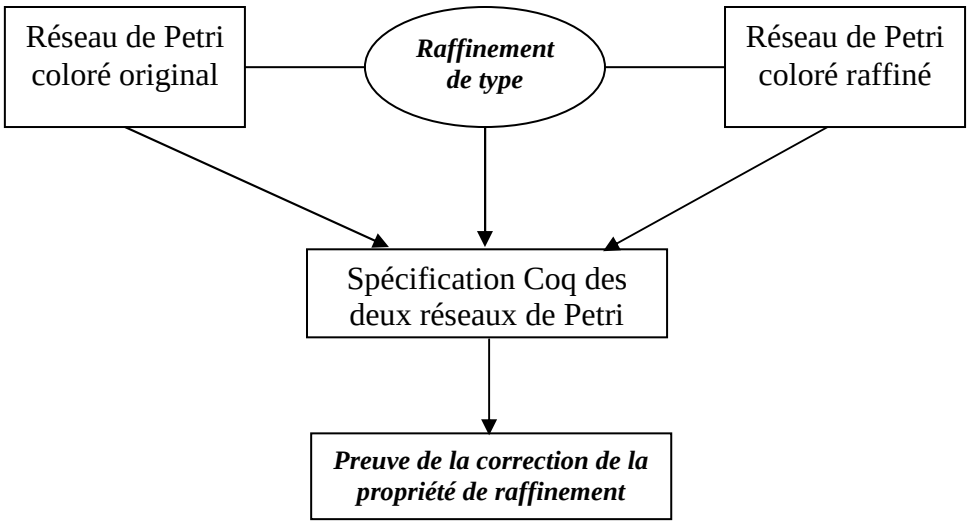

Fig 3.2 Processus de preuve de la propriété de raffinement de type par Coq.

- Spécification des deux réseaux de Petri colorés (réseau d'origine et son raffinement) en Coq, telle que : les couleurs sont traduites en types inductifs Coq, les composants des réseaux (places, transitions et arcs) sont représentés par des chaînes de caractères indexées par des entiers, et les ensembles de composants des réseaux sont traduits sous la forme de listes.
- L'égalité des ensembles de composants deux à deux, entre le réseau original et le réseau raffiné est assurée par une fonction d'égalité des listes spécifiée en Coq. Des fonctions Coq définissent les règles sur le sous-typage des couleurs entre les deux réseaux.
- Chacune des clauses de la propriété de raffinement : égalité des ensembles de composants, et propriété de sous-typage des couleurs, est dérivée en une propriété Coq, et prouvé grâce à ce dernier.

L'approche présentée dans [CHO10] possède l'avantage de la généricité de la preuve fournie à la propriété de raffinement, qui peut être appliquée à tout exemple de réseau de Petri. Toutefois, la spécification d'un réseau de Petri en Coq n'est pas générique, elle dépend de l'exemple étudié, ce qui peut s'avérer fastidieux lorsqu'il s'agit d'un réseau de grande taille.

3.3. La méthode B :

La **méthode B** a été inventée au milieu des années 80 par J.R Abrial [ABR96]. Comme la notation Z [SPI92], aussi créée par J.R Abrial, elle est basée sur la théorie des ensembles. Mais si Z est restée limitée à la spécification, la méthode B englobe le processus complet de développement d'un logiciel, de la spécification abstraite au code exécutable, tout en s'assurant de la correction de l'implantation vis-à-vis de la spécification, en utilisant la preuve formelle.

La méthode B est mise en œuvre par un système appelé l'atelier B ou le langage B [CLE03], ce qui signifie qu'elle a dépassé le stade de l'expérimentation. D'ailleurs, elle a été exploitée dans de nombreux domaines industriels, plus particulièrement sur des systèmes où l'aspect sécuritaire est critique. Les applications industrielles de la méthode B appartiennent principalement aux domaines des transports ferroviaires, la plus spectaculaire d'entre elles étant celle réalisée sur le projet Meteor [BEH99]. Meteor est l'acronyme de **MET**ro **E**st **O**uest **R**apide, c'est le nom de projet donné à la « ligne 14 » du métro de Paris.

En 1992, la RATP a choisi la société MTI (Matra Transport International) pour prendre en charge le développement d'un système de pilotage automatique d'un métro, en utilisant la méthode B. Ce système possède des contraintes de sécurité critiques car il s'occupe : du démarrage et arrêt du métro, du contrôle de la vitesse et de l'ouverture et fermeture des portes.

Un processus formel complet en méthode B a été appliqué pour la réalisation du logiciel de pilotage automatique et l'assurance de sa sécurité. Ce processus englobe les étapes suivantes :

- élaboration du cahier des charges,
- construction d'un modèle abstrait en B,
- construction d'un modèle concret en B0 (B0 sous-langage du langage B)
- et enfin code exécutable en ADA.

Au final, le nombre de lignes de code générés pour la réalisation de ce projet était de 87000 lignes, et le processus a été repris pour de nombreux autres projets similaires : métros de Barcelone, Madrid, New Delhi, Le Caire, navette de l'aéroport Charles De Gaulle de Paris…
Depuis, le champ d'action de la méthode B a été étendu à d'autres domaines : bases de données et traitement de l'information [HOA95], [MAT99], les protocoles de communication [BIE96], [LAN98] ou les systèmes distribués [BUT96].

Principaux concepts de la méthode B :

Une spécification B est composée d'un ensemble de modules appelés *machines abstraites*, interagissant entre elles grâce à des mécanismes particuliers. Une machine abstraite modélise une partie d'un système décrit par un ensemble de données ou variables, par des opérations qui modifient leur état et par des propriétés exprimées par des prédicats du premier ordre. Toutes ces informations sont organisées en *clauses* qui définissent :

- ***VARIABLES* :** liste des variables utilisées dans la machine abstraite. Elles peuvent rester abstraites jusqu'à la phase d'implantation.
- ***INVARIANT :*** prédicats portant sur les variables de la machine, et qui fixent les propriétés que leurs valeurs effectives doivent toujours respecter.
- ***CONSTANTS :*** éléments immuables du système modélisé (ex : limite supérieure d'une variable).
- ***SETS :*** ensembles de variables ou de constantes de la machine abstraite, soit de façon explicite (ex : ensembles énumérés), soit par construction (ex : les entiers n, n> n1 et n< n2).
- ***PROPERTIES :*** contraintes introduites sur les ensembles et les constantes.

- ***INITIALISATION :*** état initial du système (ex : valeur initiale d'une variable).
- ***DEFINITIONS :*** cette clause est définie comme une abréviation, éventuellement paramétrée, pour un prédicat, une expression ou une substitution. Une définition peut être utilisée dans les autres clauses de la machine abstraite. Chaque utilisation d'une définition est remplacée par le texte correspondant, où les paramètres formels sont remplacés par les paramètres réels.
- ***OPERATIONS :*** fonctions et procédures qui agissent sur les constantes et variables de la machine abstraite. Chaque opération est spécifiée par une substitution généralisée [PAU02].

Exemple :

Machine abstraite définissant la notion de point.

```
MACHINE point

VARIABLES x, y
INVARIANT x ∈ 0..1024 ∧ y ∈ 0..768
INITIALISATION x,y := 0,0

OPERATIONS
v ← abs = v := x ;
v ← ord = v := y ;
move (dx, dy) =
     PRE x+ dx ≤ 1024 ∧ y+ dy ≤ 768
     THEN (x,y) := (x+ dx, y+ dy)
     END

END
```

Fig 3.3 Exemple d'une machine abstraite modélisant la notion de point.

- *x* et *y* sont deux variables appartenant aux intervalles [0, 1024] et [0, 768].
- *x* et *y* sont initialisées à la valeur zéro.
- Le symbole (:=) représente la substitution d'affectation.
- L'opération *move* définit le changement de position du point, elle est définie par une substitution pré-conditionnée [PAU02].

Les machines abstraites initiales subissent des transformations successives et progressives durant le développement appelées *raffinements*. Ces derniers sont des machines dans lesquels les clauses sont affinées (d'autres variables ou types sont rajoutées, les indéterminismes dus aux substitutions sont levés …).

Le dernier niveau de raffinement permet d'obtenir un modèle exécutable constitué *d'implantations*, des machines qui ne peuvent plus être raffinées. Dans ce niveau, il est procédé notamment aux choix des structures de données (tableaux pour implémenter les ensembles, affectations pour les substitutions …).

La preuve en B :

A différents niveaux de la spécification, des obligations de preuve sont générées systématiquement pour chaque composant, dont il faut prouver la véracité afin de s'assurer de la validité du modèle.

Une *obligation de preuve* [CLE03] est un lemme mathématique constitué d'une liste de prédicats appelés *hypothèses* et d'un prédicat appelé *but*, qui doit être prouvé sous ces hypothèses.

Parmi les cas de génération d'obligations de preuve :

- Chaque variable doit être contrainte pour appartenir à un ensemble (ex : x et y appartiennent aux intervalles [0, 1024] et [0, 768]).
- L'invariant doit être satisfait par l'initialisation.
- L'invariant doit être préservé après chaque application d'une opération de la machine ou de son raffinement.

En prenant pour exemple ce dernier cas de figure, l'opération agit comme un transformateur de prédicat par la substitution *S* qui la définit. Ce mécanisme génère des obligations de preuve dont l'hypothèse est l'invariant *I* de la machine courante, et le but est le prédicat obtenu suite à la transformation [*S*] *I*. L'obligation est de la forme :

$$I \Rightarrow [S]\, I.$$

En prenant l'exemple de la substitution pré-conditionnée vue dans l'exemple du point, nous obtenons une obligation de preuve de la forme :

$$I \Rightarrow [PRE\ P\ THEN\ S\ END]\, I.$$

Sémantiquement, la substitution pré-conditionnée est définie comme suit:

$$[P|S]\, I \Leftrightarrow P \wedge [S]\, I$$

Nous obtenons au final l'obligation de preuve à démontrer :

$$I \Rightarrow P \wedge [S]\, I$$

Lors de l'appel d'une opération possédant une précondition PRE *P THEN S END*, l'application de l'opération correspond à la preuve de *P*, et à l'application de la substitution *S*. Si la condition n'est pas prouvée, la substitution ne se termine pas, c'est-à-dire que le

comportement décrit par une substitution pré-conditionnée n'est garanti que si, dans son contexte d'utilisation, la précondition est vraie.

Sur l'exemple de la figure 3.2, une obligation de preuve est générée, et qui vérifie que les variables *x* et *y* appartiennent toujours aux mêmes intervalles après l'exécution de l'opération *move* :

$$x \in 0..1024 \wedge y \in 0..768 \Rightarrow$$
$$(x + dx \leq 1024 \wedge y + dy \leq 768) \wedge$$
$$[(x,y) := (x + dx, y + dy)]\ x \in 0..1024 \wedge y \in 0..768$$

La preuve en B est déléguée à un assistant de preuve propre à l'atelier B, dont le manuel d'utilisation est [CLE08].

Une présentation exhaustive des substitutions utilisées, et des obligations de preuve générées tout au long d'une spécification B figure dans [CLE03] et [PAU02].

3.4. Réseaux de Petri et méthode B :

Même si la méthode B couvre tout le cycle de développement de logiciel, un modèle en B reste difficilement compréhensible par un non initié, contrairement à un modèle graphique. Dans cette optique, [BON00] propose une méthodologie de développement basée sur la combinaison de deux méthodes formelles : les réseaux de Petri et la méthode B. Cette méthode a pour but de faciliter le passage du cahier des charges aux spécifications. Elle englobe les étapes suivantes (figure 3.4) :

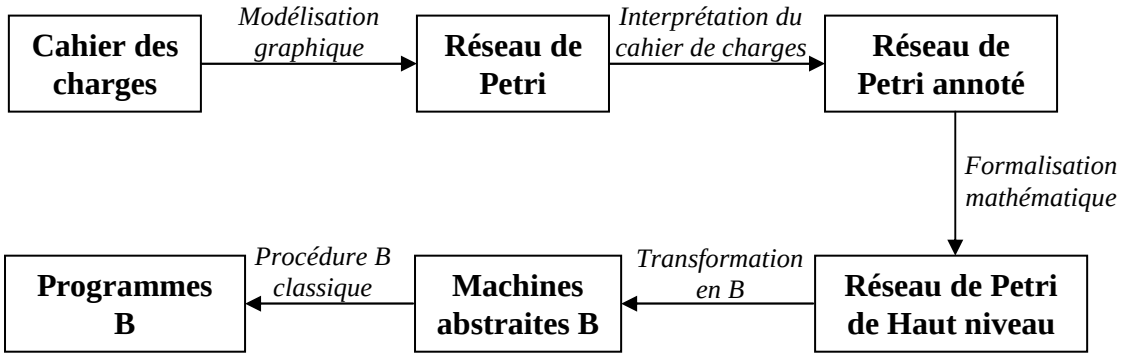

Fig 3.4 Méthodologie de développement basée sur les réseaux de Petri et la méthode B.

1. La représentation graphique du cahier de charges sous forme d'un réseau de Petri ;
2. Annotation du réseau en langage naturel ;

3. Transformation des annotations pour obtenir un réseau de Petri de haut niveau. Le choix de ces derniers s'explique par les limites des réseaux de Petri ordinaires pour représenter les problèmes complexes, notamment pour cause de taille trop importante ;
4. Transformation du réseau obtenu en machines abstraites B ;
5. Poursuite du développement par une procédure B classique.

Afin d'illustrer la démarche de transformation d'un réseau de Petri en B, nous reprenons l'exemple utilisé dans [BON00]. Il modélise un réseau ferroviaire (figure3.5), et est décrit comme suit :

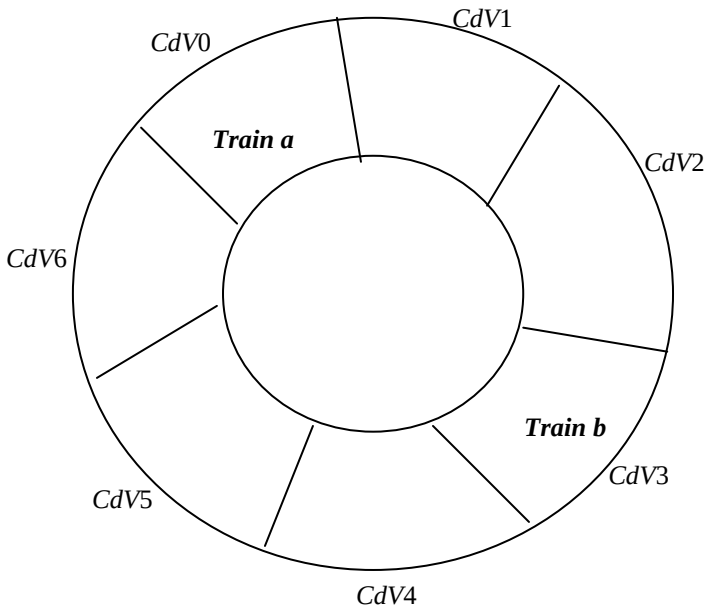

Fig 3.5 Représentation de l'exemple du réseau ferroviaire.

- Le système est un réseau de sept circuits de voies élémentaires, disposées en boucle fermée et numérotés de 0 à 6.
- Des trains de deux types (a et b) circulent sur le réseau dans un sens donné.

Deux contraintes de sécurité sont rajoutées :

- Il ne peut pas y avoir deux trains sur le même circuit de voie.
- Il doit toujours y avoir un circuit de voie libre entre deux trains, c'est à dire entre deux circuits de voie occupés.

A partir de cette description informelle, et après les étapes de représentation graphique et d'affinage du modèle, nous obtenons un réseau de Petri coloré (figure 3.6), constitué de :

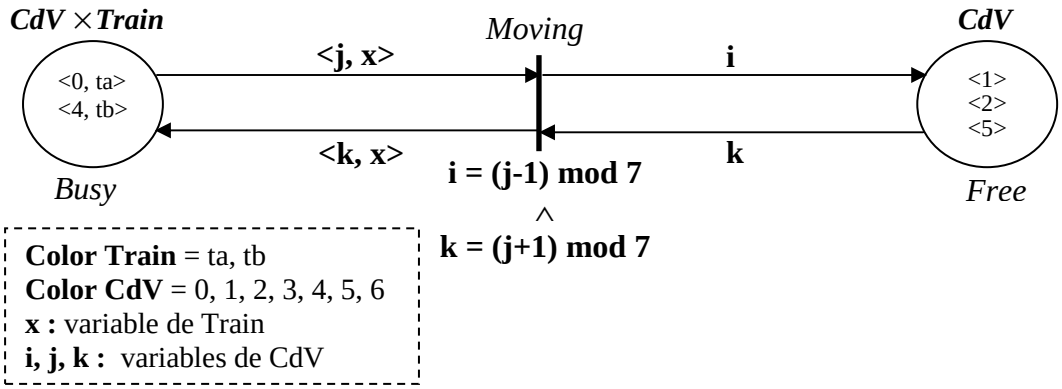

Fig 3.6 Réseau de Petri coloré modélisant le réseau ferroviaire.

- Une place *Free* : contenant des jetons de la couleur *CdV*, des entiers naturels représentant les numéros de circuits de voie libres.
- Une place *Busy* : contenant des paires de type *CdV*×*Train*, pour modéliser les circuits de voie occupés par un type de train.
- Une transition *Moving* : représente l'action de libération et d'occupation d'un circuit de voie par un train. Une garde est associée à la transition *Moving*, qui permet de calculer les circuits de voie libérés et occupés après le franchissement. Ce calcul est réalisé grâce à la fonction modulo pour respecter la contrainte du nombre de circuits de voie disponibles sur le réseau ferroviaire (7 circuits).

Le marquage du réseau de Petri est représenté par des multi-ensembles [JEN92].
Un multi-ensemble *m* basé sur un ensemble non vide *s* est un ensemble fini d'éléments appartenant à *s*, et qui peuvent avoir plusieurs occurrences dans *m*.
Pour les besoins de la transformation, les fonctions qui manipulent les multi-ensembles ont été spécifiées en B :

- *Ms_empty*(*s*) : représente le multi-ensemble vide.
- *Ms_In* (*e*, *ms*, *s*) : fonction d'appartenance à un multi-ensemble.
- *Ms_Subset* (*ms*1, *ms*2, *s*) : fonction d'inclusion de deux multi-ensembles.
- *Ms_Add* (*ms*1, *ms*2, *s*) : fonction de rajouts des éléments d'un multi-ensemble vers un autre multi-ensemble.
- *Ms_Less* (*ms*1, *ms*2, *s*): fonction de suppression des éléments d'un multi-ensemble depuis un autre multi-ensemble.

Un algorithme de transformation est appliqué au réseau de Petri obtenu, et qui définit des règles générales pour formaliser le passage des réseaux de Petri vers B. Le réseau de Petri représentant le réseau ferroviaire est transformé en une machine abstraite B comme suit :

- A toute place du réseau est associée une variable de la machine abstraite, qui représente son marquage :

```
VARIABLES
State_Busy, State_Free
```

- Les types de trains sont modélisés par un ensemble énuméré :

```
SETS Trains = {ta, tb}
```

- Les circuits de voie sont modélisés par un ensemble défini par construction (*CdV*) :

```
PROPERTIES CdV = {elt | elt ∈ NAT ∧ elt≥0 ∧ elt≤6}
```

- Les couleurs de place *Busy* sont représentées par un produit cartésien *Trains* × *CdV*, tandis que celle de la place *Free* est l'ensemble *CdV* :

```
DEFINITIONS
ColorF_Busy == CdV × Trains;
ColorF_Free == CdV ;
```

- Des expressions relatives aux arcs définissent la transformation des jetons en entrée et en sortie lors du franchissement d'une transition, ((*j* ⤇ *x*) représente le couple (*j*, *x*)) :

```
DEFINITIONS
ArcExp_Busy_Moving == MS_Empty(ColorF_Busy)← {(j ⤇ x) ⤇ 1};
ArcExp_Free_Moving == MS_Empty(ColorF_Free)← {k ⤇ 1};
ArcExp_Moving_Busy == MS_Empty(ColorF_Busy)← {(k ⤇ x) → 1};
ArcExp_Moving_Free == MS_Empty(ColorF_Busy)← {i ⤇ 1};
```

- Le marquage initial de chaque place est défini dans la clause INITIALISATION. L'initialisation est réalisée en rajoutant des éléments au multi-ensemble vide basé sur l'ensemble des couleurs de la place :

```
INITIALISATION
State_Busy := MS_Empty(ColorF_Busy)←{(0⤇ta)⤇1,(4⤇tb)⤇ 1};
State_Free := MS_Empty(ColorF_Free)←{1 ⤇ 1, 2 ⤇ 1, 5 ⤇ 1};
```

- La garde de calcul du numéro de circuit de voie exprimée sur la transition *Moving* est traduite en un prédicat B dans la clause DEFINITIONS :

```
DEFINITIONS
Guard_Moving == i= ((j-1)mod7) ∧ k= ((j+1)mod7);
```

- La transition *Moving* est associée à des variables dans la machine abstraite, qui apparaissent dans sa garde, ainsi que dans les expressions d'arcs : *i*, *j*, *k* sont des circuits de voie, et *x* est un type de train :

```
DEFINITIONS
Var_Moving == i, j, k, x;
Type_Var_Moving == i∈CdV ∧ j∈CdV ∧ k∈CdV ∧ x∈Trains;
```

- La validation de la transition est un prédicat B défini dans la clause DEFINITIONS :

```
DEFINITIONS
Enabled_Moving == ∃Var_Moving.(Type_Var_Moving ∧ Guard_Moving
            ∧ MS_Subset(ArcExp_Free_Moving, State_Free,
            ColorF_Free)
            ∧ MS_Subset(ArcExp_Busy_Moving, State_Busy,
            ColorF_Busy));
```

- Le franchissement de la transition *Moving* est spécifié comme une opération dans la machine abstraite. Elle prend en entrée les variables de la transition, vérifie la garde et le prédicat de validation, et modifie le marquage du réseau :

```
OPERATIONSOp_Moving =
SELECT Enabled_Moving
THEN
ANY Var_Moving
WHERE
MS_Subset(ArcExp_Free_Moving, State_Free, ColorF_Free)
∧ MS_Subset(ArcExp_Busy_Moving, State_Busy, ColorF_Busy))
∧ Type_Var_Moving ∧ Guard_Moving
THEN
State_Busy :=
MS_Add(MS_Less(State_Busy, ArcExp_Busy_Moving, ColorF_Busy),
ArcExp_Moving_Busy, ColorF_Busy)
State_Busy :=
MS_Add(MS_Less(State_Free, ArcExp_Free_Moving, ColorF_Free),
ArcExp_Moving_Free, ColorF_Free);
```

- Un exemple d'expression d'une propriété en B, la contrainte de sécurité exprimée sur le réseau de Petri, garantissant qu'il y'a toujours un circuit de voie libre entre deux trains sur le réseau ferroviaire, est traduite en un invariant B :

```
INVARIANT
∀(i,j).(i∈CdV ∧ j∈CdV ∧
MS_In((i → ta), State_Busy, ColorF_Busy) ∧
MS_In((j → tb), State_Busy, ColorF_Busy) =>
(j-i) mod7 >1 ∧ (j-i) mod7 < 6;
```

Outre la définition de la transformation des réseaux de Petri en B, [BON00] a établi et prouvé une équivalence comportementale entre les deux méthodes :

- Entre le marquage du réseau et l'état de la machine abstraite ;

  Soient M le marquage d'un réseau de Petri coloré, $\{p_1, \ldots, p_n\}$ l'ensemble des places du réseau et *MA* la machine B obtenue par traduction de ce réseau. Si *State* (*M*) est le prédicat qui dénote l'état de la machine *MA*, alors

  $(State\_p_1 = M(p_1)) \wedge \ldots \wedge (State\_p_n = M(p_n))$

- Entre la sensibilisation d'une transition du réseau et le prédicat de sensibilisation correspondant dans la machine abstraite :

  Une transition *t* du réseau de Petri est sensibilisée pour un marquage *M* si et seulement si *State* (*M*) $\Rightarrow$ *Enabled_t* est valide dans la machine abstraite correspondante.

- Entre le franchissement d'une transition *t* du réseau et l'application de l'opération de franchissement correspondante à *t* dans la machine abstraite :

  M [*t*> *M'* <=> *State* (*M'*) = [*op_t*] *State* (*M*)

A partir de ce résultat, il est possible de déduire que :
I est un invariant du réseau de Petri si et seulement si *I*' (sa transformation en B) est un invariant de la machine abstraite correspondante. Ceci permet d'utiliser les techniques de preuve des réseaux de Petri et les prouveurs associés à la méthode B de manière complémentaire dans la vérification des propriétés.

Conclusion :

La combinaison de différentes méthodes formelles dans le développement de logiciels peut être motivée par le besoin d'une technique spécifique à un niveau particulier du développement. Elle permet de bénéficier des avantages de chacune des méthodes dans le processus de développement.
Dans le cas qui nous intéressent, les réseaux de Petri permettent une meilleure compréhension du modèle grâce à la représentation graphique, tandis que des méthodes de preuve formelle comme B couvrent le cycle de développement complet, et nous permettent au final d'obtenir un code exécutable.

Chapitre IV

Des réseaux de Petri vers FoCaLiZe

Dans ce chapitre, nous allons présenter une démarche de transcription des réseaux de Petri ordinaires dans le système FoCaLiZe. L'objectif de cette démarche est d'obtenir, à partir d'un modèle en réseau de Petri, une spécification FoCaLiZe, et d'utiliser les techniques de preuve fournies par FoCaLiZe pour vérifier des propriétés relatives aux réseaux de Petri.
En effet, les mécanismes de spécifications qu'offre l'environnement FoCaLiZe permettent la transcription des composants des réseaux de Petri (place, transition et rac) en espèces FoCaLiZe, ensuite nous utilisons la bibliothèque des ensemble finis de FoCaLiZe pour générer des espèces qui modélisent les ensembles de composants (ensemble des places, ensemble des arcs et ensemble des transitions) afin de capturer la structure globale d'un réseau de Petri. Enfin, nous introduirons la propriété de raffinement des réseaux de Petri (section 2.5), que nous vérifierons sur un cas concret.

4.1. Démarche générale de transcription :

Lors de la traduction d'un réseau de Petri vers l'environnement FoCaLiZe, nous considérons un réseau de Petri ordinaire composé d'un ensemble des places, d'un ensemble d'arcs et d'un ensemble des transitions. Ces composants sont décrits comme suit :

- Une place est caractérisée par son nom, P_1, P_2, … P_n, et par son marquage, un entier, désigne le nombre de jetons de la place à un instant donné.
- Une transition est caractérisée par son nom, T_1, T_2, … T_m.
- Enfin un arc est caractérisé par son nom, A_1, A_2, … A_k, son poids (un entier qui conditionne le franchissement de la transition en relation) et caractérisé aussi par l'identifiant de la place et de la transition qu'il relie.

Les grandes lignes de notre démarche sont les suivantes :
Une espèce FoCaLiZe sera dérivée pour modéliser une place. Le type support de l'espèce dérivée doit être une paire (exprimée sous forme d'un produit) composée du nom de la place et de son marquage pour représenter l'état de la place. Les opérations pour manipuler une place, telles que la création d'une nouvelle place, la modification de son marquage…, sont reportées vers des fonctions au niveau de l'espèce correspondante.

Nous procèdons de manière similaire avec les transitions et les arcs. Une espèce sera dérivée pour modéliser une transition, sa représentation est de type *string* qui sert à identifier la transition. Une espèce est dérivée pour modéliser un arc, sa représentation est un triplet pour représenter son poids et la place et la transition à ses extrémités. Les opérations pour récupérer le poids d'un arc, récupérer sa place ou sa transition sont des fonctions au niveau de l'espèce concernée.

Puis trois autres espèces sont dérivées pour modéliser les ensembles des composants de base : ensemble des places, ensemble des transitions et ensemble des arcs. Ces espèces permettent de regrouper les éléments semblables du même réseau dans le même groupe.

Enfin, Une espèce est dérivée pour modéliser entièrement un réseau de Petri. Cette espèce sert à regrouper toutes les espèces précédentes dans une même structure en utilisant les mécanismes d'héritage et de paramétrage de FoCaLiZe. Les propriétés à vérifier sur un réseau de Petri, telle que la propriété de raffinement, sont exprimées sous formes de propriétés FoCaLiZe au niveau de cette espèce.

Finalement, pour permettre concrètement aux utilisateurs la création et la manipulation des réseaux de Petri, chacune des espèces précédentes est implémentée par une collection.

Nous pouvons résumer notre démarche de transcription dans les points suivants :

1. Chacun des composants du réseau (place, transition ou rac) sera transcrit en une espèce FoCaLiZe, dont le type support devra représenter les attributs du composant : une place (son nom et son marquage), une transition (son nom), et un arc (son nom et son poids).
2. Afin d'établir le lien entre les composants défini dans les réseaux de Petri par les nœuds, nous utiliserons le paramétrage, l'espèce représentant les arcs sera paramétrée par les espèces représentant les places et les transitions.
3. Trois autres espèces, qui héritent de la bibliothèque des ensembles finis de FoCaLiZe, seront définies pour regrouper les composants dans des ensembles de composants : ensemble de places, ensemble de transitions et ensemble d'arcs. Ceci nous permettra de profiter des méthodes et propriétés définies dans les ensembles finis dans notre spécification.
4. Une espèce globale regroupera toutes les autres espèces par paramétrage, pour modéliser la structure des réseaux de Petri.
5. Les clauses de la propriété de raffinement seront dérivées en propriétés FoCaLiZe, et introduites dans les différentes espèces pour les prouver.

6. Toutes les espèces seront implémentées par des collections, qui permettront de créer les éléments d'un exemple concret et de les manipuler grâce aux méthodes des espèces.

Le processus complet, depuis la réalisation de la spécification jusqu'à la preuve, est illustré dans la figure 4.1.

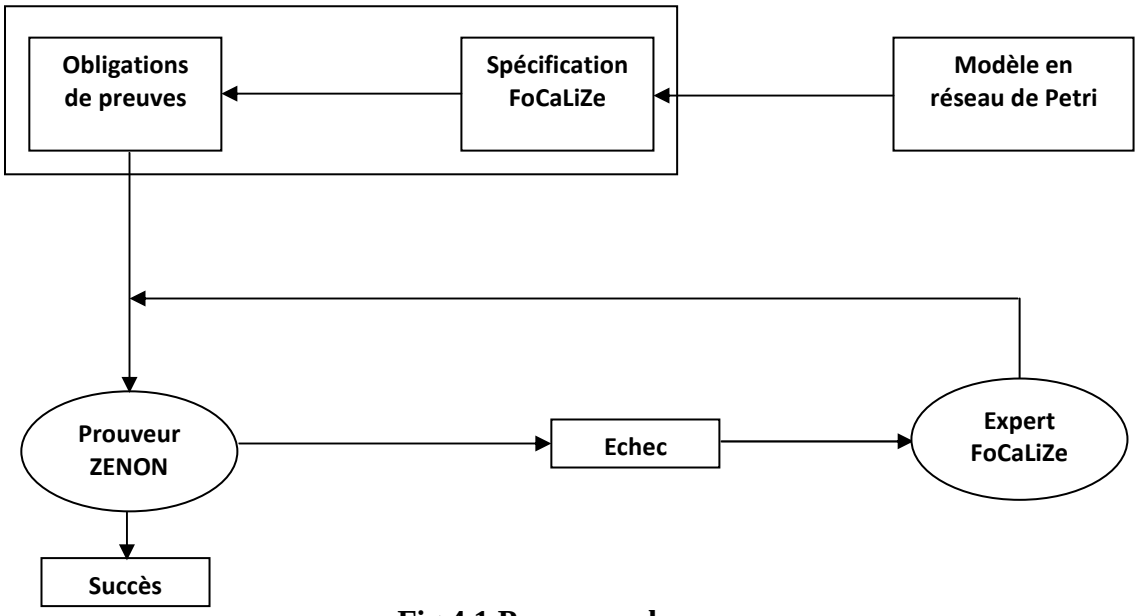

Fig 4.1 Processus de preuve.

Avant de détailler notre démarche, nous présentons quelques éléments qui seront utiles dans notre spécification.

4.2. Compléments FoCaLiZe :

Pour les besoins de notre spécification, nous introduisons des développements appartenant à des bibliothèques standards de FoCaLiZe.

4.2.1. Bibliothèque *Set_orders* :

Dans cette bibliothèque, nous avons besoin de l'espèce *Setoid* qui est spécifiée comme suit:

```
species Setoid = inherit Basic_object;

      signature equal : Self -> Self -> bool ;
      signature element : Self ;
      let different (x, y) = ~~ !equal (x, y) ;

      property equal_reflexive : all x : Self, equal (x, x) ;
      property equal_symmetric : all x y : Self,
      equal (x, y) -> equal (y, x) ;
      property equal_transitive : all x y z : Self,
      equal (x, y) -> equal (y, z) -> equal (x, z) ;
      . . .
end ;;
```

En effet, l'espèce *Setoid* modélise une collection d'éléments satisfaisant une relation d'équivalence au-dessus de l'égalité. L'espèce *Ensembles_finis* que nous allons présenter juste après est paramétrée par l'espèce *Setoid*, c'est-à-dire qu'elle permet de construire des ensembles d'éléments de genre *Setoid*. Les espèces qui modélisent les composants place, transition et arc vont hériter de l'espèce *Setoid*. Cela nous permettra de substituer le paramètre *Setoid* de l'espèce *Ensembles_finis* par l'espèce qui modélise une place, transition ou arc pour définir l'ensemble des places, l'ensemble de transitions ou l'ensemble d'arcs.

4.2.2. Bibliothèque *Ensembles_finis* :

Cette spécification définit les notions mathématiques d'ensembles finis et de listes.

```
species Ensembles_finis (A is Setoid) = inherit Partial_order ;
      signature vide : Self;
      signature sous_ensemble : Self -> Self -> bool;
      signature union : Self -> Self -> Self;
      signature inter : Self -> Self -> Self;
      signature diff : Self -> Self -> Self;
      . . .
end ;;
species Liste (A is Setoid) = inherit Ensembles_finis (A);
      representation = list (A);

      let vide : Self = [];
      let diff (e1 : Self, e2 : Self) =
          let rec aux (l, accu) =
            match l with
            | [] -> accu
            | x :: l -> aux (l, remove_element (x, accu)) in
      aux (e2, e1) ;
      . . .

end ;;
```

- L'espèce *Ensembles_finis* fournit les spécifications et les preuves de toutes les propriétés sur les fonctions relatives aux ensembles finis : intersection, inclusion, appartenance, union… .

  Voici par exemple la spécification de l'opération *inter* qui retourne l'intersection de deux ensembles en paramètre:

```
      signature inter : Self -> Self -> Self;

      property inter_spec:
          all s1 s2 : Self, all x : A,
          est_element (x, inter (s1, s2)) <->
         (est_element (x, s1) /\ est_element (x, s2));
```

Dans l'exemple *est_element* exprime l'appartenance d'un élément à un ensemble.

Le fait que l'espèce *Ensembles_finis* soit paramétrée par *Setoid*, permet dans FoCaLiZe de construire tous les ensembles dont les éléments descendent de l'espéce *Setoid*. La représentation de l'espèce *Ensembles_finis* est abstraite, ce qui donne beaucoup de flexibilité pour utiliser cette espèce au niveau spécification, pour tout besoin d'implémentation en utilisant son descendant l'espèce *Liste.*

- L'espèce *Liste* est une implémentation de l'espèce *Ensembles_finis*. Elle définit toutes les méthodes déclarées dans *Ensembles_finis*, comme le type de représentation qui est le type standard *list* défini dans la bibliothèque *Basics* (section 1.1).
- L'espèce *Liste* prend comme paramètre l'espèce *Setoid*. Ce paramètre peut être implémenté par n'importe quelle autre espèce de notre spécification qui hérite de *Setoid*.

Remarque :

En plus des bibliothèques présentées, nous aurons besoin de l'instruction *alias* :

```
type nom_t = alias type_FoCaLiZe ;;
```

Cette instruction définit une abréviation pour remplacer toutes les occurrences d'un type *type_FoCaLiZe* par un nom *nom_t* choisi par le programmeur, et ce dans un souci de lisibilité et de facilité d'écriture. Elle doit être introduite au niveau top-level de la spécification.

4.3. Transcription des composants :

Trois espèces sont définies pour représenter les trois composants du réseau :

L'espèce *Place* :

Cette espèce modélise le composant place :

```
type marquage = alias int ;;

species Place = inherit Setoid ;
representation = string* marquage ;
. . .
let new_place (x : string, y : marquage) : Self = (x, y) ;
let get_nom (x : Self) : string = fst(x) ;
let get_marquage (x : Self) : marquage = snd(x) ;

let equal (x : Self, y : Self) : bool = if ((get_nom(x) = get_nom(y))
&& (get_marquage(x)= get_marquage(y))) then True else False ;

. . .
end ;;
```

- L'espèce *Place* hérite de l'espèce *Setoid*. Ce qui nous permettra de spécifier l'espèce *Ensemble_place* ultérieurement.
- Nous définissons un type *marquage* à partir du type des entiers *int* de FoCaLiZe grâce à l'instruction *alias*.
- Le nom des places est représenté par le type chaîne de caractères (*string*).
- Le type support de l'espèce *Place* est le produit cartésien des types *string* et *marquage*. Les entités manipulées dans cette espèce sont donc des couples composés du nom de la place et de son marquage.
- Pour accéder aux éléments des couples, nous définissons deux fonctions : *get_nom* pour récupérer le nom d'une place, *get_marquage* pour avoir son marquage, elles sont définies à partir des fonctions de base *fst* et *snd* (section 1.1.2).
- Une fonction pour créer de nouvelles entités dans l'espèce *new_place*, associe au nom d'une place et à un marquage une entité du type support.
- La fonction d'égalité des couples *equal* est définie à partir de la fonction de base (=) spécifiée dans la bibliothèque *Basics*, deux couples du type support sont égaux si leurs éléments sont égaux terme à terme.

L'espèce *Transition* :

Le même principe de la transcription des places sera appliqué aux transitions :

```
species Transition = inherit Setoid ;
representation = string ;

. . .
let nom_transition (x : Self) : string = x ;
let new_transition (x : string) : Self = x ;

let equal (x : Self, y : Self) : bool = x = y ;

. . .
end ;;
```

- L'espèce *Transition* hérite de l'espèce *Setoid*. Ce qui nous permettra de spécifier l'espèce *Ensemble_transition* ultérieurement.
- Dans notre démarche, les transitions ne possèdent qu'un seul attribut, c'est leur nom. Le type support de l'espèce *Transition* est donc simplement le type *string*.
- L'espèce contient une fonction de création de nouvelles entités *new_transition*.
- La fonction d'égalité est la fonction de base (=).

L'espèce *Arc* :

Cette espèce est paramétrée par les espèces *Place* et *Transition* afin de construire le lien entre les composants.

```
type poids = alias int ;;

species Arc (P is Place, T is Transition) = inherit Setoid ;
representation = (P* T)* poids ;

. . .
let new_arc (x : P, y : T, z : poids) : Self = ((x, y), z) ;

let get_poids (x : Self) : poids = snd(x) ;
let get_place (x : Self) : P = fst (fst (x)) ;
let get_transition (x : Self) : T = snd (fst (x)) ;

let equal (x : Self, y : Self) = if (((get_place(x) = get_place(y))
&& (get_transition(x) = get_transition(y))) && (get_poids(x) =
get_poids(y))) then True else False ;

. . .
end ;;
```

- L'espèce *Arc* hérite de l'espèce *Setoid*. Ce qui nous permettra de spécifier l'espèce *Ensemble_ arc* ultérieurement.
- Comme pour le type marquage, nous définissons un type *poids* à partir du type des entiers *int* de FoCaLiZe grâce à l'instruction *alias*.
- Le type support de l'espèce *Arc* est le produit cartésien des types supports des espèces *Place* et *Transition* et du type *poids*. Les entités manipulées par l'espèce *Arc* sont des triplets (place, transition, poids), ce qui définit la relation entre les places et les transitions, et permet de créer les nœuds d'un réseau.
- Le sens des arcs est défini par le signe de son poids, l'arc est dirigé d'une place vers une transition si le poids est négatif, ou le sens inverse s'il est positif.
- De même que pour les espèces précédentes, l'espèce *Arc* contient des fonctions d'égalité (*equal*), d'accès aux entités pour récupérer le poids d'un arc, la place ou la transition à ses extrémités (*get_place*, *get_transition* et *get_poids*), ainsi que de création de nouvelles entités (*new_arc*).

4.4. Transcription des ensembles de composants :

En plus des espèces qui représentent les composants, nous spécifions trois autres espèces pour représenter les ensembles de composants. Celles-ci nous permettent de les regrouper et de les manipuler sous la forme d'ensembles finis, et de pouvoir constituer la structure d'un réseau de Petri en général.

L'espèce *Ensemble_place* :

```
species Ensemble_place (P is Place) = inherit Liste (P) ;

let create (l : list (P)) : Self = l ;
let support (l : Self) : list (P) = l ;

(* fonction d'affichage d'un ensemble de places *)
let print (l : Self) =
let rec aux (l) =
match l with
            | [] -> "]"
            | x :: l -> P!print (x) ^ "," ^ aux (l) in
      "[" ^ (aux (l));
let afficher_liste (l : Self) = print_string (print(l)) ;
. . .
end ;;
```

- L'espèce *Ensemble_place* hérite de l'espèce *Liste*. Ici l'espèce *Liste* est utilisée à la place de l'espèce *Ensembles_finis* en vue d'implémenter directement l'espèce *Ensemble_place*.
- Elle prend comme paramètre l'espèce *Place* qui implémente l'espèce *Setoid*, pour définir les éléments que contiennent les ensembles finis manipulés par l'espèce *Ensemble_place*.
- Le type support de l'espèce *Ensemble_place* est le type standard *list* (*P*). Il est hérité de l'espèce *Liste*.
- La création d'un ensemble se fait par ajouts successifs des éléments créés par l'espèce *Place* en utilisant la fonction héritée des ensembles finis *ajoute_element*.
- Nous définissons une fonction pour afficher les éléments d'une liste de places.

De la même manière, nous définissons les espèces *Ensemble_transition* et *Ensemble_arc*.

L'espèce *Ensemble_transition* :

```
species Ensemble_transition(T is Transition) = inherit Liste (T);

let create (l : list (T)) : Self = l ;
let support (l : Self) : list (T) = l ;

let print (l : Self) =
let rec aux (l) =
match l with
            | basics#[] -> "]"
            | x :: l -> T!nom_transition (x) ^ "," ^ aux (l) in
      "[" ^ (aux (l));
let afficher_liste (l :Self) = print_string (print(l)) ;

. . .
end ;;
```

L'espèce *Ensemble_arc* :

```
species Ensemble_arc (P is Place, T is Transition, A is Arc(P, T)) =
inherit Liste (A) ;

let create (l : list (A)) : Self = l ;
let support (l : Self) : list (A) = l ;

let print (l : Self) =
let rec aux (l) =
match l with
            | basics#[] -> "]"
            | x :: l -> A!print (x) ^ "," ^ aux (l) in
      "[" ^ (aux (l));
let afficher_liste (l :Self) = print_string (print(l)) ;

. . .
end ;;
```

4.5. Transcription du réseau :

La dernière espèce de notre spécification est celle qui regroupe toutes les précédentes espèces et modélise la structure d'un réseau de Petri.

```
species Reseau (P is Place, T is Transition, A is Arc(P, T), EP is
Ensemble_place(P), ET is Ensemble_transition(T), EA is Ensemble_arc
(P, T, A)) = inherit Setoid ;
representation = (EP* ET)* EA ;

let new_reseau (x : EP, y : ET, z : EA) : Self = ((x, y), z) ;

let get_ensemble_place (x : Self) : EP = fst (fst (x)) ;
let get_ensemble_transition (x : Self) : ET = snd (fst (x)) ;
let get_ensemble_arc (x : Self) : EA = snd (x) ;

let equal (x : Self, y : Self) : bool = if
((EP!equal(get_ensemble_place(x), get_ensemble_place(y)) &&
ET!equal(get_ensemble_transition(x), get_ensemble_transition(y))) &&
EA!equal(get_ensemble_arc(x), get_ensemble_arc(y))) then True
else False ;

. . .
end ;
```

- L'espèce *Reseau* est paramétrée par toutes les autres espèces de la spécification.
- Le type support de l'espèce *Reseau* est le produit cartésien des types supports des trois espèces ensembles, ce qui traduit la structure réelle d'un réseau de Petri composée d'un ensemble de places et leurs marquages, d'un ensemble d'arcs et leurs poids et d'un ensemble de transitions.

- Des fonctions sont définies pour accéder aux ensembles de composants du réseau, *get_ensemble_place*, *get_ensemble_transition* et *get_ensemble_arc* pour récupérer, respectivement, l'ensemble des places, l'ensemble des transitions et l'ensemble des arcs d'un réseau de Petri.
- La fonction *new_reseau* permet de créer de nouveaux réseaux de Petri à partir d'ensembles de composants définis par ailleurs.

Grâce à la hiérarchie d'espèces définie dans notre spécification, nous avons capturé la structure générale des réseaux de Petri. Il nous est maintenant possible de passer au niveau implémentation de notre méthode, afin de pouvoir l'appliquer à des exemples concrets, et ceci grâce aux collections.

Les collections :

Chacune des espèces de la spécification sera implémentée par une collection. Le niveau utilisateur sera constitué de sept collections au total :

```
collection Place_collection = implement Place ; end ;;
collection Transition_collection = implement Transition ; end ;;
collection Arc_collection = implement Arc (Place_collection,
Transition_collection); end ;;

collection Ensemble_place_collection = implement Ensemble_place
(Place_collection); end ;;
collection Ensemble_transition_collection = implement Ensemble_transition
(Transition_collection); end;;
collection Ensemble_arc_collection = implement Ensemble_arc
(Place_collection, Transition_collection, Arc_collection); end ;;

collection Reseau_collection = implement Reseau (Place_collection,
Transition_collection, Arc_collection, Ensemble_place_collection,
Ensemble_transition_collection,  Ensemble_arc_collection) ; end ;;
```

La hiérarchie des espèces est représentée dans la figure 4.2.

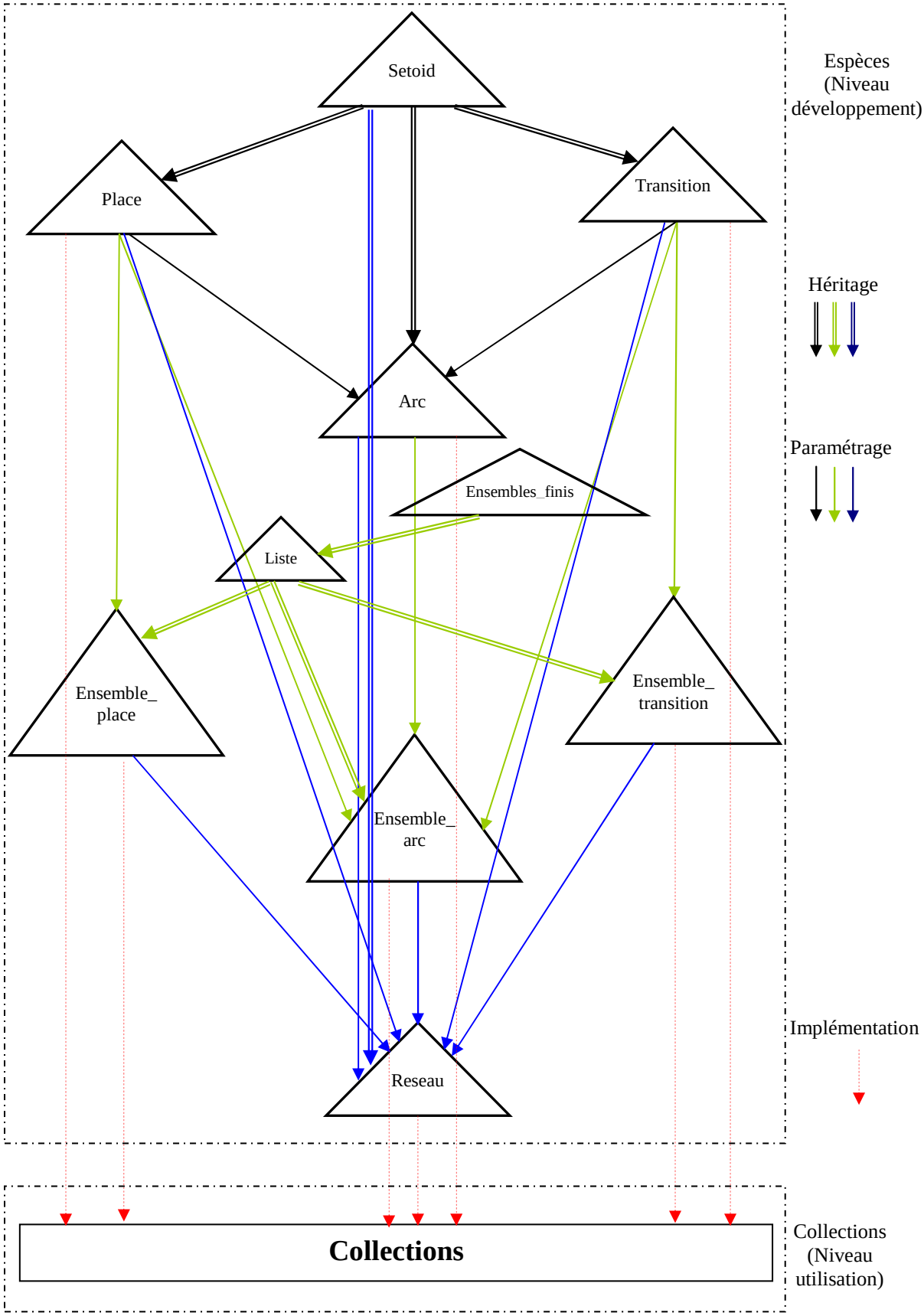

Fig 4.2 Hiérarchie des espèces représentant la modélisation d'un Réseau de Petri.

4.6. Transcription de la propriété de raffinement :

Dans cette section, nous nous intéressons à la transcription d'une propriété des réseaux de Petri ordinaires qui est la propriété de raffinement (section 2.5). Nous nous sommes inspirés dans notre choix du travail de [MAY08] dans lequel cette propriété a été spécifiée et vérifiée en Coq.

Dans ce qui suit, nous allons reprendre les clauses de cette propriété une par une, et présenter leur transcription en FoCaLiZe.

Soient $R1(P1, T1, A1, Pre1, Post1, M_0 1)$ et $R2(P2, T2, A2, Pre2, Post2, M_0 2)$ deux réseaux de Petri ordinaires définis par leurs ensembles de places, de transitions et d'arcs, leurs fonctions d'incidence et leurs marquages initiaux respectifs.

1. La première clause de la propriété de raffinement est que l'ensemble des places du réseau d'origine sera inclus dans l'ensemble de places de son raffinement :

   $P1 \subseteq P2$.

```
species Ensemble_place (P is Place) = inherit Liste (P) ;
. . .
property inclusion : all ep1 ep2 : Self, sous_ensemble (ep1, ep2)
<-> all pl : P, est_element (pl, ep1) -> est_element (pl, ep2);
proof of inclusion = by property sous_ensemble_spec ;
. . .
end
```

- Les fonctions *sous_ensemble* et *est_element* sont héritées de l'espèce *Liste*. Elles vérifient l'inclusion de deux ensembles de places et l'appartenance d'une place à un ensemble.
- Nous spécifions une propriété d'inclusion sur deux ensembles de places : un ensemble de places ep_1 est inclus dans ep_2 si et seulement si quelle que soit l'entité *p* de l'espèce *Place*, si *p* appartient à ep_1 alors *p* appartient à ep_2.
- Cette propriété couvre également la clause de conservation du marquage initial des places du réseau original dans le réseau raffiné :

  Le marquage initial des places de P1 du réseau R1 reste inchangé dans le réseau R2.

  En effet, la propriété d'inclusion concerne les éléments du type support de l'espèce *Place* qui sont constitués du nom des places et de leurs marquages.

- La propriété *inclusion* est prouvée par la propriété *sous_ensemble_spec* importée de la bibliothèque des ensembles finis. Cette propriété est spécifiée dans l'espèce *Liste* pour l'inclusion de deux ensembles finis.

2. Le même principe sera adopté pour les transitions ($T1 \subseteq T2$) définies dans l'espèce *Ensemble_transition*, et les arcs ($A1 \subseteq A2$) dans l'espèce *Ensemble_arc* :

```
species Ensemble_transition(T is Transition) = inherit Liste (T);
. . .
property inclusion : all et1 et2 : Self, sous_ensemble (et1, et2) <->
(all tr : T, est_element (tr, et1) -> est_element (tr, et2));
proof of inclusion = by property sous_ensemble_spec ;
. . .
end ;;

species Ensemble_arc (P is Place, T is Transition, A is Arc(P, T)) =
inherit Ensembles_finis (A), Liste (A) ;
. . .
property inclusion : all ea1 ea2 : Self, sous_ensemble (ea1, ea2) <->
(all ar : A, est_element (ar, ea1) -> est_element (ar, ea2));
proof of inclusion = by property sous_ensemble_spec ;
. . .
end ;;
```

3. La dernière clause de la propriété est celle qui exprime la condition imposée sur les arcs qui n'appartiennent pas au réseau d'origine *R*1, et appartiennent à son raffinement *R*2, comme suit :
   Aucun arc ne relie une place de P1 à une transition de T2\T1 (transitions appartenant à T2 et n'appartenant pas à T1).
   Aucun arc ne relie une transition de T1 à une place de P2\P1.
   Aucun arc ne relie une transition de T2\T1 à une place de P1.

   Tout d'abord nous définissons une fonction *arc_direction* dans l'espèce *Reseau*, qui prend comme arguments deux ensembles de l'espèce *Ensemble_place* (*ep*1 et *ep*2), deux de l'espèce *Ensemble_transition* (*et*1 et *et*2) et deux de l'espèce *Ensemble_arc* (*ea*1 et *ea*2), et retourne un booléen.

```
species Reseau (P is Place, T is Transition, A is Arc(P, T), EP is
Ensemble_place(P), ET is Ensemble_transition(T), EA is Ensemble_arc
(P, T, A)) = inherit Setoid ;
. . .
let arc_direction (ep1 : EP, ep2 : EP, et1 : ET, et2 : ET, ea1 : EA,
ea2 : EA) : bool =
      let rec aux (l) =
      match l with
      | [] -> True
      | t::q -> if ((EP!est_element(A!get_place(t), ep1) &&
ET!est_element(A!get_transition(t), ET!diff(et2, et1))) ||
((EP!est_element (A!get_place(t), EP!diff(ep2, ep1)) &&
ET!est_element (A!get_transition(t), et1)) && (A!get_poids(t) >0x
0))) then False else aux (q)
in aux (EA!support(EA!diff (ea2, ea1)));
. . .
end ;;
```

- La fonction *arc_direction* calcule *diff*(*ea*2, *ea*1) l'ensemble des éléments appartenant à *ea*2 et n'appartenant pas à *ea*1, et ce grâce à la fonction *diff* héritée de l'espèce *Liste*. Elle renvoie la valeur *False* dans deux cas : si un élément de *diff*(*ea*2, *ea*1) relie un élément de *ep*1 à un élément de *diff*(*et2*, *et1*) ou l'inverse, ou s'il relie un élément de *et*1 à un élément de *diff*(*ep*2, *ep*1). Elle renvoie *True* sinon.

  Ensuite, nous définissons une fonction *arc_direction_reseau* qui spécifie les conditions sur les arcs appartenant à l'ensemble différence entre les ensembles d'arcs de deux réseaux de Petri, et ce en appliquant la fonction *arc_direction* sur les ensembles de composants des deux réseaux.

```
species Reseau . . .
. . .
let arc_direction_reseau (r1 : Self, r2 : Self) = if
(arc_direction(get_ensemble_place(r1), get_ensemble_place(r2),
get_ensemble_transition(r1), get_ensemble_transition(r2),
get_ensemble_arc(r1), get_ensemble_arc(r2))) then true else false ;
. . .
end ;;
```

A toute définition d'une fonction booléenne en FoCaLiZe, il possible d'associer un théorème, qui capture l'essence de la définition sous une forme logique.

```
species Reseau . . .
. . .
theorem condition_arc_direction_reseau : all r1 r2 : Self,
arc_direction_reseau (r1, r2) <->
(arc_direction(get_ensemble_place(r1), get_ensemble_place(r2),
get_ensemble_transition(r1), get_ensemble_transition(r2),
get_ensemble_arc(r1), get_ensemble_arc(r2)))
proof = by definition of arc_direction_reseau ;
. . .
end ;;
```

- Le théorème *condition_arc_direction_reseau* spécifie la dernière clause de la propriété de raffinement.

4.7. Application sur un exemple :

Afin d'appliquer notre démarche, nous reprenons l'exemple de la figure 2.5 (section 2.5). Nous commençons par la création des éléments :

```
(* Création places du réseau R1 *)
let place1_r1 = Place_collection!new_marquage ("P1", 0) ;;
. . .
let place4_r1 = Place_collection!new_marquage ("P4", 2) ;;

(* Création places du réseau R2 *)
let place1_r2 = Place_collection!new_marquage ("P1", 0) ;;
. . .
let place5_r2 = Place_collection!new_marquage ("P5", 1) ;;
let place6_r2 = Place_collection!new_marquage ("P6", 0) ;;

(* Création transitions du réseau R1 *)
let transition1_r1 = Transition_collection!new_transition ("T1") ;;
let transition2_r1 = Transition_collection!new_transition ("T2") ;;
(* Création transitions du réseau R2 *)
let transition1_r2 = Transition_collection!new_transition ("T1") ;;
let transition2_r2 = Transition_collection!new_transition ("T2") ;;
let transition3_r2 = Transition_collection!new_transition ("T3") ;;

(* Création arcs du réseau R1 *)
let arc1_r1 = Arc_collection!new_arc (#place1_r1, #transition1_r1, (-1)) ;;
. . .
let arc6_r1 = Arc_collection!new_arc (#place1_r1, #transition2_r1, 1) ;;

(* Création arcs du réseau R2 *)
let arc1_r2 = Arc_collection!new_arc (#place1_r2, #transition1_r2, (-1)) ;;
. . .
let arc7_r2 = Arc_collection!new_arc (#place5_r2, #transition2_r2, (-1)) ;;
let arc8_r2 = Arc_collection!new_arc (#place5_r2, #transition3_r2, (-1)) ;;
let arc9_r2 = Arc_collection!new_arc (#place6_r2, #transition3_r2, 1) ;;
```

```
(* Création ensembles de places réseau R1 *)
let ensemble_places_r1 =
Ensemble_place_collection!ajoute_element (#place1_r1,
. . .
Ensemble_place_collection!ajoute_element (#place4_r1,
#ensemble_places_vide_r1)))) ;;
(* Création ensembles de places réseau R2 *)
let ensemble_places_r2 =
Ensemble_place_collection!ajoute_element (#place1_r2,
. . .
Ensemble_place_collection!ajoute_element (#place6_r2,
#ensemble_places_vide_r2))))) ;;

(* Création ensembles de transitions réseau R1 *)
let ensemble_transitions_r1 =
Ensemble_transition_collection!ajoute_element (#transition1_r1,
. . .,
ensemble_transitions_vide_r1)) ;;
(* Création ensembles de transitions réseau R2 *)
let ensemble_transitions_r2 =
Ensemble_transition_collection!ajoute_element (#transition1_r2,
. . .,
#ensemble_transitions_vide_r2))) ;;

(* Création ensembles d'arcs réseau R1 *)
let ensemble_arcs_r1 =
Ensemble_arc_collection!ajoute_element (#arc1_r1,
. . .,
#ensemble_arcs_vide_r1)))))) ;;
(* Création ensembles d'arcs réseau R2 *)
let ensemble_arcs_r2 =
Ensemble_arc_collection!ajoute_element (#arc1_r2,
. . .,
#ensemble_arcs_vide_r2)))))) ;;


(* Création des réseaux r1 et r2 *)
let reseau1 = Reseau_collection!new_reseau (#ensemble_places_r1,
#ensemble_transitions_r1, #ensemble_arcs_r1) ;;
let reseau2 = Reseau_collection!new_reseau (#ensemble_places_r2,
#ensemble_transitions_r2, #ensemble_arcs_r2) ;;
```

Nous pouvons appliquer les fonctions d'inclusions des ensembles de places, de transitions, et d'arcs ainsi que la fonction de direction des arcs sur notre exemple, pour s'assurer que toutes les clauses de la propriété de raffinement sont vérifiées sur notre exemple.

1. Dans le cas où une des clauses est insatisfaite, comme la non inclusion des ensembles de composants sur l'exemple de la figure 4.3, et la non inclusion des ensembles de places, la fonction d'inclusion appliquée sur les ensembles de places renvoie la valeur *false* et la propriété n'est pas vérifiée.

```
. . .
(* Création places du réseau R1 *)
let place1_r1 = Place_collection!new_marquage ("P1", 0) ;;
let place2_r1 = Place_collection!new_marquage ("P2", 1) ;;
let place3_r1 = Place_collection!new_marquage ("P3", 1) ;;
let place4_r1 = Place_collection!new_marquage ("P4", 2) ;;

(* Création places du réseau R2 *)
let place1_r2 = Place_collection!new_marquage ("P1", 0) ;;
let place3_r2 = Place_collection!new_marquage ("P3", 1) ;;
let place4_r2 = Place_collection!new_marquage ("P4", 2) ;;
let place5_r2 = Place_collection!new_marquage ("P5", 1) ;;
let place6_r2 = Place_collection!new_marquage ("P6", 0) ;;
...
```

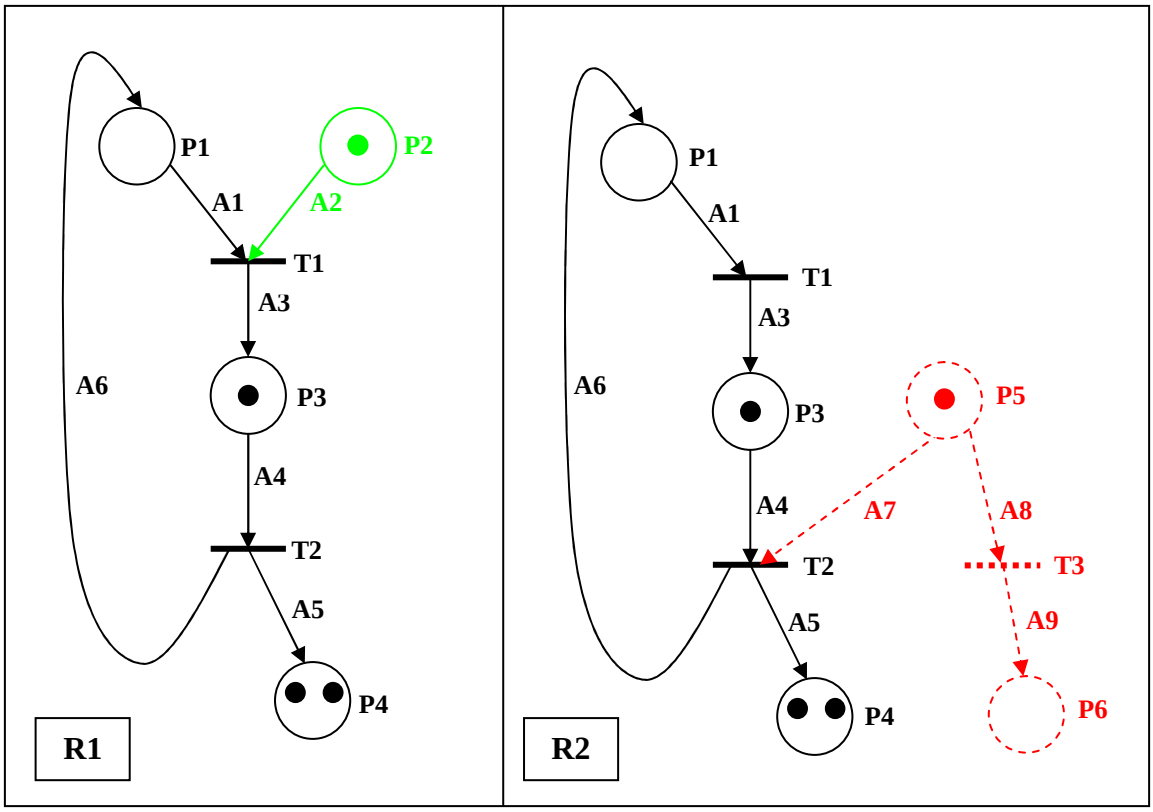

Fig 4.3 Propriété d'inclusion des places non vérifiée

2. Une direction erronée des arcs supplémentaires dans le réseau raffiné mène également à une propriété de raffinement insatisfaite.

   Prenons l'exemple de la figure 4.4. L'arc A7 est dirigé en sortie de la transition T2 vers la place P5, son poids est par conséquent positif, et ceci ne correspond pas à la dernière clause du lemme de raffinement, du coup la fonction *arc_direction_reseau* renvoie la valeur *false*.

```
(* Création arcs du réseau 1 *)
let arc1_r1 = Arc_collection!new_arc (#place1_r1, #transition1_r1, (-1)) ;;
. . .
let arc6_r1 = Arc_collection!new_arc (#place1_r1, #transition2_r1, 1) ;;

(* Création arcs du réseau 2 *)
let arc1_r2 = Arc_collection!new_arc (#place1_r2, #transition1_r2, (-1)) ;;
. . .
let arc7_r2 = Arc_collection!new_arc (#place5_r2, #transition2_r2, 1) ;;
let arc8_r2 = Arc_collection!new_arc (#place5_r2, #transition3_r2, (-1)) ;;
let arc9_r2 = Arc_collection!new_arc (#place6_r2, #transition3_r2, 1) ;;
```

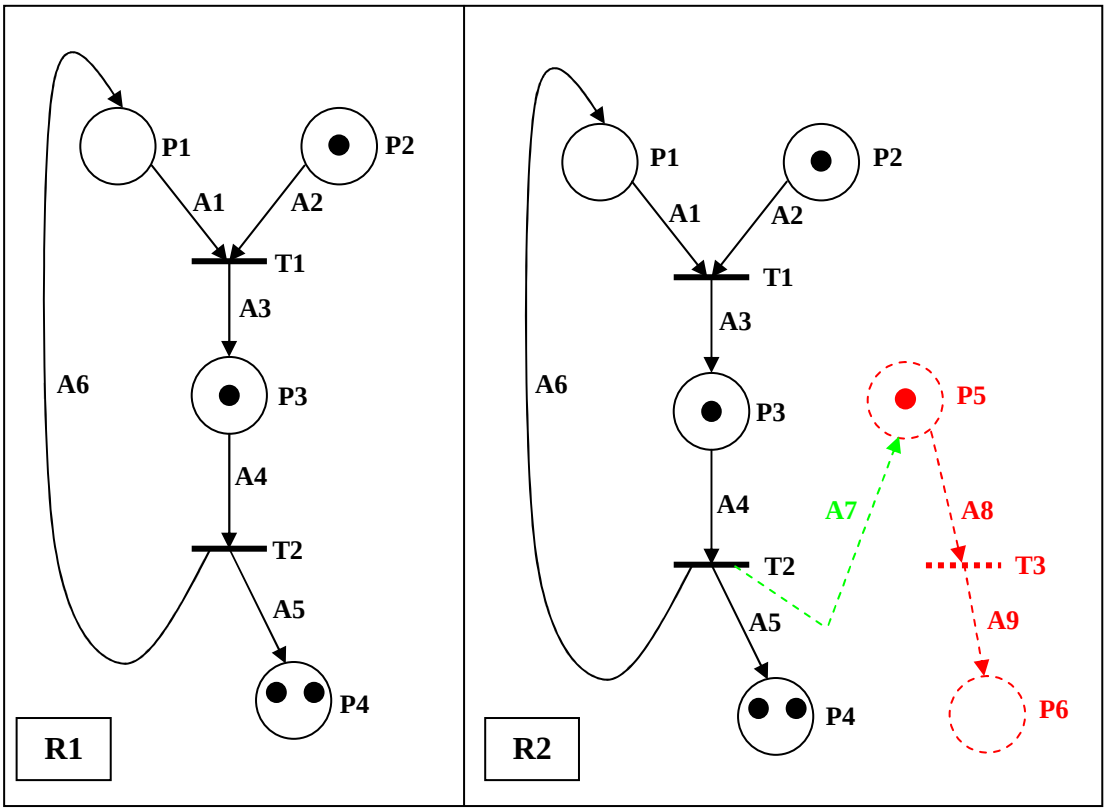

Fig 4.4 Propriété de direction des arcs supplémentaires non vérifiée.

Conclusion :

La démarche de transcription proposée dans ce chapitre nous a permis d'obtenir une spécification FoCaLiZe à partir du modèle en réseau de Petri ordinaire, et de vérifier la propriété de raffinement des réseaux de Petri ordinaires sur un cas concret.

Nous nous sommes limités dans notre travail à l'aspect statique des réseaux de Petri. Cela est dû à la difficulté de modéliser le comportement d'un réseau de Petri de manière totalement générique, sans une connaissance préalable de sa structure. De plus, l'évolution d'un réseau de Petri s'apparente à une succession d'évènements, et la spécifier en termes de fonctions contrôlées par un utilisateur ne traduit pas fidèlement le processus d'évolution.

Toutefois, FoCaLiZe dispose de constructions et d'outils efficaces, qui permettent de spécifier et de prouver des propriétés mathématiques à partir d'un modèle formel comme les réseaux de Petri. Grâce à la richesse de sa librairie, nous nous sommes appuyés sur des structures prédéfinies pour modéliser les composants des réseaux de Petri, individuellement sous la forme d'espèces, ou regroupés dans des ensembles finis. Ceci facilite la spécification de propriétés, selon qu'elles traitent de composants spécifiques, ou d'ensembles de composants, ce second cas étant illustré par la propriété de raffinement prise en charge par notre démarche.

Conclusion

Nous avons présenté une démarche de transcription des réseaux de Petri ordinaires vers une spécification FoCaLiZe. Pour cela nous avons dérivé la structure générale du réseau (places, arcs et transitions) en une hiérarchie d'espèces FoCaLiZe, correspondant aux liens entre les composants et dont les types supports représentent les propriétés des composants (identifiants, marquage et poids). Les opérations définies dans les espèces nous ont permis ensuite de créer et de manipuler des exemples concrets de réseaux de Petri.

Pour illustrer l'aspect preuve, nous avons prouvé la propriété de raffinement des réseaux de Petri ordinaires en utilisant le prouveur automatique Zenon. L'introduction de Zenon dans Focalize a grandement facilité notre tâche, car elle limite la participation du concepteur dans l'élaboration de la preuve, en automatisant le processus. De plus, son usage ne nécessite pas de connaissances approfondies, en dehors de la logique du premier ordre.
La fiabilité des preuves produites par Zenon a été ensuite assurée par Coq.
Il aurait été possible de réaliser des preuves de propriétés dans une spécification FoCaLiZe directement en Coq, mais ceci nécessite un réel niveau d'expertise dans l'utilisation de cet outil et surtout beaucoup de créativité pour produire les preuves.

Dans le travail de [MAY08], dont nous nous sommes inspirés dans notre démarche, la propriété de raffinement est prouvée par Coq, sur un exemple particulier. Notre démarche, pour sa part, se veut générique, nous pouvons appliquer notre spécification FoCaLiZe pour vérifier le raffinement sur tout exemple de réseau de Petri ordinaire.

Pour conclure nous présentons deux perspectives qui sont la suite naturelle de nos travaux.

- Nous avons réalisé notre transcription en suivant des règles précises pour chaque étape. Aussi, le processus pourrait être automatisé afin d'obtenir, systématiquement, une spécification FoCaLiZe. En le couplant ensuite, avec un éditeur de réseau de Petri (PIPE, CPN …), nous pourrions, à partir d'un exemple concret, obtenir une hiérarchie d'espèces de façon automatique et du coup, réduire significativement l'intervention du concepteur dans la transcription.

La disponibilité de cet outil permettrait d'associer dans une méthode globale de développement : les réseaux de Petri pour utiliser un modèle de départ sous forme graphique, et FoCaLiZe pour l'aspect preuve.

- Pour terminer, on peut remarquer que nous nous sommes limités dans notre travail à la transcription des réseaux de Petri ordinaires. Néanmoins, son extension à des réseaux de haut niveau semble tout à fait réalisable. Comme l'aspect statique des réseaux de Petri est constitué d'éléments identiques, qu'ils soient ordinaires ou de haut niveau, notre spécification peut servir de base pour l'intégration d'autres types de réseaux, sans subir de modifications profondes. En ce qui concerne le marquage, FoCaLiZe dispose d'un large choix de types, simples ou complexes, pour prendre en charge différentes natures de jetons. Nous pensons notamment à la notion de couleur dans les réseaux de Petri colorés, qui ont été traitées en Coq dans [CHO10], travail qui peut servir de référence pour une future transcription des réseaux colorés en FoCaLiZe.

Bibliographie

[ABB07] M. Abbas. Transcription des spécifications UML vers le système FoCAL. Thèse de Magister, USTHB, Alger, Juillet 2007.

[ABR96] J.R. Abrial. The B book. Cambridge University Press, aout 1996.

[ART10] A. Artale. Formal Methods. Faculty of Computer Science-Free University of Bolzano, 2010.

[ATT09] C. Attiogbé. Semantic Embedding of Petri Nets into Event-B. IM FMT, 2009.

[BEH99] P. Behm, P. Benoit, A. Faivre et J.M. Meynadier. METEOR : A Successful Application of B in a Large Project. In Wing et al, pp369-387, 1999.

[BER03] B. Berthomieu, P.O. Ribet, F. Vernadat, L'outil TINA, Construction d'espaces d'états abstraits pour les réseaux de Petri et réseaux Temporels. Modélisation des Systèmes Réactifs, MSR'2003, Hermes, 2003.

[BER11] Y. Bertot et P. Castéran. Le Coq' Art (V8). Texts in Theoritical Computer Science, 26 janvier 2011.

[BIE96] P. Bieber. Interprétation d'un modèle de sécurité. Techniques et Sciences Informatiques, vol.15, n6, Avril1996.

[BIL99] J. Billington, M. Diaz, and G. Rozenberg. Application of Petri nets to communication networks. In Springer Verlag Lecture Notes in Computer Science, editor, *Advances in Petri nets*, volume 1605, 1999.

[BON00] P. Bon. Du cahier de charge aux spécifications formelles : une méthode basée sur les réseaux de Petri de haut niveau. PhD thesis, Université des sciences et technologies de Lille, Juin 2000.

[BON07] R. Bonichon, D. Delahaye et D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs, LPAR'07 Logic for Programming Artificial Intelligence and Reasoning, Yerevan (Armenia), pp.151-165, *Series LNAI*, 2007.

[BOU00] S. Boulmé. Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel. PhD thesis, Université de Paris VI, 2000.

[BRA83] G.W. Brams. Réseaux de Petri : Théorie et pratique. Masson, 1983.

[BUR90] J.R. Burch, E. Clarke, D.L. Dill, L.J. Hwang and K. McMillan. Symbolic Model Checking, 10^{20} states and beyond. In Proceedings of the 5^{th} Annual IEEE Symposium On Logic in Computer Science, pages 428-439, 1990.

[BUT96] M. Butler et M. Walder. Distributed System Development in B. In Habrias, pp 155-168, 1996.

[CHO10] Christine Choppy, Micaela Mayero, and Laure Petrucci. Coloured Petri net refinement specification and correctness proof with Coq. Volume 6, pp195-202, 2010.

[CLA99.a] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, Cambridge, Massachusetts, 1999.

[CLA99.b] E. Clarke, O. Grumberg, M. Minea and D. Peled. State space reduction using partial order techniques. International Journal of Software Tools for Technology Transfer, pp 279-287, 1999.

[CLE03] CLEARSY. Manuel de référence du langage B, version 1.8.5, 2003.

[CLE08] CLEARSY. Prouveur interactif, manuel utilisateur, version 3.7, 2008.

[COQ10.a] The Coq Development Team. The Coq Proof Assistant Reference Manual v8.3. LogiCal Project, 20 octobre 2010.

[COQ10.b] G. Huet, G. Kahn et C. Paulin-Mohring. The Coq Proof Assistant, a Tutorial v8.3. LogiCal Project, 14 octobre 2010.

[DEL06] D. Delahaye, J.F. Etienne and V. Donzeau-Gouge. Certifying airport security regulations using the Focal environment. In J. Misra, T. Nipkow, and E.Sekerinski, editors, FM 2006 : Formal Methods, 14th International Symposium on Formal Methods, Proceedings, volume 4085 of Lecture Notes in Computer Science, pages 48–63, Springer, 2006.

[DEL10] D.Delahaye, C.Dubois et P. Tollitte. Génération de code fonctionnel certifié à partir de spécifications inductives dans l'environnement FoCaLiZe, Journées Francophones des Langages Applicatifs, INIRIA, Janvier 2010.

[DIA01] M. Diaz. Les Réseaux de Petri : Modèles Fondamentaux. Hermès Science Publications, Paris, 2001.

[DOL04] D. Doligez. Zenon, first-order prover with coq-checkable output. 2nd Workshop on Coq and Rewriting, 2004.

[ETI06] Jean-Frédéric Etienne. Modélisation de la réglementation de l'aviation civile en FoCAL. Séminaire de données et apprentissage artificiel, 2006.

[EVA05] S. Evagelista. High Level Petri Nets Analysis with Helena. International Conference On Application and Theory of Petri Nets, Miami, USA, Vol. 3536(26), pp.455-464, *Series LNCS*, January 2005.

[FCL09.a] T. Hardin, F. Pessaux, P. Weis, D. Doligez. FoCaLiZe 0.6.0 Reference Manual, CNAM, INRIA, et LIP6, December 2009.

[FCL09.b] FoCaLiZe Team. A Short Tutorial for FoCaLiZe : Implementing Sets, CNAM, INRIA, et LIP6, July 2009.

[FEC05] S. Fechter. Sémantique des traits orientés objet de Focal. PhD Thesis, Université de Paris 6, 18 Juillet 2005.

[FOC06.a] L'équipe de développement FoCAL. Tutoriel FoCAl, Version 0.3, CNAM, INRIA, et LIP6, Septembre 2006.

[FOC06.b] FoC developpement team. FoCAL Reference Manual, CNAM, INRIA, et LIP6, 2006.

[GEN87] H.J. Genrich. Predicate/Transition Nets. In W. Bauer, W. Reisig and G. Rozenberg, editors, Petri Nets : Central models and their properties, pp 208-247, Berlin, FRG, 1987. Appeared in lecture notes on computer science 254.

[HAC72] M. Hack. Analysis of production schemata by Petri nets. Technical report, MAC, MIT, February 1972.

[HAS02] B. Hasselblatt et A. Katok, Handbook of Dynamical Systems, Elsevier. Vol. 1 Part A, 2002.

[HOA95] J.P. Hoare. Application of the B Method to CICS. Chap. 6, Prentice Hall International, 1995.

[HOL70] A.W. Holt and F. Commoner. Events and conditions, record of the project MAC. In *Proc. of the Conference on ACM*, pages 3-52, New York, 1970.

[HOL97] G.J. Holzmann. The Model checker SPIN. IEEE Transactions on Software Engineering, pp 279-295, Mai 1997.

[HOW80] William A. Howard. Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479-490. Academic Press, 1980.

[JAU05] M. Jaume and C. Morisset. Formalisation and implementation of access control models. In Information Assurance and Security (IAS'05) International Conference on Information Technology, ITCC, pages 703–708. IEEE CS Press, 2005.

[JEN92] K. Jensen: Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use, Vol1, Berlin, Springer-Verlag, 1992.

[JEN08] K. Jensen and L. Kristensen. Coloured Petri Net, Modelling and Validation of Concurrent Systems. Monograph to be published by Springer Verlag, 2008.

[LAK01] C. Lakos et G. Lewis. Incremental state space construction of coloured Petri nets. 22nd Int. Conf. Application and Theory of Petri Nets (ICATPN'01), Newcastle, UK, volume 2075 of Lecture Notes in Computer Science, pp 263–282, Juin 2001.

[LAN98] J.L. Lannet. Using The B Method to model protocols. AFADL 98, 1998.

[LAR97] K.G. Larsen, P. Pettersson and W. Yi. UPPAAL in Nutshell. Journal of software tools for technology transfer, vol. 1, n° ½, pp 134-152, 1997.

[LER05] X. Leroy. The Objective Caml system release 3.09, Documentation and user's manual, INRIA, octobre 2005.

[LIS93] B. Liskov and J.M. Wing. A new definition of the subtype relation. In ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, pages 118-141, London, UK, Springer-Verlag, 1993.

[MAT99] B. Matthews et E. Locuratolo. Formal development for databases in ASSO and B. Lecture Notes in Computer Science, Formal Methods, 1999.

[MAY08] M. Mayero, C. Choppy, L. Petrucci. Experimenting Formal Proofs of Petri Nets Refinements. Electronic Notes in Theoretical Computer Science, pp 213-254, 2008.

[MCM00] K. McMillan. A methodology for hardware verification using compositional model checking. Science of Computer Programming, pp 279–309, 2000.

[MOL82] M.K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transaction on Computers*, C-31(9) : pp 913-917, Septembre 1982.

[MON03] J.F. Monin. Understanding Formal Methods. Springer, 2003.

[NAT80] S.O. Natkin. Les réseaux de Petri stochastiques et leur application à l'évaluation des systèmes informatiques. Phd thesis, Conservatoire National des Arts et Metiers (CNAM), Paris, Juin 1980.

[PAU02] Christine Paulin. Génie logiciel : Introduction à la méthode B. CNRS, Maitrise 2001-2002.

[PET62] C.A. Petri. *Kommunikation mit Automaten*. Phd dissertation, Institut fÄur Instrumentelle Mathematik, University of Bonn, West Germany, 1962.

[PRE03] V. Prevosto. Conception et implantation du langage Foc pour le développement de logiciels certifiés. PhD thesis, Université de Paris VI, 2003.

[PRE05] V. Prevosto. FoCAL pas à pas : une introduction au langage. Projet FoCAL, INRIA, 07 Décembre 2005.

[RAM74] C. Ramchandani. Analysis of Asynchronous Concurrent Systems by Timed Petri nets. Thèse de doctorat, Massachusetts Institute of Technology, Department of Electrical Engineering, Cambridge, Massachusetts, 1974.

[REI91] W. Reinsig. Petri Nets and algebraic specification, high level Petri Nets, pp 137-170. Springer-Verlag, Avril 1991.

[RIO04] R. Riobo et T. Hardin. Les objets de mathématiques. RSTI – L'objet, Octobre 2004.

[SPI92] J. Spivey. The Z notation, a reference manual. Prentice Hall, 2eme edition, 1999.

[SCO06] G. Scorletti et G. Binet. Réseaux de Petri, Cours Master. Université de Caen, Basse-Normandie, 2006.

[VAL98] A.Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, Springer-Verlag, 1998.

[YOV97] S. Yovine. Kronos : A Verification Tool for Real-time Systems. Journal of software tools for technology transfer, vol. 1, n° ½, pp 123-133, 1997.