Génération de règles de coordination à partir de réseaux de Pétri colorés

Adja Ndeye Sylla[1,2], Maxime Louvel[1], François Pacull[1], Éric Rutten[2]

[1] Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
17 rue des Martyrs 38000 Grenoble, France.
AdjaNdeye.sylla@cea.fr, maxime.louvel@cea.fr, francois.pacull@cea.fr
[2] INRIA, Grenoble, France
eric.rutten@inria.fr

Résumé

Cet article présente un environnement de génération de règles de coordination à partir de réseaux de Pétri colorés. L'environnement proposé est basé sur un langage dédié appelé cpnDSL. À partir d'une spécification cpnDSL décrivant un système donné, des règles de coordination correctes par construction et directement exécutables dans l'environement LINC sont générées. Un modèle vérifiable pour valider le comportement du système et un modèle graphique pour permettre la discussion entre les différents membres du projet sont également générés. Une étude de cas issue du domaine du transport est présentée pour illustrer l'approche proposée.

1 Introduction

Aujourd'hui, les systèmes tels que l'automatisation des bâtiments, le contrôle de processus industriels ou la gestion des réseaux de transport sont constitués de nombreux composants hétérogènes et communicants, pouvant être localisés dans des endroits différents. Les interactions entre ces composants posent de nombreux problèmes tels que la synchronisation, le partage de données et les accès concurrents. Ces interactions peuvent aussi causer des comportements indésirables tels que l'interblocage, la famine et l'interblocage actif.

Dans ce contexte, les langages et modèles de coordination [22] sont essentiels. Ils fournissent un moyen simple et efficace pour gérer les interactions entre les différents composants. À l'instar des premières propositions telles que Linda [6] et Gamma [2] ou les travaux les plus récents comme UbiComo [5] et le paradigme Holo [3], nous avons défini avec LINC [19], un environnement de coordination.

LINC utilise une approche orientée ressource, un langage de règles et un moteur d'exécution des règles. La logique d'une application à développer, est définie comme étant un ensemble de règles qui manipulent des ressources.

Initialement, ces règles étaient écrites manuellement par des personnes familières avec la coordination et les systèmes distribués. Dernièrement, dans le contexte de l'automatisation des bâtiments, nous avons défini un langage dédié (Domain Specific Language ou DSL) qui permet, grâce à une interface graphique, de définir des scénarios impliquant des capteurs et des actionneurs [21]. À partir de ces scénarios, des règles LINC sont automatiquement générées et sont dynamiquement rajoutées au système en cours d'exécution. Les principaux avantages de cette approche sont les suivants : d'abord, elle ouvre la description de scénarios à des personnes plus familières avec le domaine de l'application, même si elles n'ont pas de connaissances en LINC. Ensuite, le DSL et l'interface graphique empêchent (par construction) aux utilisateurs de développer des règles qui ne sont pas correctes par rapport au domaine cible (i.e. l'automatisation des bâtiments). Par exemple, il n'est pas permis de lire des données d'un actionneur

ou d'envoyer une commande à un capteur. Cependant, le langage est limité à un domaine spécifique et l'interface doit être adaptée pour aborder d'autres domaines.

La prochaine étape est d'aller vers un DSL plus générique qui ouvre le champ à plusieurs domaines d'application tout en gardant à l'esprit l'idée de générer, par construction, des règles cohérentes. Bien que cet objectif soit tout à fait facile à gérer avec un langage restreint, il devient évidemment plus difficile lorsque le langage gagne en généralité.

Cet article, présente nos premiers résultats concernant l'utilisation d'un langage graphique et formel (i.e. le réseau de Pétri coloré [15]) pour la modélisation des systèmes considérés et la génération des règles LINC correspondantes à partir des modèles. Les règles générées peuvent être directement exécutées par le moteur de règles LINC sans intervention humaine.

Cette approche présente plusieurs avantages. Premièrement, nous ne sommes pas limités à un domaine d'application spécifique. Deuxièmement, même si le réseau de Pétri coloré n'est pas la première pensée des personnes non techniques pour expliquer leurs problèmes, il est très bien accepté pour la communication et la discussion. Troisièmement, l'approche proposée permet la vérification de différentes propriétés sur les modèles afin de valider le comportement du système. Vu que nous avons, avec le moteur de règles LINC, un environnement d'exécution en mesure de préserver les propriétés du réseau de Pétri, nous pouvons assurer, par construction, l'exactitude des règles générées.

Ce document est structuré comme suit : la Section 2 présente les concepts de base qui ont été utilisés. Ensuite, la Section 3 décrit l'approche proposée. La Section 4 présente une étude de cas, issue du domaine du transport, pour démontrer l'applicabilité de notre approche. La Section 5 discute les travaux connexes. Enfin, la Section 6 conclut le papier et présente les perspectives.

2 Concepts utilisés

Cette section présente le contexte de l'article. Elle décrit brièvement les systèmes à événements discrets qui sont les bases de nos modèles. Ensuite, elle présente LINC, pour rendre le papier autonome. Plus de détails sur LINC peuvent être trouvés dans [19]. Enfin, elle présente les réseaux de Pétri colorés.

2.1 Système à événements discrets

Contrairement aux systèmes dont les comportements sont décrits par des équations différentielles, un système à événements discrets (SED), est un système qui possède un espace d'états discret et dont l'évolution est liée à l'occurrence d'évènements instantanés et asynchrones [7]. Le comportement d'un SED est décrit en énumérant les différents états qu'il peut prendre et les transitions entre ces états. Les transitions sont associées aux évènements.

Prenons l'exemple d'un parking. Les voitures sont à la recherche de places de parking libres pour se garer. Lorsque qu'une place de parking libre est disponible, une voiture se gare et la place devient occupée. Ce système est constitué de deux composants : les *voitures* et les *places de parking*. Une *voiture* possède un identifiant (*ID*) et a deux états : elle est soit *à la recherche d'une place de parking* ou elle est *garée*. Une *place de parking* possède un *ID* et est soit *libre* ou *occupée*.

Les SEDs sont souvent constitués de plusieurs composants qui interagissent pour réaliser une fonction. Ces interactions impliquent la gestion de nombreux problèmes tels que le partage de données, les accès concurrents et la synchronisation des composants qui sont connus comme étant des problèmes difficiles à résoudre lorsque les composants sont déployés sur un système distribué asynchrone. Ces interactions peuvent également causer des comportements indésirables tels que l'interblocage et la famine. Les SEDs peuvent profiter des avantages fournis par les environnements d'exécution tels que LINC et les langages de modélisation formels comme les réseaux de Pétri [10]. Les deux sont présentés aux sections suivantes.

2.2 LINC

LINC est un environnement de coordination utilisé pour développer et déployer des applications sur des systèmes distribués. Une application LINC est un ensemble de règles qui manipulent des ressources. Les ressources sont associées aux concepts du monde réel comme une voiture, son état ou son propriétaire. Une ressource est stockée sous la forme d'un tuple pouvant contenir différents types d'informations (e.g. une voiture, son état, son identifiant, son propriétaire). Dans l'exemple du parking, une ressource qui représente l'état courant d'une voiture garée pourrait être : *("car120","parked")*.

Le langage de coordination LINC est basé sur trois paradigmes :

Mémoire associative [6] : elle consiste à représenter le système comme étant un ensemble de sacs contenant des ressources qui sont sous la forme de tuples. Un sac possède un nom et contient des ressources de même nature. Les ressources sont manipulées à l'aide de trois opérations : *rd()*, *get()* et *put()*. Le *rd()* permet de vérifier la présence d'une ressource donnée dans un sac. Le *get()* permet de consommer une ressource et le *put()* permet d'insérer une ressource. Ces opérations sont utilisées dans des règles de production.

Règles de production [9] : une règle de production est composée d'une partie *précondition* et d'une partie *performance*. La précondition utilise l'opération *rd()* pour vérifier certaines conditions sur le système. Elle est exécutée par un moteur d'inférence. La performance utilise les trois opérations. Le *rd()* est utilisé pour vérifier certaines conditions. Les opérations *get()* et *put()* sont utilisées pour mettre à jour l'état du système. Lorsque toutes les conditions vérifiées en précondition sont satisfaites, le moteur de règles exécute la performance.

Transactions distribuées [4] : elles sont utilisées dans la partie performance pour assurer la propriété du tout-ou-rien. Elles permettent de regrouper dans un même ensemble indivisible : la vérification des conditions (les opérations *rd*) et la mise à jour de l'état du système (les opérations *get* et les opérations *put*). La performance n'aboutit que lorsque toutes ses opérations réussissent. Ainsi, une série d'opérations dans la performance peut avorter si, par exemple, la vérification d'une condition à travers un *rd()* n'est plus vraie ou si une opération *get()* ne peut pas être effectuée parce que la ressource considérée a été consommée par une autre règle.

La combinaison de ces trois paradigmes permet de garantir que le système passe soit d'un état cohérent à un autre ou reste dans son état courant, qui, lui est cohérent.

```
["Cars"].rd(c, "isLooking")&                                    1
["ParkingSpots"].rd(p,"free")
::                                                              3
{ ["Cars"].get(c, "isLooking");
  ["Cars"].put(c, "parked");                                    5
  ["ParkingSpots"].get(p,"free");
  ["ParkingSpots"].put(p,"occupied");                           7
}.
```

Listing 1 – Exemple de règle LINC

Le Listing 1 présente un exemple de règle LINC. Cette rège implémente l'exemple du parking. La *précondition* (située avant le symbole : :) utilise l'opération *rd* pour vérifier la présence des ressources *(c,"isLooking")* et *(p,"free")* respectivement dans les sacs *Cars* et *ParkingSpots*. Cela signifie que la précondition est vraie s'il y a au moins une voiture qui veut se garer et une place de parking qui est libre. Sans ces conditions, il n'y a rien à faire. Les variables *c* (ID d'une voiture à la recherche d'une place de parking) et *p* (ID d'une place qui est libre) sont instanciées par le moteur d'inférence.

Lorsque le moteur d'inférence a trouvé dans la précondition une paire de ressources correspondant à une *voiture* voulant se garer et une *place de parking* libre, la performance (la partie entre parenthèses)

est déclenchée pour les valeurs particulières des variables c (la voiture) et p (la place de parking). La performance utilise les opérations *get* et *put* pour consommer les ressources correspondant à l'état initial du système et insérer les nouvelles ressources qui décrivent le nouvel état du système. Ceci est fait dans une transaction, de façon atomique. Si la voiture décide de ne plus se garer ou la place de parking n'est plus libre, la transaction échoue. Le système reste dans un état cohérent décrit par le fait que la voiture ne veut plus se garer ou la place de parking n'est plus libre.

2.3 Réseau de Pétri coloré

Un réseau de Pétri [10] est composé de *places*, de *jetons*, de *transitions* et d'*arcs*. Les jetons sont contenus dans les places et peuvent être utilisés pour représenter les concepts du monde réel. Dans l'exemple du parking, un jeton peut représenter une voiture, son état ou son propriétaire.

Avec les réseaux de Pétri classiques (ordinaires et généralisés), les jetons ne peuvent pas être distingués. Ils n'ont pas de type. Cela peut être gênant lors de la modélisation d'un système dont les données doivent être distinguées. Par exemple, dans un parking, les places de stationnement sont différentes.

Pour contourner cette limitation, nous allons utiliser le réseau de Pétri coloré (RdPC) : une abréviation du réseau de Pétri classique. Le RdPC combine les fonctionnalités du réseau de Pétri avec les propriétés d'un langage de programmation pour permettre la distinction des jetons. Un type appelé couleur est associé à chaque place pour définir le type des jetons qu'elle peut contenir. Ainsi, les jetons peuvent être utilisés pour modéliser les données manipulées par un système.

L'objectif de cet article est d'utiliser le RdPC pour modéliser des systèmes à événements discrets et générer les règles LINC correspondantes. Les systèmes considérés sont constitués de nombreux composants (e.g. voitures et de places de parking). Dans LINC, ces composants sont représentés en utilisant des ressources. Une ressource est un tuple composé d'informations (e.g. ID, nom, type) et d'un état (i.e. l'état courant du composant). Dans ce travail, pour permettre la génération automatique de règles LINC à partir des RdPCs conçus, nous utilisons les jetons pour représenter à la fois l'état d'un composant et ses informations. Par conséquent, les couleurs sont définies sous la forme de tuples incluant un état et des informations. Par exemple, l'identifiant d'une place de parking et son état donnent *("parkingSpot12","free")*. Les places sont utilisées pour contenir, comme les sacs de LINC, des composants de même nature.

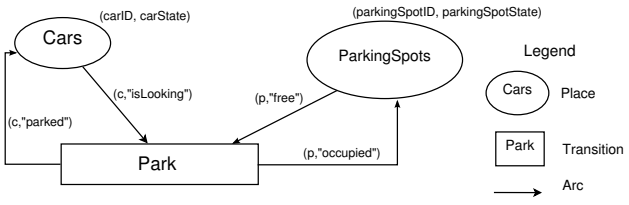


FIGURE 1 – Exemple de réseau de Pétri coloré

Le RdPC conçu pour modéliser l'exemple du parking est présenté à la Figure 1. Il est composé de deux places *Cars* et *ParkingSpots*, d'une transition *Park* et de quatre arcs. La place *Cars* contient des jetons représentant les différentes voitures. Sa couleur est définie sous la forme de tuple : *(carID, carState)* où *carID* est une chaîne caractères et *carState* est un type énuméré représentant les différents états d'une voiture. La place *ParkingSpots* contient des jetons qui représentent les différentes places de parking. Lorsqu'une voiture est à la recherche d'une place et qu'une place est libre, la transition *Park* est franchie. Elle modifie les états de la voiture et de la place de parking considérées. L'état de la voiture devient *parked* et l'état de la place est modifié à *occupied*. Les jetons *(c,"isLooking")* et *(p,"free")*

sont consommés et les nouveaux jetons *(c,"parked")* et *(p,"free")* sont insérés dans les places *Cars* et *ParkingSpots*. Les jetons sont consommés et insérés de façon atomique.

Le RdPC permet également de concevoir des modèles hiérarchiques. Un RdPC peut être une transition dans un RdPC qui est plus grand. Le RdPC hiérarchique (RdPCH) peut être utilisé pour modéliser le comportement global d'un système. Dans cet article, les places du RdPCH sont utilisées pour représenter les composants principaux du système. Les transitions sont des RdPCs et correspondent aux différentes actions que peuvent effectuer ces composants.

3 Nouvelle approche pour développer des applications LINC

L'approche proposée consiste à générer des règles LINC à partir de réseaux de Pétri colorés. L'application à développer est d'abord modélisée sous la forme de RdPCs. Cela permet les échanges entre les différents membres du projet, lors des premières phases de la conception. Ensuite, les RdPCs redéfinis, sont vérifiés pour valider le comportement de l'application. Enfin, les règles LINC correspondantes sont automatiquement générées à partir des modèles validés.

Pour permettre l'édition des RdPCs, au lieu d'utiliser les éditeurs graphiques tels que *CPN Tools* [15], nous avons défini un langage dédié [20] appelé cpnDSL. Une description de la syntaxe de ce langage est donnée au paragraphe 3.3. La principale raison d'utiliser le langage cpnDSL est d'éviter de gérer le placement manuel des différents éléments graphiques qui peut être lourd lorsque la taille du réseau de Pétri considéré devient grande.

La Figure 2 résume la contribution de cet article : à partir d'une description cpnDSL nous générons :

- une représentation graphique pour permettre la discussion entre les différents membres du projet, en particulier avec les personnes non techniques, experts dans le domaine de l'application ;
- un modèle vérifiable pour valider le comportement de l'application ;
- des règles, correctes par construction, directement exécutables dans LINC.

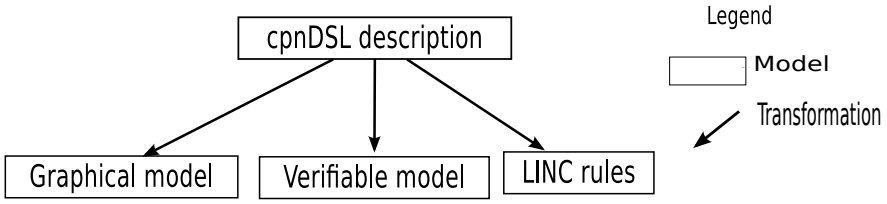
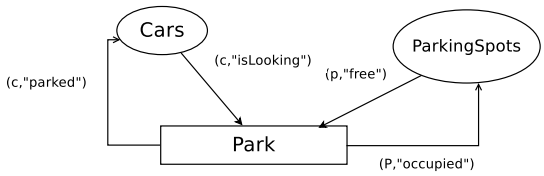


FIGURE 2 – L'approche proposée

L'approche proposée est maintenant détaillée, en commençant par la génération des règles LINC.

3.1 Génération des règles LINC

L'idée de générer des règles LINC à partir des modèles à base de réseaux de Pétri colorés vient de l'observation selon laquelle les deux présentent des similitudes. Considérons la Figure 3 qui présente l'exemple du parking. Le RdPC (Figure 3a) est composé d'une *transition Park* et deux *places Cars* et *ParkingSpots* contenant des *jetons* représentant respectivement les voitures et les places de parking. La



(a) Réseau de Pétri coloré (RdPC)

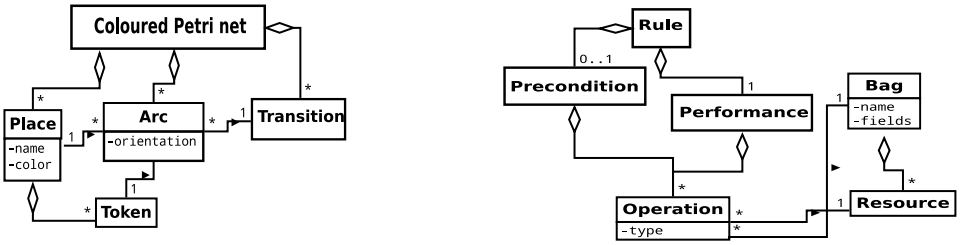
```
["Cars"].rd(c, "isLooking")&
["ParkingSpots"].rd(p,"free")                    2
::
{ ["Cars"].get(c, "isLooking");                  4
  ["Cars"].put(c, "parked");
  ["ParkingSpots"].get(p,"free");                6
  ["ParkingSpots"].put(p,"occupied");
}.                                               8
```

(b) Règle LINC

FIGURE 3 – Exemple d'un RdPC et d'une règle LINC

transition vérifie des conditions sur le système (i.e. une voiture à la recherche d'une place et une place libre) et modifie son état. La règle LINC (Figure 3b) utilise deux *sacs* Cars et ParkingSpots contenant des *ressources* représentant respectivement les voitures et les places de parking. Elle vérifie également les mêmes conditions et change l'état du système de la même manière.

La génération des règles LINC est basée sur les principes de la transformation de modèles [24]. Nous avons d'abord défini deux méta-modèles, un pour le RdPC et un autre pour les règles LINC. Puis, nous avons défini la transformation permettant de passer de l'un à l'autre.



(a) Méta-modèle du RdPC

(b) Méta-modèle des règles LINC

FIGURE 4 – Méta-modèles du RdPC et des règles LINC

Un méta-modèle définit les différents éléments qui peuvent être trouvés dans le modèle et leurs interactions. Le méta-modèle du RdPC est présenté à la Figure 4a. Les éléments sont *Place*, *Transition*, *Arc* et *Jeton*. La Figure 4b présente le méta-modèle des règles LINC. Il est composé de *sacs* contenant des *ressources*, de *règles* (composées d'une *précondition* et d'une *performance*) et d'*opérations* manipulant des ressources.

3.1.1 Transformation des réseaux de Pétri colorés en régles LINC

La transformation définie pour passer du RdPC aux règles LINC est résumée au Tableau 1a.

Jeton $\rightarrow$ **Ressource** Les couleurs des *jetons* ont été définies sous la forme de tuples incluant des informations sur les composants et leurs états. Cela permet d'associer un *jeton* à une *ressource*.

Place $\rightarrow$ **Sac** Une *place* d'un RdPC contient des jetons. Un *sac* de LINC contient des ressources. Par conséquent, une *place* est associée à un *sac*. Le nom d'une place et sa couleur sont utilisés pour définir le nom et les champs du sac correspondant.

Réseau de Pétri coloré	Règle LINC
Jeton	*Ressource*
Place nom couleur	*Sac* nom champs
Arc orientation PtoT BOTHDIR TtoP	*Opération* type get rd put
Transition	*Performance*
Arcs PtoT, BOTHDIR	*Précondition*

(a) RdPCs vers règles LINC

Règle LINC	Réseau de Pétri coloré
Ressource	*Jeton*
Sac nom champs	*Place* nom couleur
Opération type rd get put	*Arc* orientation BOTHDIR PtoT TtoP
Performance	*Transition*

(b) Règles LINC vers RdPCs

TABLE 1 – Transformations définies

Arc → Opération Les *arcs* sont associés aux *opérations*. Les arcs définissent si les jetons doivent être présents, consommés ou insérés lors du franchissement d'une transition. Les opérations de LINC sont utilisées pour vérifier la présence d'une ressource, la consommer ou l'insérer.

- Un arc orienté d'une place vers une transition (*PtoT*) spécifiant que le jeton doit être consommé est associé à l'opération *get* (pour consommer la ressource) ;
- Un arc orienté dans les deux sens (*BOTHDIR*) précisant que le jeton doit être présent est associé à l'opération *rd* (pour vérifier la présence de la ressource) ;
- un arc orienté d'une transition vers une place (*TtoP*) précisant que le jeton doit être inséré est associé à l'opération *put* (pour insérer la ressource).

Transition → Performance d'une règle Une *transition* est associée à une *performance*. Comme une transition, une performance assure l'exécution atomique d'une suite d'actions (en utilisant des transactions). Considérons l'exemple du parking. Le système change d'état lorsqu'une voiture est à la recherche d'une place de parking et qu'une place est libre. Pour changer l'état du système, la transition *Park* (Fig. 3a) consomme les jetons *(c,"isLooking")* et *(p,"free")* et insère les nouveaux jetons *(c,"parked")* et *(p,"occupied")* respectivement dans les places *Cars* et *ParkingSpots*, de façon atomique. Comme la transition, la performance de la règle (Fig. 3b après le symbole : :) consomme les ressources *(c,"isLooking")* et *(p,"free")* et insère les nouvelles ressources *(c,"parked")* et *(p,"occupied")*, de façon atomique.

Génération de la précondition d'une règle La précondition d'une règle spécifie, de façon explicite, que la performance peut être exécutée lorsque les ressources spécifiées sont présentes. Elle se compose d'un ensemble d'opérations *rd* pour vérifier la présence des ressources. Dans le RdPC, la notion de précondition est implicite, une transition est franchissable dès que les jetons spécifiés sont présents dans ses places d'entrée. La précondition est générée en utilisant l'opération *rd* pour vérifier que les ressources sont présentes. Ainsi, dans la précondition, il y a un *rd* pour chaque :

- arc *PtoT* (le jeton doit être consommé lors du franchissement de la transition) ;
- arc *BOTHDIR* (le jeton doit être présent lors du franchissement de la transition).

L'inverse de cette transformation (Tableau. 1a) permet de générer des RdPCs à partir des règles LINC.

3.1.2 Transformation des règles LINC en réseaux de Pétri colorés

La génération de RdPCs à partir des règles LINC est également utile. Elle offre la représentation graphique et les propriétés de vérification aux applications LINC qui ont été développées manuellement. Le Tableau 1b définit l'inverse de la transformation présentée précedemment. La principale différence est que pour une règle la *précondition* spécifie, de manière explicite, que la performance peut être exécutée lorsque les ressources indiquées sont présentes. Dans un RdPC, la notion de précondition est implicite. Ainsi, pour la génération d'un RdPC à partir d'une règle, la précondition n'est pas considérée.

3.2 Génération du modèle vérifiable

Les réseaux de Pétri peuvent être vérifiés à l'aide des model-checkers existants tels que LoLA [23], Marcie [13] et Helena [11]. Ils vérifient le blocage et d'autres propriétés (exprimées en logique temporelle) relatives au comportement dynamique du système (e.g. si un état qui satisfait une propriété donnée est toujours accessible). Certains model-checkers (e.g. Helena) supportent les réseaux de Pétri colorés et peuvent être utilisés pour vérifier les RdPCs considérés dans cet article. Cependant, ils permettent la vérification d'un nombre limité de propriétés, exprimées soit en logique temporelle linéaire LTL) ou en logique temporelle arborescente (CTL) et pas dans les deux [18]. D'autres model-checkers ne supportant pas les RdPCs (e.g. LoLA 2.0) peuvent être utilisés pour vérifier plusieurs propriétés (exprimées en LTL et CTL). De plus, le rapport du model-checking contest de 2014 [16] montre que les model-checkers donnent de meilleurs résultats sur les réseaux de Pétri classiques (pour l'exploration de l'espace d'état, l'accessibilité et les propriétés exprimées en CTL).

Dans ce papier, pour effectuer la vérification, un RdPC est d'abord automatiquement transformé en un réseau de Pétri classique (RdP). Ensuite, le RdP est vérifié en utilisant les model-checkers existants. Pour générer un RdP à partir d'un RdPC, les couleurs sont transformées en places. Une couleur est soit une chaîne de caractères (e.g. les IDs des voitures) soit une énumération (e.g. les états des voitures). Une couleur qui est une chaîne de caractères est transformée en une seule place. Une couleur énumérée est transformée en plusieurs places ; une place par énumérateur. Par exemple, la couleur *carStatus "isLooking"|"parked"* donne deux places *IsLooking* et *Parked*. Cette transformation permet de représenter explicitement les états du système qui étaient inclus dans les couleurs des jetons pour se rapprocher des ressources de LINC.

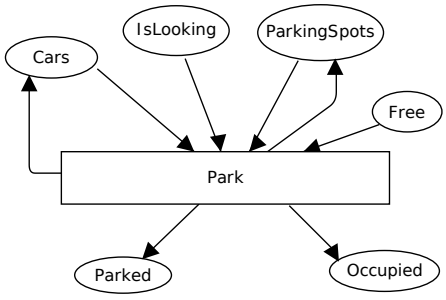


FIGURE 5 – Réseau de Pétri généré à partir du RdPC de la Figure 1

La Figure 5 présente le RdP généré à partir du RdPC de la Figure 1. Ces deux modèles ont le même comportement mais le RdP contient plus de places, correspondant aux couleurs du RdPC. Ce RdP peut être vérifié à l'aide des outils existants.

3.3 Le language cpnDSL

Plutôt que de dessiner les réseaux de Pétri colorés, nous avons défini un langage textuel pour les décrire : cpnDSL. La représentation graphique correspondant à une description textuelle est automatiquement générée. Dans cpnDSL, chaque élément du RdPC est décrit en utilisant un mot-clé correspondant à son nom. Par exemple, les places, les arcs et les transitions sont respectivement décrits en utilisant les mots-clés *Places*, *Arcs* et *Transitions*. Nous avons également fourni un outil d'édition du langage cpnDSL. Cet éditeur supporte la vérification de syntaxe, l'auto-complétion et la coloration syntaxique. Le langage cpnDSL et l'éditeur ont été développés en utilisant le framework Eclipse Xtext [12].

```
PetriNet CarParking
Colours
  name carID String name carState "isLooking"|"parked"
  name parkingSpotID String name parkingSpotState "free"|"occupied"

Places
   name Cars colour (carID,carState) tokens (c,"isLookingFor")
   name ParkingSpots colour (parkingSpotID,parkingSpotState) tokens (p,"free")

Transitions
    name Park
    Arcs
        type PtoT from Cars to Park inscription (c,"isLooking")
        type PtoT from ParkingSpots to Park inscription (p,"free")
        type TtoP from Park to Cars inscription (c,"parked")
        type TtoP from Park to ParkingSpots inscription (p,"occupied")
```

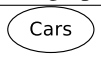
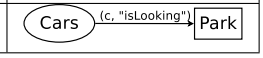
Listing 2 – Description cpnDSL de l'exemple du parking

Le Listing 2 présente la description cpnDSL du RdPC présenté à la Figure 1. Dans un premier temps, toutes les couleurs et les places sont décrites. Ensuite, les transitions sont définies, ici, il s'agit d'une seule transition (*Park*) avec quatre arcs. Un arc est décrit en utilisant les mots-clés *type*, *from*, *to* et *inscription* pour définir respectivement son orientation, son origine, sa destination et son inscription (i.e. le jeton lié à l'arc). L'orientation d'un arc est soit *PtoT* (d'une place à une transition), *TtoP* (d'une transition à une place) ou *BOTHDIR* (dans les deux sens).

3.4 Génération du modèle graphique

À partir d'une description cpnDSL, nous générons la représentation graphique du RdPC correspondant au format SVG (Scalable Vector Graphic). Chaque élément du langage cpnDSL (i.e. place, transition et arc) est associé à une notation graphique (Cf.Tableau 2).

TABLE 2 – Notation graphique des éléments de cpnDSL

cpnDSL	Notation graphique
`Places name Cars colour (carID,carState)`	
`Transitions name Park`	
`Arcs type PtoT from Cars to Park inscription (c,"isLooking")`	

Pour la génération des fichiers SVG, le langage de description de graphes DOT [17] a été utilisé comme langue pivot. Une spécification cpnDSL est d'abord transformée en une description DOT. Ensuite, celle-ci est exécutée, en utilisant le programme *en ligne de commande dot*, pour générer une image SVG dans laquelle les éléments graphiques sont automatiquement placés. Un exemple de fichier SVG généré est présenté dans l'étude de cas (Figure 7b).

4 Étude de cas

Cette section illustre l'approche proposée à travers une étude de cas. Nous avons décidé d'améliorer un démonstrateur de smart parking actuellement déployé dans nos locaux. Ce démonstrateur intègre des bornes de recharge et des capteurs capables de détecter les voitures. Il gère la surveillance, la comptabilité et fournit des statistiques sur l'utilisation des places de parking et des bornes de recharge.

L'objectif de cette étude de cas est d'ajouter une nouvelle fonctionnalité au démonstrateur : la réservation des places de parking pour aider un conducteur de voiture à trouver une place de parking libre. Dans le processus normal, le conducteur d'une voiture va à la place qui lui est réservée et se gare. Cependant, il est possible qu'un autre conducteur, n'utilisant pas le système ou ne respectant les règles, occupe la place. Dans ce cas, la présence de la "mauvaise" voiture est détectée et le système réaffecte une autre place à la voiture initiale. Lorsqu'une voiture garée s'en va, le système considère la place comme étant à nouveau libre.

Pour implémenter la réservation des places de parking, nous l'avons d'abord décrite en utilisant le langage cpnDSL. Ensuite, les modèles graphiques ont été générés pour la discussion entre les différents membres du projet. Cela a permis d'affiner le modèle et de prendre en compte des aspects qui ont été initialement ignorés. Une fois d'accord sur le comportement du système, les règles LINC correspondantes ont été générées. Elles étaient correctes par construction et ont été directement exécutées sans modification. Nous détaillons les différentes étapes dans les paragraphes qui suivent.

4.1 Description du comportement du système

La partie logique du système de réservation des places de parking est constituée de deux composants : les *voitures* et les *places de parking*. Une voiture a cinq états : *driven*, *looking for a parking spot*, *assigned to a parking spot*, *going to a parking spot* et *parked*. Une place de parking a trois états : *reserved*, *occupied* et *free*. Une voiture peut effectuer les actions suivantes : *ask for a parking spot*, *reserve a parking spot*, *go to a parking spot*, *park* et *leave*.

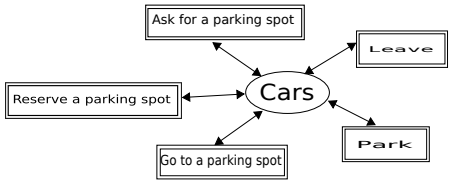


FIGURE 6 – RdPC hiérarchique modélisant le comportement global du système de réservation

La Figure 6 présente le RdPC hierarchique correspondant au comportement global du système de réservation de places. Il contient une seule place *Cars* et cinq transitions représentant les différentes actions que les voitures peuvent effectuer. Chaque transition est un RdPC.

4.2 Génération des modèles graphiques et des règles LINC

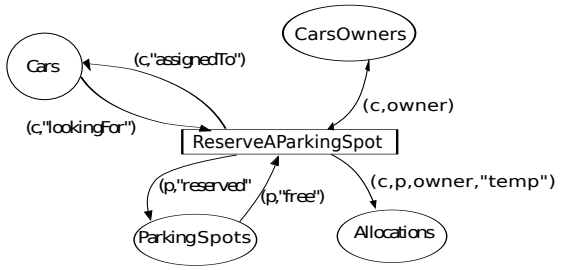
Pour chaque transition du modèle global, nous avons d'abord décrit le RdPC correspondant avec le langage cpnDSL, puis nous avons généré la représentation graphique et les règles LINC.

La Figure 7 présente sur le côté gauche (7a) la description cnpDSL de l'action *Reserve a parking spot*, et sur le côté droit (7b), le fichier SVG généré. La règle LINC générée est présentée à la Figure 8b.

Contrairement à la règle, dans le fichier SVG, les différents composants (e.g. les voitures, leurs propriétaires, les places de parking) et leurs changements d'état sont explicitement représentés en connec-

```
1  PetriNet ReserveAParkingSpot
   Colours
3   name carID String name carState "lookingFor"|"assignedTo"
          name carOwner String
    name parkingSpotID String name parkingSpotState
          "free"|"reserved"
5   name allocationType String
   Places
7     name Cars colour (carID,carState)
      name ParkingSpots colour (parkingSpotID,parkingSpotState)
9     name CarsOwners colour (carID,CarOwner)
      name Allocations colour
           (carID,parkingSpotID,carOwner,allocationType)
11 Transitions
       name ReserveAParkingSpot
13     Arcs
           type PtoT from Cars to ReserveAParkingSpot inscription
                (c,"lookingFor")
15         type PtoT from ParkingSpots to ReserveAParkingSpot
                inscription (p,"free")
           type BOTHDIR from CarsOwners to ReserveAParkingSpot
                inscription (c,owner)
17         type TtoP from ReserveAParkingSpot to Cars inscription
                (c,"assignedTo")
           type TtoP from ReserveAParkingSpot to ParkingSpots
                inscription (p,"reserved")
19         type TtoP from ReserveAParkingSpot to Allocations
                inscription (c,p,owner,"temp")
```

(a) Description cpnDSL



(b) Fichier SVG généré

FIGURE 7 – Description cpnDSL et fichier SVG généré

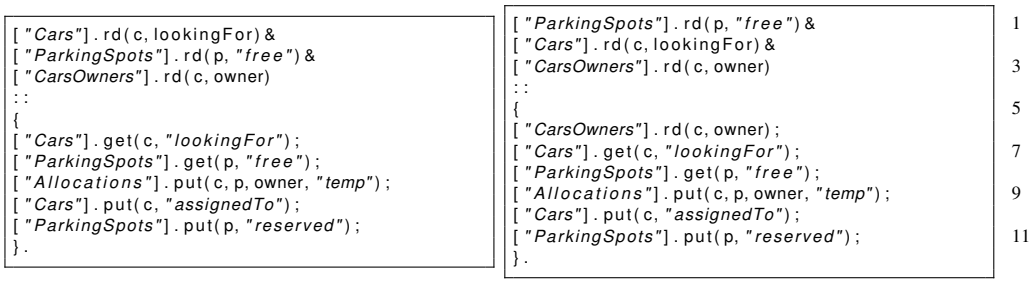
tant des éléments graphiques. Quelqu'un (sans connaissances en coordination et LINC) peut facilement voir que l'action *ReserveAParkingSpot* requiert une voiture cherchant une place de parking, le propriétaire de la voiture, une place libre et associe la voiture à la place de parking tout en modifiant leurs états. Grâce à la notation graphique, les RdPCs sont plus appropriés à la discussion avec des personnes non techniques que règles LINC. Ainsi, une fois générées, ces représentations graphiques ont été utilisées pour discuter et convenir sur le comportement du système. Cela a été utile pour décider par exemple si c'était mieux de modéliser les interactions en considérant la voiture comme étant l'élément central plutôt que la place de parking.

Un autre avantage du RdPC (Fig. 7b) par rapport à la règle (Fig. 8b) est qu'il permet de connaître le comportement du système avant son exécution, en vérifiant le modèle. Considérons l'action *ReserveAParkingSpot* et supposons que l'on oublie que l'état d'une voiture à la recherche d'une place (*looking for*) doit être changé (à *assignedTo*). Dans ce cas, lors de l'écriture manuelle de la règle correspondante, nous omettons l'opération *put* (Ligne 10). Lors de l'élaboration du RdPC, nous n'avons pas un arc orienté de la transition *ReserveAParkingSpot* à la place *Cars*. Cela va complètement changer le comportement du système (l'état de la voiture n'est pas connu et une place lui est réservée). Pour la règle, ce problème est détecté au moment de l'exécution, lorsque des bugs sont observés. En vérifiant le RdPC, ce problème peut être détecté plus tôt. La vérification consiste à générer le RdP correspondant et à vérifier que le système se comporte normalement (i.e. si une voiture est à la recherche d'une place et qu'une place est libre alors elle sera toujours attribuée à la voiture). Cette vérification a été effectuée en utilisant le model-checker LoLA [23] et le résultat a été négatif.

4.3 Comparaison d'une règle générée à une règle écrite manuellement

Les règles générées sont lisibles et compréhensibles par un humain. Ainsi, nous avons décidé de les comparer aux règles qui auraient pu être écrites à la main. Nous avons demandé à un membre du projet d'écrire les règles à partir des représentations graphiques. La Figure 8 présente la règle générée et celle écrite manuellement pour la transition *ReserveAParkingSpot*.

Deux différences peuvent être notées. Dans la précondition, l'ordre des opérations n'est pas le même.



(a) Règle écrite manuellement

(b) Règle générée

FIGURE 8 – Comparaison d'une règle générée à une règle écrite manuellement

La règle écrite manuellement (Listing 8a) vérifie d'abord qu'une voiture est à la recherche d'une place (Ligne 1) avant de chercher une place libre (Ligne 2). Dans la règle générée (Listing 8b) cela est fait dans l'ordre inverse. Pour le programmeur, il est plus logique de vérifier d'abord qu'une voiture est à la recherche d'une place. En effet, il n'y a pas besoin de chercher une place si aucune voiture ne veut se garer. Cela permet de réduire la taille de l'arbre qui est construit par le moteur d'inférence lors de l'évaluation de la précondition.

Dans la performance, la règle générée vérifie avec l'opération *rd()* que le *propriétaire de la voiture* n'a pas changé (Ligne 6). La règle écrite manuellement ne vérifie pas cette condition parce que le programmeur sait que le propriétaire de la voiture (connu dans la partie précondition) ne change pas dans le processus.

Le programmeur dispose de connaissances supplémentaires qui lui permettent d'optimiser les règles en conséquence. Ces types d'informations ne sont pas actuellement représentés dans les RdPCs, par conséquent, ils ne sont pas pris en compte dans les règles générées. Pour surmonter cela, nous pourrions, par exemple, annoter les RdPCs. Étant donné que les règles générées sont lisibles par un être humain, il est également possible de les optimiser manuellement une fois générées.

En conclusion, la description des RdPCs à l'aide d'un langage textuel (cnpDSL) qui est traité par des outils robustes conçus pour dessiner de grands graphes permet d'envisager des systèmes complexes. La représentation graphique est une véritable aide pour affiner le problème à résoudre, surtout en présence de personnes non techniques (non familières avec LINC) qui sont plus concernées par le domaine de l'application. Enfin, les règles générées donnent les résultats escomptés quand elles sont exécutées, mais, actuellement, elles ne sont pas optimisées. La génération automatique est presque immédiate, alors que l'écriture des règles à la main prend plus de temps et requiert quelques phases de débogage simples mais nécessaires.

5 Travaux connexes

Dans [26], les auteurs présentent une approche pour vérifier les systèmes orientés événements, décrits à l'aide du langage DeraDSL. Un réseau de Pétri représentant l'état courant du système est généré lors de la conception ou à l'exécution. Ce RdP est alors vérifié à l'aide des outils existants. Dans notre approche, le cpnDSL décrivant le système est transformé en un RdP qui est vérifié une seule fois, au moment de la conception. Le cnpDSL validé est utilisé pour générer des règles de coordination correctes assurant l'exécution normale du système. La vérification n'est pas nécessaire à l'exécution parce que le moteur d'exécution de LINC préserve les propriétés vérifiées au niveau des RdPs .

Dans [25], les auteurs proposent une méthode permettant de générer des logiciels de protocoles à partir de réseaux de Pétri colorés. Ils ajoutent des annotations syntaxiques (i.e. pragmatiques) sur les modèles pour permettre la génération de code. Dans notre approche, les similarités entre les RdPCs et les règles LINC ont permis de ne pas ajouter des pragmatiques sur les modèles, pour permettre la génération des règles.

Dans [8], les auteurs proposent un framework pour modéliser des applications distribuées et générer le code correspondant sous la forme de composants Java (JavaBeans). Pour permettre la modélisation, ils ont défini un langage de spécification appelé CO-OPN. Il est basé sur les types de données algébriques, le réseau de Pétri et le modèle de coordination IWIM. À partir des spécifications CO-OPN, ils génèrent des composants Java. Certains aspects du langage CO-OPN et des systèmes distribués tels que l'atomicité et le non-déterminisme n'ont pas été faciles à implémenter. Ils ne sont pas supportés par le langage Java. Les composants générés doivent être connectés ou composés pour obtenir l'application finale. Ils peuvent aussi être intégrés dans une application existante en utilisant des outils dédiés. Dans notre approche, nous avons utilisé le réseau de Pétri coloré (RdPC) pour modéliser des applications distribuées et nous avons généré le code correspondant en utilisant un langage de coordination (i.e. LINC) qui supporte différents aspects des systèmes distribués (e.g. l'atomicité et non-déterminisme). Ainsi, la génération de code a été simple. Chaque concept du RdPC a un équivalent dans LINC. Les applications générées peuvent être exécutées directement sans intervention.

6 Conclusion

Ce document a présenté une approche pour générer des règles LINC à partir de réseaux de Pétri colorés (RdPCs). Il a défini un langage dédié appelé cpnDSL pour permettre la description textuelle des RdPCs. Cela permet aux utilisateurs de ne pas gérer le placement manuel des différents éléments graphiques. Une application à développer est d'abord décrite en utilisant le langage cpnDSL. La représentation graphique correspondante est automatiquement générée et est utilisée pour discuter sur le comportement du système, éventuellement avec des personnes non techniques. La description cpnDSL est ensuite utilisée pour valider le comportement de l'application, en utilisant les outils de vérification existants. Enfin, la description cpnDSL vérifiée est utilisée pour générer des règles LINC. Ces règles peuvent être directement exécutées dans le moteur de règles LINC sans aucune modification.

Nous avons illustré l'intérêt de l'approche proposée à travers une étude de cas qui a consisté à l'amélioration d'un système existant (i.e. un démonstrateur de smart parking). Dans cette étude de cas, nous avons comparé les règles générées à celles écrites manuellement. Cela a montré que les deux types de règles ont le même comportement. Cependant les règles générées peuvent être optimisées si certaines connaissances utilisées par le programmateur sont rajoutées sur les réseaux de Pétri colorés. Cette optimisation est la première partie de nos futurs travaux. La deuxième partie sera axée sur le contrôle des réseaux de Pétri [14], pour imposer au modèles, les propriétés désirées. En effet, la vérification permet de savoir si un modèle satisfait ou pas certaines propriétés mais ne fait pas en sorte qu'elles soient vérifiées. Si une propriété souhaitée n'est pas satisfaite, il faut modifier le modèle en utilisant le contre-exemple donné (éventuellement) par le model-checker, et effectuer à nouveau la vérification. Lors de l'utilisation des techniques de contrôle, si une solution (qui fait en sorte que le modèle vérifie la propriété) existe, elle sera automatiquement trouvée et appliquée. Elle peut prendre la forme de places de contrôle ajoutées au RdP, selon une approche suivie par exemple dans un contexte différent par le projet Gadara [1].

Références

[1] *10th International Workshop on Discrete Event Systems*. International Federation of Automatic Control, 2010.

[2] J.P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the chemical reaction model : Fifteen years after. In *Multiset Processing*. Springer, 2001.

[3] J. Barbosa, F. Dillenburg, G. Lermen, A. Garzão, C. Costa, and J. Rosa. Towards a programming model for context-aware applications. *Computer Languages, Systems & Structures*, 2012.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987.

[5] M. Bortenschlager, G. Castelli, A. Rosi, and F. Zambonelli. A context-sensitive infrastructure for coordinating agents in ubiquitous environments. *Multiagent and Grid Systems*, 2009.

[6] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 1989.

[7] C.G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Springer, 2008.

[8] S. Chachkov and D. Buchs. From formal specifications to ready-to-use software components : the concurrent object oriented petri net approach. In *Application of Concurrency to System Design. International Conference on Proceedings.* IEEE, 2001.

[9] T. Cooper. *Rule-based programming under OPS5*. Morgan Kaufmann Publishers Inc., 1988.

[10] R. David and H. Alla. Petri nets for modeling of dynamic systems : A survey. *Automatica*, 1994.

[11] S. Evangelista. High level petri nets analysis with helena. In *Applications and Theory of Petri Nets*. Springer, 2005.

[12] M. Eysholdt and H. Behrens. Xtext : implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010.

[13] M. Heiner, C. Rohr, and M. Schwarick. Marcie–model checking and reachability analysis done efficiently. In *Application and Theory of Petri Nets and Concurrency*. Springer, 2013.

[14] L.E. Holloway, B.H. Krogh, and A. Giua. A survey of petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems*, 1997.

[15] K. Jensen, L.M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 2007.

[16] F. Kordon et al. Html results from the model checking contest petri net. http ://mcc.lip6.fr/2014, 2014.

[17] E. Koutsofios, S. North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.

[18] K. Lautenbach. Verifiable design of a satellite-based train control system with petri nets. 2014.

[19] M. Louvel and F. Pacull. Linc : A compact yet powerful coordination environment. In *Coordination Models and Languages*. Springer, 2014.

[20] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 2005.

[21] F. Pacull, L-F. Ducreux, et al. Self-organisation for building automation systems : Middleware linc as an integration tool. In *Industrial Electronics Society, IECON 39th Annual Conference of the IEEE*, 2013.

[22] G.A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in computers*, 1998.

[23] K. Schmidt. Lola a low level analyser. In *Application and Theory of Petri Nets*. Springer, 2000.

[24] S. Sendall and W. Kozaczynski. Model transformation the heart and soul of model-driven software development. Technical report, EPFL - Microsoft, 2003.

[25] K.I. Simonsen, L.M. Kristensen, and E. Kindler. Pragmatics annotated coloured petri nets for protocol software generation and verification. Technical report, Technical University of Denmark, 2014.

[26] H. Tran and U. Zdun. Event actors based approach for supporting analysis and verification of event-driven architectures. In *Enterprise Distributed Object Computing Conference, 17th IEEE International*, 2013.