Systèmes concurrents hiérarchiques : équivalence, analyse et structuration

Pierre Bouvier

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Systèmes concurrents hiérarchiques : équivalence, analyse et structuration

Hierarchical concurrent systems: equivalence, analysis and structuring

Présentée par :

Pierre BOUVIER

Direction de thèse :

Hubert GARAVEL Directeur de recherche, Centre Inria de l'Université Grenoble Alpes	Directeur de thèse
Radu MATEESCU Directeur de recherche, Centre Inria de l'Université Grenoble Alpes	Co-directeur de thèse

Rapporteurs :

Yann THIERRY-MIEG
MAITRE DE CONFERENCES HDR, Sorbonne Université
François VERNADAT
PROFESSEUR DES UNIVERSITES, INSA de Toulouse

Thèse soutenue publiquement le **12 octobre 2023**, devant le jury composé de :

Hubert GARAVEL DIRECTEUR DE RECHERCHE, Centre Inria de l'Université Grenoble Alpes	Directeur de thèse
Radu MATEESCU DIRECTEUR DE RECHERCHE, Centre Inria de l'Université Grenoble Alpes	Co-directeur de thèse
Yann THIERRY-MIEG MAITRE DE CONFERENCES HDR, Sorbonne Université	Rapporteur
François VERNADAT PROFESSEUR DES UNIVERSITES, INSA de Toulouse	Rapporteur
Mihaela SIGHIREANU PROFESSEUR DES UNIVERSITES, ENS Paris-Saclay	Examinatrice
Mnacho ECHENIM PROFESSEUR DES UNIVERSITES, Grenoble INP	Président

Abstract

In the context of the verification of concurrent systems, we study Petri nets and one of their extensions, NUPNs (for "Nested-Unit Petri nets"), which provide them modularity and hierarchy through a structure of sequential processes, nested in the form of a tree.

We propose various techniques to address several questions concerning Petri nets and NUPNs. First, we present approaches to decide efficiently the isomorphism of Petri nets and NUPNs. Then, we define algorithms to detect dead places and dead transitions, which enable the elimination of dead code, as well as algorithms to detect concurrent places, which provide information about the concurrency of nets. Finally, we tackle the automated translation of Petri nets into networks of communicating automata, into NUPNs with nested processes, or into different process algebras. These algorithms have been implemented and tested on large collections of nets from academia, industry, or stemming from international competitions.

Résumé

Dans le cadre de la vérification des systèmes concurrents, nous étudions les réseaux de Petri et une de leurs extensions, les NUPN (acronyme anglais de « réseaux de Petri à unités imbriquées »), qui leur apportent modularité et hiérarchie par une structure en processus séquentiels, imbriqués sous la forme d'un arbre.

Nous proposons diverses techniques permettant de résoudre plusieurs questions relatives aux réseaux de Petri et aux NUPN. D'abord, nous exposons des approches pour décider efficacement l'isomorphisme des réseaux de Petri et des NUPN. Puis, nous définissons des algorithmes pour détecter les places mortes et les transitions mortes, ce qui permet l'élimination du code mort, ainsi que des algorithmes pour détecter les places concurrentes, qui donnent des informations sur la concurrence des réseaux. Enfin, nous abordons la traduction automatisée des réseaux de Petri en réseaux d'automates communicants, en NUPN avec des processus imbriqués, ou en différentes algèbres de processus. Ces algorithmes ont été implantés et testés sur de grandes collections de réseaux d'origine académique, industrielle, ou provenant de compétitions internationales.

Remerciements

Je tiens au préalable à remercier Hubert Garavel pour le remarquable encadrement qu'il m'a apporté, découlant de sa gentillesse hors du commun, de son optimisme, de son expérience, de ses grandes capacités de synthèse et de son impressionnant souci du détail. Il m'a donné l'opportunité d'explorer de nombreuses pistes de recherches, en portant sur elles un regard à la fois pertinent, critique et constructif, ce qui m'a incité à innover. La pertinence et la profondeur des nombreux conseils qu'il a su me distiller font qu'ils dépassent considérablement le cadre de ce doctorat.

Le soutien de Radu Mateescu a été déterminant pour le bon déroulement de ces travaux et à l'élaboration de cette thèse. Je le remercie chaleureusement pour son optimisme, sa bienveillance, sa souplesse d'organisation et son regard aussi original qu'avisé.

Je suis reconnaissant envers Yann Thierry-Mieg et Francois Vernadat, ainsi qu'envers Mnacho Echenim et Mihaela Sighireanu, pour avoir accepté d'examiner mon travail et pour prendre part à la soutenance de cette thèse.

Merci à Nicolas Amat, Fabrice Kordon, Lom Messan-Hillah et Hernán Ponce de León pour l'ouverture, la disponibilité et l'amabilité dont ils ont fait preuve lors de nos collaborations.

Faire partie de l'équipe CONVECS, formée de membres croyant si vigoureusement en ce qu'ils font et d'une telle virtuosité est autant une satisfaction qu'un honneur. Merci à eux pour leur accueil, pour leur esprit de cohésion et pour la bonne ambiance qui en résulte.

Je suis grandement redevable envers Ajay, Lina, Maxime, Myriam, Philippe et Supriya, qui m'ont aidé à surmonter des épreuves difficiles et avoir transformé mon regard sur le monde.

Je crains que je ne ferais jamais suffisamment preuve de gratitude pour mon aimante famille, en particulier envers mon père, pour m'avoir transmis sa curiosité, envers ma mère, pour son dévouement exigeant et envers mon frère, dont la sagesse laisserait suggérer que c'est lui, le grand frère.

Certaines expériences ont été effectuées sur la plateforme Grid'5000, soutenue par un groupe d'intérêt scientifique regroupant l'Inria, le CNRS, le RENATER et d'autres universités et organisations (voir www.grid5000.fr).

Table des matières

Première partie

Introduction

Chapitre 1

Introduction aux systèmes concurrents hiérarchiques

La concurrence est une notion-clef en informatique, apparaissant naturellement lors de l'élaboration de nombreux systèmes. Mais son appréhension est un défi pour le cerveau humain, qui a des difficultés à suivre les évolutions simultanées des systèmes concurrents.

Les méthodes formelles sont des techniques visant à faciliter la mise au point de systèmes fiables et complexes, en reposant sur des outils mathématiques rigoureux. Cette thèse s'inscrit dans la vérification de modèles, qui propose des formalismes et outils visant à faciliter la modélisation, l'analyse et la vérification de tels systèmes.

Parmi ces formalismes figurent les réseaux de Petri, introduits en 1962, qui sont populaires car ils facilitent la représentation précise et abstraite de concepts aussi essentiels que le parallélisme, la synchronisation, le non-déterminisme, les conflits, etc.

Cette thèse vise à explorer les systèmes concurrents hiérarchiques, à travers le concept des NUPN, une extension des réseaux de Petri introduite en 2015, qui leur apporte les notions de modularité et de hiérarchie, au moyen d'unités représentant des processus.

Ce chapitre est destiné à présenter le contexte et les notions utiles pour bien comprendre les systèmes concurrents hiérarchiques, puis à aborder les contributions de cette thèse.

1.1 Systèmes séquentiels, parallèles et concurrents

Dans un système informatique, un composant *séquentiel* (aussi appelé, selon le contexte, programme séquentiel, processus, tâche, fil d'exécution, automate, etc.) est fondamentalement constitué d'états et de transitions. Par exemple, en langage d'assemblage, les états et transitions s'assimilent à des positions du compteur de programme et à l'exécution d'instructions. Le lancement d'un programme séquentiel correspond à son *initialisation* depuis un état dit *initial*. Partant d'un état, une transition peut être exécutée (on dit aussi : tirée ou franchie), menant vers un nouvel état. Un programme séquentiel ne peut être que

dans un seul état à la fois. Il peut se terminer en atteignant un état dit *final*.

Un système *concurrent* est constitué d'un ensemble de sous-composants séquentiels s'exécutant indépendamment mais coopérant pour accomplir un objectif commun. Un système *parallèle* est un système exécutant simultanément plusieurs opérations. Les deux termes « concurrent » et « parallèle » sont certes voisins, mais ne sont pas synonymes : un système peut très bien à la fois être concurrent et non parallèle (par exemple, un système d'exploitation multitâche préemptif tournant sur un unique processeur n'ayant qu'un seul fil d'exécution), ou être parallèle et non concurrent (par exemple, un processeur vectoriel exécutant une instruction en parallèle sur plusieurs nombres scalaires sans recourir, d'un point de vue externe, à une quelconque forme de concurrence).

Enfin, les termes « concurrent » et « parallèle » doivent être distingués de la notion de système *distribué*, qui désigne un système réparti dans l'espace, où les différents composants sont exécutés par des machines situées dans des endroits différents. La proximité de l'utilisation de ces termes dans le langage courant s'explique par le fait que l'informatique parallèle et l'informatique distribuée reposent sur les fondements théoriques de la concurrence pour étayer leurs développements.

La concurrence joue un rôle à la fois ancien et majeur dans l'informatique. Elle est autant liée à des impératifs de performance et de disponibilité qu'à la nature intrinsèquement concurrente des systèmes informatiques. Les attentes en termes de performance, de fiabilité et de coûts vis-à-vis de systèmes toujours plus complexes sont croissantes, ce qui rend le recours à la concurrence à la fois omniprésent et critique, aussi bien pour la recherche et l'industrie.

Un réseau d'ordinateurs est l'archétype d'un système distribué, parallèle et concurrent. Cela ne concerne non seulement le lancement de calculs à distance (telle que la production d'images de synthèse dans des fermes de rendu), mais également la gestion de tâches distribuées (telles que les bases de données accédées simultanément par des millions d'utilisateurs). Cette informatique distribuée a donné naissance à l'informatique en nuage, constituée de plates-formes reposant sur la virtualisation, proposant des moyens de calcul et d'hébergement à la demande. L'évolution de ces systèmes répond à des contraintes diverses, qui imposent le recours à la concurrence. Ils doivent : (i) fournir une abstraction des systèmes utilisés ; (ii) être résilients face aux erreurs et aux pannes de machines ou de réseaux et (iii) avoir la capacité à monter en charge en fonction du nombre d'utilisateurs, de leur localisation et de leurs demandes.

Les circuits logiques sont, eux aussi, intrinsèquement parallèles et concurrents, fait exploité par les concepteurs de puces afin d'améliorer leurs performances. Depuis plusieurs décennies, les concepteurs de processeurs ont notamment recours à un large éventail de techniques (exécution pipelinée, superscalaire, dans le désordre, spéculative, etc.) pour exécuter en parallèle des instructions, tout en masquant cette complexité à leurs utilisateurs, qui ne perçoivent que des traces d'exécutions séquentielles. De manière similaire, les concepteurs de matériels doivent proposer des circuits toujours plus performants et efficaces. La progression des procédés lithographiques permet certes un « budget transistors » toujours croissant,

mais les progrès des transistors sur le plan des performances séquentielles sont désormais limités : ces deux points impliquent une utilisation toujours plus grande de la concurrence et du parallélisme pour améliorer les circuits. De ce fait, les concepteurs intègrent toujours davantage de composants, qui sont de plus en plus hétérogènes et possèdent des interconnexions complexes, telles que des réseaux sur puces. En outre, la concurrence et le parallélisme sont de moins en moins cachés aux utilisateurs, qui doivent maintenant composer avec des processeurs multicœurs (éventuellement hétérogènes), des hiérarchies mémoire complexes et parfois des processeurs exposant explicitement le parallélisme des instructions.

Enfin, les systèmes embarqués et cyber-physiques combinent, d'une part, une multitude de capteurs et d'actionneurs et, d'autre part, des contrôleurs et processeurs effectuant des calculs complexes. Ces composants sont interconnectés et interagissent ensemble, de manière parallèle et concurrente, afin de remplir des fonctions voulues par les utilisateurs finaux.

1.2 Mémoire partagée et communication par messages

Pour accomplir un objectif commun, les composants séquentiels d'un système concurrent doivent être en mesure de synchroniser leurs actions et d'échanger des données. Il est possible de distinguer deux grands paradigmes pour cela :

— Le premier consiste en une *mémoire partagée* associée à un mécanisme permettant aux processus d'accéder ou d'écrire de manière cohérente dans cette mémoire. Ce premier principe est le plus ancien [Dij65, Dij68], car il s'agit d'une réutilisation des méthodes de programmation séquentielle pour les programmes concurrents, éventuellement exécutés sur des machines mono-processeurs. Ce paradigme est utilisé dans les systèmes POSIX ainsi que dans de nombreux langages de programmation. Il existe une large palette de mécanismes pour contrôler les opérations sur la mémoire ; certains sont de bas niveau, tels que les verrous et les sémaphores, tandis que d'autres sont de plus haut niveau, tels que les moniteurs et les mémoires transactionnelles logicielles. Malheureusement, ces mécanismes sont d'emploi délicat, car leur mauvaise utilisation conduit aisément à des situations de corruption de la mémoire ou d'interblocages.

— Le second principe repose sur le *passage de messages*, qui remonte aux travaux concernant *nucleus* [Han70, Han02], le premier micro-noyau de système d'exploitation et concernant l'algèbre de processus CSP [Hoa78]. Le passage de messages est décliné en de nombreuses variantes. Il peut être binaire (limité à deux processus) ou n-aire (applicable à un nombre arbitraire de processus). Il peut être *unidirectionnel* (lors de l'échange, un seul processus émet une valeur) ou être *multidirectionnel* (les processus s'accordent mutuellement sur une valeur commune). Il peut être réalisé par un *rendez-vous* [Hoa78] (afin de pouvoir effectuer l'échange, les processus impliqués atteignent chacun des états spécifiques), ou non. Il peut être *synchrone* (un processus émetteur attend la confirmation de réception du message) ou *asynchrone*.

1.3 Vérification formelle pour les systèmes concurrents

Conceptuellement, le fonctionnement d'un système séquentiel est relativement simple à comprendre : pour en déterminer son comportement, il suffit de modéliser les actions réalisées par chacune de ses transitions, puis de déterminer le *flot de contrôle* du système, c'est-à-dire l'enchaînement des transitions lors de l'exécution du système.

À l'inverse, la concurrence est très mal appréhendée par les concepteurs de systèmes, puisque nos cerveaux sont inadaptés à suivre les évolutions multiples et simultanées de plusieurs processus exécutés en concurrence : non seulement l'ordonnancement des composants séquentiels d'un système concurrent n'est pas unique et peut varier entre différentes exécutions, mais les transitions d'un processus donné dépendent aussi des transitions déjà exécutées et en cours d'exécution par les autres processus. De ce fait, il devient nécessaire de considérer qu'à l'exécution, tous les entrelacements entre processus sont possibles, à l'échelle de leurs transitions individuelles, ce qui augmente considérablement la complexité.

Cette difficulté conduit à des erreurs de conception et d'implantation difficiles à détecter ou à corriger, qu'il s'agisse de non-conformité aux spécifications, d'interblocages, de situations de compétition, de famines et ainsi de suite. Ces erreurs non-corrigées sont redoutables en raison de leurs conséquences coûteuses, et parfois tragiques, dans le monde réel. Or, on exige davantage de sûreté et de sécurité de la part des systèmes informatiques, dont leur importance est désormais capitale.

Pour réduire le risque d'erreurs liées à la concurrence, il est fréquent que les concepteurs n'exploitent pas tous les bénéfices possibles du parallélisme. Mais sans pour autant parvenir à éliminer totalement ce risque.

D'où la naissance de *méthodes formelles* (pour une synthèse, voir des ouvrages récents [WLBF09] [GG13] [BH14] [GtBvdP20]) qui répondent à ces problématiques, en proposant, d'une part, des formalismes permettant la modélisation des systèmes concurrents et, d'autre part, des outils permettant de vérifier les modèles créés pour découvrir automatiquement les erreurs ou obtenir une preuve de leur absence. Les approches de vérification formelle se classent selon trois grandes familles :

— La *vérification de modèles* [CHVB18], dans laquelle une exploration exhaustive de tous les états possibles que le système est susceptible de prendre au cours de son exécution est réalisée, ce qui produit un *système de transition d'états*, qui est un graphe orienté fini, aussi appelé *espace d'états*. Ce graphe est ensuite utilisé pour valider certaines propriétés, généralement décrites sous la forme de formules en logique temporelle.

— L'*interprétation abstraite*, qui effectue une abstraction simplificatrice de la sémantique d'un système, permettant ainsi de calculer une sur-approximation de son comportement au moyen d'un calcul de point fixe. Ceci permet de prouver automatiquement l'absence de certaines erreurs, ou de garantir certaines propriétés comportementales sans avoir à lancer le programme. Les outils Astrée [CCF+05], CPAchecker [BK11] et Verasco [JLB+15] s'inscrivent dans cette famille.

— La *preuve de théorèmes*, où un assistant de preuve (par exemple, Coq [CH88] ou Isabelle [Pau89, NP92]) tente de produire une preuve mathématique démontrant le respect d'une spécification par le système. Cette preuve est composée d'une description du système, d'un ensemble d'axiomes logiques et d'un ensemble de règles d'inférence.

Nous optons dans ce travail pour la vérification de modèles, car il s'agit d'une famille bien adaptée à la vérification de systèmes concurrents. En effet, elle permet de décrire, puis d'analyser, toutes les traces d'exécution possibles, résultant de l'exécution concurrente de multiples processus.

1.4 Modélisation formelle des systèmes concurrents

Pour les besoins de la vérification de modèles, de nombreux formalismes mathématiques très divers permettant de décrire des systèmes concurrents ont été introduits.

Les systèmes *synchrones*[1] sont un modèle très étudié (notamment en France), intermédiaire entre les systèmes séquentiels et concurrents. Ils reposent sur l'hypothèse d'une horloge partagée entre tous les composants séquentiels, lesquels changent d'états internes et se synchronisent au gré de cette horloge. Une transition élémentaire de l'ensemble du système est la combinaison des transitions de tous les composants séquentiels lors d'une impulsion de l'horloge. Des exemples de formalismes asynchrones sont Esterel [BG92], Lustre [HCRP91] et Signal [BLJ91].

La concurrence synchrone est pertinente dans nombre de cas (il s'agit notamment d'une restriction pratique permettant de réduire le risque d'erreurs liées à la concurrence), mais le recours systématique à une horloge partagée entre tous les composants concurrents n'est pas raisonnable. En effet, de nombreux systèmes, dont les systèmes distribués [Lam78], reposent sur des composants séquentiels évoluant indépendamment, sans partager d'horloge. Nous nous orientons de ce fait vers un autre modèle, sans horloge partagée, qui correspond à celui des systèmes *asynchrones*. Notons que, si besoin est, la description d'une horloge partagée par certains processus reste possible au sein d'un modèle asynchrone.

Les formalismes permettant la modélisation, l'analyse et la vérification des systèmes concurrents asynchrones se classent selon trois grandes familles, qui remontent des années soixante aux années quatre-vingts :

— Les *automates communicants* [Arn82] [Pet90] sont un formalisme simple permettant de représenter un système concurrent sous la forme d'un ensemble d'automates, qui peuvent communiquer à l'aide de transitions *synchronisées* entre plusieurs automates (autrement dit, il s'agit de transitions non-*locales*). Ceci permet la réalisation de certaines constructions essentielles des systèmes concurrents, dont figurent entre autres les barrières, les synchronisations et les rendez-vous, qui reposent sur une modification des flots de contrôle des processus.

1. Cette notion de parallélisme synchrone est différente du passage synchrone de messages, vu ci-dessus.

Les automates communicants possèdent en outre des représentations graphiques, plus accessibles aux utilisateurs non experts que des descriptions mathématiques abstraites.

Enfin, notons que certaines variantes intègrent des structures de données, par exemple les *automates temporisés* ajoutent la notion de temps par le biais des horloges [BDL+11].

— Les *réseaux de Petri* [Pet77, Rei85, Mur89], introduits en 1962, reposent sur un petit nombre de concepts simples qui permettent la description abstraite des principes de la concurrence, tels que le parallélisme, le non-déterminisme, la synchronisation et les conflits.

L'expressivité, la sémantique formellement définie et la présence d'une représentation graphiques des réseaux de Petri en font un modèle de choix dans un grand nombre de domaines, qu'il s'agisse des télécommunications, de l'ingénierie système, des circuits, des logiciels, sans oublier d'autres disciplines comme la biologie et la chimie.

En ce sens, il existe un format normalisé dédié à leur description, le langage de balisage de réseaux de Petri (PNML, en anglais « Petri Net Markup Language ») [ISO11] et de nombreux logiciels consacrés aux réseaux de Petri ont été développés.

La pertinence des réseaux de Petri est confirmée par le dynamisme des compétitions de vérification, en particulier du « Model Checking Contest » [ABC+19, KHHH+21], qui repose sur eux en guise de formalisme compris par tous les outils participants, au moyen du format normalisé PNML.

— Les *langages algébriques* reposant sur la théorie des *calculs de processus*, introduits en 1978 par Tony Hoare, dont les principaux représentants sont CSP [Hoa85], CCS [Mil89], ACP [BK85] et LOTOS [ISO89].

Ces langages de programmation proposent de modéliser des systèmes en décrivant leurs transitions élémentaires et leurs liens, ce qui est fait en imbriquant (récursivement) des opérateurs décrivant des actions exécutées en séquence, des actions exécutées en parallèle, des choix opérés entre plusieurs branches, et ainsi de suite.

Cette phrase d'Ed Brinksma résume un des points forts de ces langages : « les algèbres de processus ont été inventées pour ne pas avoir à débattre au sujet des états » [2].

Pour permettre la modélisation, puis l'analyse et la vérification de systèmes réels complexes pouvant être de grande taille, ces grandes familles représentent, pour les développeurs d'outils de vérification de modèles, des formalismes qui sont complémentaires et non pas exclusifs, aussi bien d'un point de vue mathématique que pratique.

Par exemple, pour vérifier une propriété sur une spécification écrite dans le langage LNT [GLS17], la boîte à outils CADP (en anglais « Construction and Analysis of Distributed Processes ») [GLMS13], développée au centre Inria de l'Université Grenoble Alpes,

2. En anglais : « Process algebra has been invented not to discuss about states ».

compile d'abord la spécification donnée en LOTOS, puis en un réseau de Petri *interprété* [Gar89, GS90]. L'espace d'états de ce réseau est ensuite exploré (plus précisément ses *marquages accessibles*, dans la terminologie des réseaux de Petri), ce qui permet finalement de générer un graphe spécifique (plus exactement, un *système de transitions étiquetées*), lequel est exploité pour vérifier des formules en logique temporelle décrivant la ou les propriétés considérées.

Cette thèse est consacrée à l'exploration des réseaux de Petri, notamment pour les besoins des algèbres de processus. Nous allons présenter les notions de modularité et de hiérarchie, qui sont héritées des automates communicants et des algèbres de processus.

1.5 Réseaux de Petri et modularité

Les automates communicants représentent des processus concurrents et synchronisés. Les réseaux de Petri traditionnels sont quant à eux dépourvus de toute notion de processus.

La notion de modularité, qui permet la distinction des différents processus composant un système concurrent, ainsi que les opérations possibles pour chacun d'entre eux est pourtant primordiale pour modéliser correctement certains de ces systèmes.

Ceci est particulièrement vrai lorsque les processus sont de nature hétérogène. Cette distinction peut s'imposer pour des raisons de sécurité, par exemple lors d'un échange de clefs en cryptographie ou dans le cadre d'un modèle client-serveur.

Elle peut aussi s'avérer nécessaire pour des raisons physiques : au sein d'un objet connecté, un actionneur mécanique n'est pas piloté par un serveur central, mais par un de ses ordinateurs embarqués ; de même, les opérations relatives à une zone mémoire privée ne peuvent être réalisées que par le processus possédant cette mémoire.

Dans la mesure où les transitions peuvent être synchronisées entre plusieurs processus, elles ne fournissent pas une base simple pour définir les processus dans un réseau de Petri. Il semble plus naturel de se reposer sur les places.

Dans un réseau de Petri, un marquage (c'est-à-dire un ensemble de *jetons*, contenus dans les *places*) correspond à un état global du système : d'un point de vue pratique, il s'agit l'union des états de tous les processus. Puisqu'un processus est un composant séquentiel, il ne peut pas être simultanément dans plusieurs états : cela se traduit par le fait qu'il ne possède au plus qu'un seul jeton à la fois, ce qui définit un état local du processus considéré.

Dans la suite de ce document, nous considérons alors qu'un processus est constitué par un ensemble de places, qui représentent tous les états potentiellement accessibles du processus. De ce fait, pour décrire les processus, il est logique d'utiliser des « boîtes » englobant les places (mais pas les transitions) du réseau.

1.6 Réseaux de Petri et hiérarchie

Cette section vise à mettre en valeur les apports de la notion d'organisation hiérarchique des processus pour la représentation des systèmes concurrents. Notons que le besoin d'organiser hiérarchiquement des processus est ancien : il apparait notamment dans [Han70].

Nous avons vu dans la section 1.1 que les notions de concurrence et de parallélisme, bien que liées, sont distinctes. Pour les concepteurs d'un système, la concurrence est un moyen permettant de le décrire de manière abstraite, comme étant un ensemble de sous-systèmes interagissant entre eux, tandis que le parallélisme les amène à considérer l'implantation concrète de ce système : sera-il exécuté sur plusieurs machines ? Combien de processeurs l'exécuteront ? Concrètement, un processus « physique » d'un système réel peut être l'implantation de plusieurs processus « logiques » dans son modèle correspondant, sous réserve que dans ce processus physique, à tout instant, un seul de ces processus logiques peut être en cours d'exécution.

Cette différence entre processus « logique » et « physique » s'illustre lors de la création de processus enfants dans les systèmes d'exploitation POSIX. Conceptuellement, un processus parent invoque un ensemble de processus enfants, puis attend leur terminaison. Ceci est implanté en pratique par l'utilisation conjointe des primitives POSIX *fork* et *exec*, qui ne permettent l'invocation que d'un seul processus enfant, sans interrompre le processus parent. Ce comportement est un détail d'implantation valable seulement pour POSIX et ne contribue pas à une modélisation plus précise.

Cette primitive fork tend à effacer l'organisation hiérarchique entre un processus parent et ses processus enfants, qui deviennent alors concurrents avec leur parent. Cette hiérarchie permet pourtant aux concepteurs de modéliser plus librement des systèmes complexes, en évitant la production de modèles « spaghettis », difficiles à relire et à comprendre. La hiérarchie est également utile pour les outils de vérification, qui peuvent obtenir statiquement des informations plus précises sur les processus pouvant potentiellement s'exécuter de manière concurrente, ou non.

Ainsi, un concepteur de systèmes devrait pouvoir décrire des modèles raffinés, présentant autant de processus « logiques » que possible, organisés de manière hiérarchique, de sorte à pouvoir exprimer librement l'imbrication d'opérations séquentielles et d'opérations concurrentes. Cette organisation hiérarchique des processus correspond à ce que proposent les algèbres de processus, en autorisant la composition d'opérateurs imbriqués. Les automates communicants et les réseaux de Petri traditionnels ne permettent pas de telles constructions.

Pour décrire la hiérarchie des processus au sein des réseaux de Petri, les « boîtes » envisagées dans la section 1.5 doivent donc pouvoir être imbriquées les unes dans les autres, de sorte à former un arbre.

Pour exprimer proprement l'organisation modulaire en composants et sous-composants séquentiels des systèmes concurrents, deux processus imbriqués ne doivent pas pouvoir être concurrents. Dans un réseau de Petri, ceci correspond au fait que la présence d'un jeton dans une « boîte » implique l'absence de jeton dans toutes les « boîtes » (récursivement)

englobantes ou contenues.

1.7 Le modèle NUPN

Le réseau de Petri produit par la boîte à outils CADP (voir la section 1.4) est dit « interprété » car il est enrichi d'informations de plus haut niveau qui seraient perdues avec un réseau de Petri traditionnel : (i) une partie données avec des variables typées et des annotations sur les transitions ; (ii) une structure en *unités* représentant des composants séquentiels, encapsulant et regroupant les places ; ces unités sont récursivement imbriquées, formant ainsi un arbre enraciné.

Si l'on omet la partie données de ce réseau de Petri interprété mais que l'on préserve sa structure en unités, on obtient précisément les NUPN [Gar19, Gar15] (en anglais « Nested-Unit Petri Nets »), une extension récente (2015) des réseaux de Petri ordinaires et saufs.

Cette structure en unités ne change ni les définitions, ni les propriétés des réseaux de Petri, incluant celles liées à leur sémantique dynamique, mais leur apporte la notion manquante de processus.

En permettant de représenter ces informations relatives aux processus, les NUPN sont également utiles pour les phases ultérieures d'analyse et de vérification ; ceci autorise en particulier une exploration plus efficace de leurs espaces d'états, en réduisant le nombre de variables utilisées dans les diagrammes de décision binaire.

Concrètement, dans le formalisme NUPN, une unité représente un composant séquentiel et encapsule des places. À toute place est attribuée une unique unité : il n'existe pas de place « partagée ». Mais les unités peuvent être (récursivement) imbriquées. À ce propos, il existe une unité spéciale, l'*unité racine*, qui englobe toutes les autres unités. La structure en unités forme ainsi un arbre enraciné. Ainsi, le formalisme NUPN permet d'exprimer les systèmes concurrents et hiérarchiques.

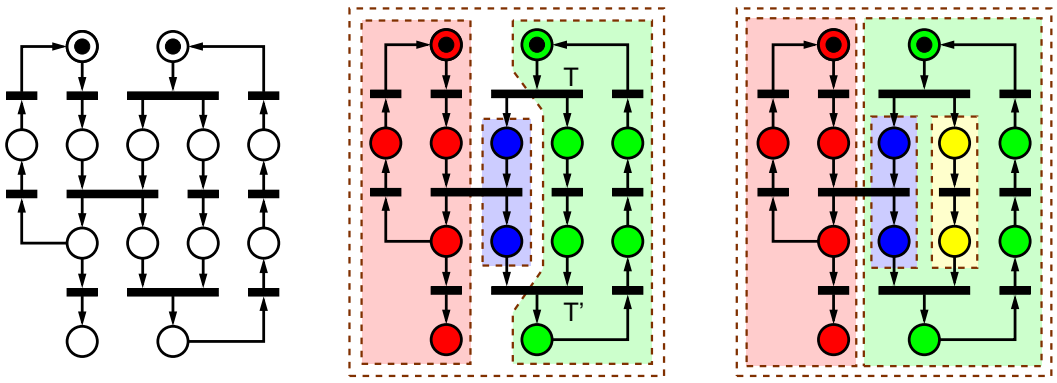

FIGURE 1.1 – Réseau de Petri (à gauche) représenté avec des processus, soit sans hiérarchie (au milieu), soit avec hiérarchie (à droite)

La figure 1.1 illustre à sa gauche un réseau de Petri, qui ne présente pas la notion de

processus, ainsi que deux NUPN équivalents. Graphiquement, chaque unité est représentée par une boîte, matérialisée par un rectangle en pointillés.

Le NUPN en partie centrale de la figure ne permet pas de distinguer les processus parents des processus enfants et les transitions T et T' semblent être des synchronisations entre frères alors qu'il s'agit (respectivement) d'une « fourchette » et d'une « jointure » impliquant un processus parent avec ses enfants. À l'inverse, celui de la partie droite de la figure représente bien la hiérarchie des processus par l'imbrication des boîtes : on perçoit quatre processus, un rouge, un vert, un bleu et un jaune ; on remarque également que le processus vert est parent des processus bleu et jaune ; enfin, notons que les processus bleu et jaune sont concurrents.

1.8 Contributions

Dans ce travail, nous proposons d'étudier les systèmes concurrents hiérarchiques à travers les réseaux de Petri, et plus particulièrement à travers le concept des NUPN, qui pour rappel, enrichissent les réseaux de Petri d'une structure additionnelle décrivant des processus concurrents organisés hiérarchiquement.

Ce mémoire de thèse est organisé en quatre parties, intitulées comme suit : (I) introduction, (II) équivalence, (III) analyse et (IV) structuration.

La première partie, intitulée **introduction**, s'articule en trois chapitres :

- Le chapitre 1 est ce présent chapitre.
- Le chapitre 2 présente les notations mathématiques adoptées dans ce document, rappelle les définitions essentielles pour les graphes et pour les réseaux de Petri.
- Le chapitre 3 présente quant à lui le formalisme NUPN, son environnement logiciel, ainsi que les jeux de tests utilisés par la suite.

La deuxième partie, intitulée **équivalence**, est constituée du chapitre 4, qui caractérise la notion d'équivalence de réseaux de Petri ou de NUPN, en terme d'isomorphisme. Ce chapitre propose ensuite une série d'approches diverses pour l'identification de réseaux isomorphes ou non-isomorphes au sein de quatre collections de jeux de tests, la plus importante regroupant plus de 241 000 réseaux, dont de nombreux « doublons ». Ceci permet d'évaluer la diversité d'une collection existante, d'en supprimer les doublons, ou encore d'en extraire le plus grand sous-ensemble possible de réseaux « uniques ».

La troisième partie, intitulée **analyse**, est consacrée à l'analyse de propriétés relatives à la concurrence au sein des réseaux de Petri, plus exactement, au calcul de trois problèmes d'accessibilité pour les réseaux ordinaires et saufs :

- Le chapitre 5 vise la détermination des places mortes (ensemble des places jamais marquées) et des transitions mortes (ensemble des transitions jamais activées),

correspondant en programmation à du code mort.

— Le chapitre 6 se destine au calcul du problème des places concurrentes, donnant l'ensemble des paires de places pouvant être simultanément marquées, cette relation binaire apporte des contraintes utilisées par les chapitres suivants pour reconstituer les unités manquantes des réseaux de Petri.

La quatrième partie, intitulée **structuration**, vise à reconstruire automatiquement la structure hiérarchique manquante des réseaux de Petri, afin de produire les NUPN et les algèbres de processus correspondants :

— Le chapitre 7 décrit des approches diverses pour structurer tout réseau de Petri ordinaire et sauf en un réseau d'automates communicants, sous la forme d'un NUPN plat (autrement dit, sans unités imbriquées).

— Le chapitre 8 vise à reconstituer un NUPN hiérarchique (avec des unités imbriquées), depuis un réseau de Petri ordinaire et sauf, ou bien depuis un NUPN possédant une structure en unités sous-optimale (par exemple, à plat).

— Le chapitre 9 propose un schéma de traduction allant d'un NUPN vers les algèbres de processus, où les unités sont retranscrites en processus imbriqués, où les places deviennent des états locaux et où les transitions peuvent librement être synchronisées entre les processus.

En plus de posséder un réel intérêt pratique, cette approche originale, consistant à traduire les réseaux de Petri en algèbres de processus, en passant (éventuellement) par les automates communicants, devrait permettre de rapprocher les communautés de ces trois familles de formalismes.

Finalement, le chapitre 10 donne les conclusions de cette thèse et trace des perspectives pour de futures recherches.

Chapitre 2

Définitions et notations

Ce chapitre présente quelques notations mathématiques, puis aborde des définitions liées aux graphes et aux réseaux de Petri, que nous utiliserons dans la suite de cette thèse.

2.1 Notations mathématiques

Cette section présente les notations utilisées par la suite de ce document, concernant les opérateurs logiques, les fonctions, les ensembles, les multi-ensembles et les n-uplets.

Définition 2.1 *Concernant les* opérateurs logiques*, nous définissons :*

- $\top$ *la proposition* vraie *;*
- $\bot$ *la proposition* fausse *;*
- $\perp\!\!\!\top$ *la proposition* indéfinie *(en logique ternaire)*[1] *;*
- $\neg P$ *la* négation *de la proposition* P *;*
- $P \vee Q$ *la* disjonction *des propositions* P *et* Q *;*
- $P \wedge Q$ *la* conjonction *des propositions* P *et* Q *;*
- $P \Rightarrow Q$ *l'*implication *logique, c.-à-d.* $(\neg P) \vee Q$ *;*
- $P \Leftrightarrow Q$ *l'*équivalence *logique, c.-à-d.* $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Définition 2.2 *Concernant les* fonctions*, nous définissons :*

- $f \circ g$ *la* composée *de la fonction* g *par la fonction* f *;*
- $\mathsf{Id} \stackrel{\text{def}}{=} x \mapsto x$ *la fonction* identité *;*
- f^{-1} *la* réciproque *de la bijection* f*, autrement dit telle que* $f \circ f^{-1} \stackrel{\text{def}}{=} \mathsf{Id}$ *;*

1. $\perp\!\!\!\top$ est à la fois la lettre phénicienne Zayin, la lettre Zeta en grec archaïque et la superposition des graphies de $\top$ et $\bot$. Mais il faut distinguer cette lettre du Z utilisée en électronique pour caractériser la notion de haute impédance.

— $x \mapsto \log_2(x)$ *le* logarithme binaire *;*

— $x \mapsto \lceil x \rceil$ *la fonction* partie entière par excès.

Définition 2.3 *Concernant les* ensembles*, nous définissons :*

— $\varnothing$ *l'*ensemble vide *;*

— $\mathbb{N}$ *l'ensemble des* entiers naturels *;*

— $\{x_1, \ldots, x_n\}$ *le plus petit ensemble contenant chacun des éléments* x_1 *à* x_n [2] *;*

— $\{m : M\}$ *l'*intervalle *des entiers naturels de* m *à* M*, c.-à-d.* $\{k \in \mathbb{N} \mid m \leq k \leq M\}$ *;*

— $E \subseteq E'$ *le prédicat* E est sous-ensemble de E'*, c.-à-d.* $(\forall x \in E, x \in E')$ *;*

— $E \subset E'$ *le prédicat* E est sous-ensemble strict de E'*, c.-à-d.* $(E \neq E') \wedge (E \subseteq E')$ *;*

— $E \cap E'$ *l'*intersection *des ensembles* E *et* E' *;*

— $E \cup E'$ *l'*union *des ensembles* E *et* E' *;*

— $E \uplus E'$ *l'*union disjointe *des ensembles* E *et* E'*, c.-à-d.* $E \cup E'$*, défini* ssi $E \cap E' = \varnothing$ *;*

— $E \setminus E'$ *la* différence *des ensembles* E *et* E'*, c.-à-d.* $\{x \in E \mid x \notin E'\}$ *;*

— $|E|$ *le* cardinal *de l'ensemble* E *;*

— $\sum_{x \in E} \mathsf{f}(x)$ *la* somme *des* $\mathsf{f}(x)$ *tels que* $x \in E$*, où* f *est une fonction et* E *un ensemble* [3] *;*

— $\prod_{x \in E} \mathsf{f}(x)$ *le* produit *des* $\mathsf{f}(x)$ *tels que* $x \in E$*, où* f *est une fonction et* E *un ensemble* [4] *;*

— $\mathbb{1}_E : x \mapsto |E \cap \{x\}|$ *la fonction* indicatrice *de l'ensemble* E*, à valeurs dans* $\{0, 1\}$ *;*

— $E \mapsto \mathsf{choix}(E)$ *une fonction tirant un élément quelconque de* E*.*

Définition 2.4 *Concernant les* ensembles ordonnés*, nous définissons :*

— $\# : E \mapsto (\#_E : E \to \mathbb{N}^*)$ *est une fonction donnant à tout ensemble* E *une fonction* $\#_E$*, bijective de* E *vers* $\{1 : |E|\}$*, c.-à-d. une fonction (déterministe) donnant un ordre quelconque sur ses éléments* [5] *;*

— $\mathsf{min}(E)$ *le plus petit élément de l'ensemble* E *selon l'ordre* $\#_E$ *;*

— $\mathsf{max}(E)$ *le plus grand élément de l'ensemble* E *selon l'ordre* $\#_E$ *;*

— $x <_E y \stackrel{\text{def}}{=} (\#_E(x) < \#_E(y))$ *la relation binaire, définie* ssi $x, y \in E$ *et découlant* [6] *de l'ordre* $\#_E$*.*

Définition 2.5 *Concernant les* multi-ensembles*, nous définissons :*

— $\{\!|\, x_1, \ldots, x_n \,|\!\}$ *le multi-ensemble constitué des éléments* x_1 *à* x_n*, représenté de manière formelle par la fonction suivante :* $x \mapsto \mathbb{1}_{\{x_1\}}(x) + \ldots + \mathbb{1}_{\{x_n\}}(x)$ *;*

2. Ainsi, les occurrences multiples d'un élément sont autorisées, mais ignorées, c.-à-d. $\{x, x\} = \{x\}$.
3. En particulier, nous avons : $\forall \mathsf{f}, \sum_{x \in \varnothing} \mathsf{f}(x) = 0$.
4. En particulier, nous avons : $\forall \mathsf{f}, \prod_{x \in \varnothing} \mathsf{f}(x) = 1$.
5. À noter que : $\forall E, \forall E', (E = E') \Leftrightarrow (\#_E = \#'_E)$.
6. Le symbole E sera omis lorsque l'ordre est évident (par exemple, l'ordre des entiers naturels pour $\mathbb{N}$).

— $\mathrm{dom}_M \stackrel{\text{def}}{=} \{x \mid M(x) > 0\}$ *le* domaine *du multi-ensemble* M ;

— $M \subseteq M'$ *le prédicat* M est sous-multi-ensemble de M', *c.-à-d.* $\forall x, M(x) \leq M'(x)$;

— $M \subset M'$ *le prédicat* M est strictement inférieur à M', *c.-à-d.* $\forall x, M(x) < M'(x)$;

— $M + M'$ *la somme des multi-ensembles* M *et* M', *concrètement, il s'agit de la somme de leurs fonctions correspondantes ;*

— $M - M'$ *la différence des multi-ensembles* M *et* M', *c.-à-d. le multi-ensemble* M'' *tel que* $M = M' + M''$, *défini* ssi $M \geq M'$;

Définition 2.6 *Concernant les* n-uplets *(aussi appelés* vecteurs, *ou* listes*), nous définissons :*

— $(x_1, \ldots, x_n)$ *l'*uplet *de* dimension n *(aussi appelé* n-uplet*), constitué des éléments allant de* x_1 *à* x_n ;

— $\bot_{(n)}$ *l'*n-uplet nul, *autrement dit* $\bot_{(n)} \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$, *tel que* $\forall i \in \{1 : n\}, x_i \stackrel{\text{def}}{=} \bot$;

— $U[i]$ *le* $i^{\text{ème}}$ *élément*[7] *de l'*n-uplet U ;

— $U[m; M] \stackrel{\text{def}}{=} (U[m], \ldots, U[M])$ *l'opérateur* tranche, *qui retourne un vecteur contenant tous les éléments du* n-uplet U *ayant un indice compris dans* $\{m : M\}$;

— $U_1 \,\&\, U_2 \stackrel{\text{def}}{=} (U_1[1] \wedge U_2[1], \ldots, U_1[|U_1|] \wedge U_2[|U_1|])$ *l'opérateur bit-à-bit* `et`, *qui a pour argument deux* n-uplets U_1 *et* U_2 *de même dimension et à valeurs dans* $\{\bot, \top\}$.

Définition 2.7 *Concernant les* produits, *nous définissons :*

— $E \times E' \stackrel{\text{def}}{=} \{(x, y) \mid (x \in E) \wedge (y \in E')\}$ *le* produit cartésien *des ensembles* E *et* E' ;

— $E \otimes E' \stackrel{\text{def}}{=} \{\{x, y\} \mid (x \in E) \wedge (y \in E')\}$ *un produit des ensembles* E *et* E', *dérivé du produit cartésien, où les couples sont remplacés par des paires et des singletons, autrement dit, en remplaçant des éléments ordonnés par des éléments non-ordonnés.*

2.2 Graphes

Nous rappelons ensuite les définitions usuelles de *graphe*, de graphe *coloré*, de *clique* et d'*isomorphisme* de graphes. En effet, de nombreux problèmes relatifs aux réseaux de Petri et à la vérification de modèles s'expriment et se résolvent avantageusement par la théorie des graphes.

2.2.1 Graphes non-orientés et cliques

Définition 2.8 *Un* graphe non-orienté *est un couple* (V, E) *tel que :*

— V *est appelé un ensemble de sommets, dont nous supposerons, sans restreindre la généralité, que* $V \cap \mathbb{N} = \varnothing$;

7. En particulier, nous avons : $\forall i \in \{1 : n\}, (x_1, \ldots, x_n)[i] = x_i$.

— $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V\}$ *est appelé un ensemble d'arêtes.*

Définition 2.9 *Soit un graphe (V, E) non-orienté. Alors :*

— *deux sommets $v_1, v_2 \in V$ reliés par une arête $\{v_1, v_2\} \in E$ sont dits* adjacents *;*

— *un sommet adjacent à lui-même est une* boucle *;*

— *le graphe est dit* connexe ssi *tout couple de sommets est transitivement adjacent ;*

— *un* cycle *est un sous-graphe possédant un sommet transitivement adjacent à lui-même ;*

— *une* clique *est un sous-ensemble de sommets du graphe dont tous les éléments sont mutuellement adjacents, c.-à-d. $C \subseteq V$ et $\{\{v_1, v_2\} \in (C \otimes C) \mid v_1 \neq v_2\} \subseteq E$;*

— *un* stable *est un sous-ensemble de sommets du graphe dont tous les éléments sont mutuellement non-adjacents, c.-à-d. $I \subseteq V$ et $\{\{v_1, v_2\} \in (I \otimes I) \mid v_1 \neq v_2\} \cap E = \varnothing$;*

— *une clique C de (V, E) est* maximale *s'il n'existe pas de clique C' telle que $C \subset C'$;*

— *une clique C de (V, E) est* maximum *si pour toute clique C', $|C'| \leq |C|$.*

2.2.2 Graphes étiquetés, colorés et isomorphes

Définition 2.10 *Un* graphe non-orienté et étiqueté sur ses sommets (*graphe étiqueté en abrégé), est un triplet (V, E, l) tel que :*

— *(V, E) est un graphe non-orienté ;*

— *$\mathsf{l} : V \to \mathbb{N}$ est une fonction surjective sur un intervalle d'entiers incluant* 0.

Définition 2.11 *Soit un graphe (V, E) non-orienté. Alors :*

— *le graphe étiqueté (V, E, c) est dit* coloré ssi *la fonction c vérifie la propriété suivante $\forall v_1, v_2 \in V, (v_1 \neq v_2) \wedge (\{v_1, v_2\} \in E) \Rightarrow \mathsf{c}(v_1) \neq \mathsf{c}(v_2)$; on dit alors que c associe à chaque sommet une* couleur *;*

— *le graphe est dit k-*coloriable *(avec $k \in \mathbb{N}$) s'il existe un graphe coloré (V, E, c') dont la fonction c' a pour co-domaine $\{0 : k-1\}$;*

— *le* nombre chromatique *d'un graphe donné est le plus petit entier naturel k tel que le graphe soit k-coloriable.*

Définition 2.12 *Deux graphes étiquetés $G = (V, E, \mathsf{l})$ et $G' = (V', E', \mathsf{l}')$ sont* isomorphes ssi *il existe une bijection $\pi_v : V \to V'$ telle que :*

— $\forall v_1, v_2 \in V, \{v_1, v_2\} \in E \Leftrightarrow \{\pi_v(v_1), \pi_v(v_2)\} \in E'$.

— $\forall v \in V, \mathsf{l}(v) = \mathsf{l}'(\pi_v(v))$.

La présente définition s'applique aussi aux graphes non-étiquetés, en retirant la dernière ligne.

2.2.3 Graphes orientés

Définition 2.13 *Un* graphe orienté *est un couple* (V, A) *tel que :*

- V *est appelé un ensemble de sommets, tel que* $V \cap \mathbb{N} = \varnothing$ *;*
- $A \subseteq \{(v_1, v_2) \mid v_1, v_2 \in V\}$ *est appelé un ensemble d'arcs.*

Définition 2.14 *Soit un graphe* (V, A) *orienté. Alors :*

- *deux sommets* $v_1, v_2 \in V$ *reliés par une arc* $(v_1, v_2) \in A$ *sont dits* adjacents *;*
- *un sommet adjacent à lui-même est une* boucle *;*
- *un* cycle *est un sous-graphe possédant un sommet transitivement adjacent à lui-même ;*
- *une* forêt *est un graphe ne possédant pas de cycle ;*
- *un* arbre *(aussi appelé* arbre enraciné*) est une forêt dont un de ses sommets est transitivement adjacent à tous les autres sommets ;*
- *un* arbre couvrant *de* (V, A) *est un arbre* (V, A') *tel que* $A' \subseteq A$.

Définition 2.15 *Un* graphe orienté et étiqueté sur ses sommets (*graphe orienté étiqueté en abrégé), est un triplet* (V, A, l) *tel que :*

- (V, A) *est un graphe orienté ;*
- $\mathsf{l} : V \to \mathbb{N}$ *est une fonction surjective sur un intervalle d'entiers dont le plus petit élément est* 0.

Définition 2.16 *Deux graphes orientés étiquetés* $G = (V, A, \mathsf{l})$ *et* $G' = (V', A', \mathsf{l}')$ *sont* isomorphes ssi *il existe une bijection* $\pi_v : V \to V'$ *telle que :*

- $\forall v_1, v_2 \in V, (v_1, v_2) \in A \Leftrightarrow (\pi_v(v_1), \pi_v(v_2)) \in A'$.
- $\forall v \in V, \mathsf{l}(v) = \mathsf{l}'(\pi_v(v))$.

La présente définition s'applique aussi aux graphes orientés non-étiquetés, en retirant la dernière ligne.

2.3 Réseaux de Petri

Nous rappelons les définitions usuelles concernant les réseaux de Petri et aiguillons le lecteur vers les références classiques [Pet77, Rei85, Mur89] pour une présentation plus détaillée.

2.3.1 Réseaux ordinaires initialement marqués

Un réseau de Petri est défini graphiquement comme un graphe orienté et biparti, dont les sommets sont soit des places, soit des transitions. Les places sont représentées par des cercles et les transitions par des segments. Il existe deux types d'arcs : ceux reliant des places à des transitions et ceux reliant des transitions à des places. La figure 2.1 donne un exemple de réseau de Petri contenant 13 places (en noir), 11 transitions (en bleu) ainsi que

26 arcs (en rouge). Remarquons que dans cette figure, deux places possèdent un point en leur centre (en vert) : ces points représentent des *jetons* et les places les englobant sont dites *initialement marquées*, ou plus simplement *initiales*.

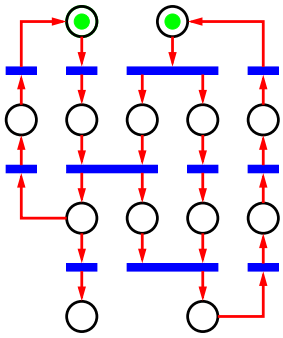

FIGURE 2.1 – Un réseau de Petri ordinaire et initialement marqué

La définition formelle des réseaux de Petri est introduite ci-dessous.

Définition 2.17 *Un* réseau de Petri *(ordinaire, initialement marqué) consiste en un* 4*-uplet* $N \stackrel{\text{def}}{=} (P, T, F, M_0)$*, où :*

- P *est un ensemble fini, non-nul ; les éléments de* P *sont appelés* places *;*
- T *est un ensemble fini tel que* $P \cap T = \varnothing$ *; les éléments de* T *sont appelés* transitions *;*
- F *est un sous-ensemble de* $(P \times T) \cup (T \times P)$ *; les éléments de* F *sont appelés* arcs *;*
- M_0 *est un multi-ensemble à valeurs dans* P *; une place* p *est* initiale ssi $M_0(p) > 0$.

Il convient de noter que de nombreux auteurs étendent cette définition des réseaux de Petri en définissant F comme un multi-ensemble (ce sont les réseaux dits *généralisés*) au lieu d'un ensemble (ce sont les réseaux dits *ordinaires*).

La définition suivante introduit quatre prédicats usuels sur les arcs des réseaux de Petri.

Définition 2.18 *Soit* $N = (P, T, F, M_0)$ *un réseau de Petri.*

- *les* places d'entrée *de la transition* t *sont* ${}^\bullet t \stackrel{\text{def}}{=} \{p \in P \mid (p, t) \in F\}$ *;*
- *les* places de sortie *de la transition* t *sont* $t^\bullet \stackrel{\text{def}}{=} \{p \in P \mid (t, p) \in F\}$ *;*
- *les* transitions d'entrée *de la place* p *sont* ${}^\bullet p \stackrel{\text{def}}{=} \{t \in T \mid (t, p) \in F\}$ *;*
- *les* transitions de sortie *de la place* p *sont* $p^\bullet \stackrel{\text{def}}{=} \{t \in T \mid (p, t) \in F\}$.

Il est à noter que ces quatre prédicats, communément usités dans les publications sur les réseaux de Petri, sont aussi appelés *pré-ensemble* de t, *post-ensemble* de t, *pré-ensemble* de p et *post-ensemble* de p.

2.3.2 Marquages accessibles et réseaux saufs

Cette section rappelle les définitions usuelles concernant la sémantique opérationnelle des réseaux de Petri.

Un *jeton* est localisé dans une place et se représente graphiquement par un point au centre de la place correspondante. Un *marquage* consiste en un ensemble de jetons. Il existe un *marquage initial* (voir M_0 dans la définition 2.17). Dans un marquage donné, une transition est dite *active* si toutes ses places d'entrée possèdent un jeton. Si tel est le cas, son *tir* consiste à retirer de chaque place d'entrée un jeton, puis à ajouter un jeton dans chaque place de sortie : cette opération produit un nouveau marquage de manière atomique (le retrait et l'ajout des jetons sont indivisibles). Un marquage est dit *accessible* s'il existe une séquence finie de tir de transitions permettant de l'obtenir depuis le marquage initial.

La définition suivante décrit formellement les règles liées au comportement dynamique des réseaux de Petri.

Définition 2.19 *Soit $N = (P, T, F, M_0)$ un réseau de Petri.*

- *un* marquage *M est un multi-ensemble à valeurs dans P, donnant le nombre de* jetons *localisés dans chaque place p ;*
- *une place p est* marquée *dans un marquage M* ssi *$p \in M$, c.-à-d.* ssi *$M(p) > 0$;*
- *une transition t est* active *dans un marquage M* ssi *$^\bullet t \subseteq M$, ce que l'on note $M \xrightarrow{t}$;*
- tirer *une transition t depuis un marquage M_1 est une opération définie* ssi *t est* active *dans M_1, et revient à produire un nouveau marquage $M_2 \stackrel{\text{def}}{=} M_1 - \mathbb{1}_{(^\bullet t)} + \mathbb{1}_{(t^\bullet)}$, c.-à-d. que chaque place d'entrée de t perd un jeton, puis chaque place de sortie de t gagne un jeton t ; on note alors, selon le contexte, $M_1 \xrightarrow{t} M_2$ ou $M_1 \longrightarrow M_2$;*
- *un marquage M_2 est* accessible *depuis un marquage M_1, ce que l'on note $M_1 \xrightarrow{*} M_2$,* ssi *M_2 est inclus dans le plus petit point fixe correspondant à la fonction suivante : $f : E \mapsto E \cup M_1 \cup \{M_2 \mid (M \in E) \wedge (M \longrightarrow M_2)\}$;*
- *un marquage M est* accessible ssi *$M_0 \xrightarrow{*} M$, ce que l'on abrège en $\xrightarrow{*} M$.*

Concrètement, une place peut s'assimiler à un compteur. Le nombre de jetons s'assimile ainsi à la valeur acquise de ce compteur au cours de l'exécution d'un programme. La notion de marquage (accessible ou non) correspond quant à elle à un état global (accessible ou non) du système modélisé. Le marquage initial n'est rien d'autre qu'une initialisation des variables du système. Enfin, une transition est un objet modifiant l'affectation de ces jetons, ce qui permet, partant d'un état global donné, d'aboutir à un nouvel état global.

Dans la suite de ce document, nous n'allons considérer (sauf mention contraire) que des réseaux de Petri dits « saufs », dont voici la définition.

Définition 2.20 *Soit $N = (P, T, F, M_0)$ un réseau de Petri.*

- *un marquage M est* sauf *si et seulement s'il contient au plus un jeton par place ;*
- *le réseau N est* sauf ssi *tout marquage accessible est sauf.*

Concrètement, un réseau sauf est un réseau dont ses places sont des variables booléennes plutôt qu'entières. Cela permet de considérer que les marquages sont des ensembles, plutôt que des multi-ensembles ou des fonctions à valeurs dans $\mathbb{N}$, ce qui permet de simplifier considérablement les notations. Par exemple, le marquage résultant du tir $M_1 \xrightarrow{t} M_2$ s'exprime désormais ainsi : $M_2 = (M_1 \setminus {}^\bullet t) \cup t^\bullet$.

La définition suivante décrit le problème de couverture d'un marquage, sur lequel nous nous appuierons ultérieurement.

Définition 2.21 *Soit $N = (P, T, F, M_0)$ un réseau de Petri. Un marquage M est dit* couvert *par un marquage accessible* ssi *le prédicat $\Re(M) \stackrel{\text{def}}{=} (\exists M' \mid (M \subseteq M' \subseteq P) \wedge (M_0 \xrightarrow{*} M'))$ est vrai, c.-à-d.,* ssi *M est un sous-ensemble de M' et que M' est un marquage accessible.*

Il est intéressant de noter que le problème de couverture d'un marquage est un sous-problème de celui de l'accessiblité d'un marquage. D'après [CEP95, L. 5] et [CEP95, Th. 15], ces deux problèmes sont PSPACE-complets.

Chapitre 3

Réseaux de Petri à unités imbriquées

Les réseaux de Petri à unités imbriquées, en abrégé, les NUPN (en anglais « Nested-Unit Petri Nets ») étendent les réseaux de Petri avec les notions de modularité (voir la section 1.5) et de hiérarchie (voir la section 1.6). Les NUPN ont été à l'origine développés afin de faciliter la traduction d'algèbres de processus en réseaux de Petri (voir la section 1.7).

Ce chapitre donne d'abord les définitions et propriétés du formalisme NUPN qui sont essentielles pour ce travail ; une présentation plus détaillée de ce modèle de calcul est proposée dans [Gar19]. Il effectue ensuite un résumé des publications à ce sujet, passant en revue divers formalismes proposés pour modéliser les systèmes concurrents hiérarchiques. Ce chapitre présente enfin les formats de fichiers et les outils disponibles pour les NUPN, et décrit la collection de modèles NUPN utilisée dans cette thèse.

3.1 Définition du modèle NUPN

La définition formelle ci-dessous des NUPN repose sur celle des réseaux de Petri ordinaires, initialement marqués et saufs.

Définition 3.1 *Un* réseau de Petri à unités imbriquées *(acronyme : NUPN) consiste en un* 8*-uplet* $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *où :*

- (P, T, F, M_0) *est un réseau de Petri ordinaire, initialement marqué et sauf, tel qu'introduit dans la section 2.3.*
- U *est un ensemble non-nul et disjoint de* $P \cup T$ *; ses éléments sont appelés* unités*.*
- u_0 *est un élément de* U*, appelé* unité racine*.*
- $\sqsubseteq$ *est un ordre partiel sur* U *tel que* $(\forall u_1, u_2 \in U)\ u_1 \sqsupseteq u_2 \stackrel{\text{def}}{=} u_2 \sqsubseteq u_1$ *et tel que* $(U, \sqsupseteq)$ *est un arbre ayant une unique racine* u_0 *:* u_0 *est la racine de* $(U, \sqsupseteq)$*, le plus petit élément de* U *pour* $\sqsupseteq$ *et le plus grand élément de* U *pour* $\sqsubseteq$*. Ainsi,* $u_1 \sqsubseteq u_2$ *exprime que soit l'unité* u_1 *est égale à l'unité* u_2*, soit* u_1 *est récursivement imbriquée dans l'unité* u_2*, c.-à-d.* $u_1 \sqsubset u_2$ *;*

— unité *est soit une bijection* $P \to U$*, soit une bijection* $P \to U \setminus \{u_0\}$*. Concrètement,* unité *associe, à chaque place de* P*, l'unité de* U *la contenant directement.*

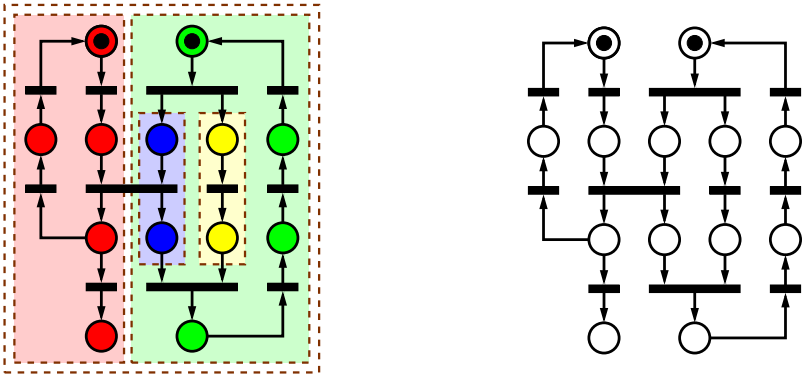

FIGURE 3.1 – Un NUPN (à gauche) et son réseau de Petri sous-jacent (à droite)

Tout NUPN possède un réseau de Petri *sous-jacent*, correspondant à la troncature du 8-uplet de la définition 3.1 à ses quatre premiers éléments. D'un point de vue pratique, un NUPN est une *extension* d'un réseau de Petri, avec des unités regroupant ses places, les unités étant récursivement imbriquées de sorte à former un arbre enraciné. La figure 3.1 montre, à gauche, un exemple de NUPN, et à droite, le réseau de Petri sous-jacent correspondant.

Toutes les notions, définitions et propriétés existantes des réseaux de Petri demeurent inchangées, y compris celles de la section 2.3. C'est pourquoi nous qualifions cette extension de *rétrocompatible*.

Il est intéressant de noter que, d'après le dernier élément de la définition 3.1, seule l'unité racine u_0 peut n'être associée à aucune place du réseau (c'est-à-dire, ne contenir directement aucune place). Ainsi, on a : $1 \leq |U| \leq |P|+1$. En conséquence, puisque P est fini, l'ensemble U est nécessairement fini et il n'existe qu'un nombre fini de NUPN ayant le même réseau de Petri sous-jacent.

La définition suivante présente les notions de base concernant la modularité des NUPN, à savoir l'attribution des places aux unités.

Définition 3.2 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN.*

— *L'ensemble des* places locales *d'une unité* u *est l'ensemble des places directement (et non transitivement) contenues dans* u *:* $\mathsf{places}(u) \stackrel{\text{def}}{=} \{p \in P \mid \mathsf{unité}(p) = u\}$ *;*

— *L'ensemble des* places globales *d'une unité* u *est constitué des places transitivement contenues dans* u *:* $\mathsf{places}^*(u) \stackrel{\text{def}}{=} \{p \in P \mid (\exists u' \in U)\ (u' \sqsubseteq u) \wedge (\mathsf{unité}(p) = u')\}$ *;*

— *Une unité* vide *est une unité n'ayant pas de place locale ; seule* u_0 *peut être vide.*

La définition suivante introduit les prédicats de base relatifs à la hiérarchie des NUPN, autrement dit, au sujet de l'imbrication des unités.

Définition 3.3 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN.*

— *Les* sous-unités *d'une unité u sont l'ensemble des unités directement imbriquées (et non transitivement) dans u et qui sont distinctes de u :*
$\mathsf{sous\text{-}unités}\,(u) \stackrel{\text{def}}{=} \{u' \in U \mid (u' \sqsubset u) \wedge (\nexists u'' \in U)\ (u' \sqsubset u'') \wedge (u'' \sqsubset u)\}$;

— *les* unités récursivement imbriquées dans une unité u *sont l'ensemble des unités transitivement imbriquées dans u, privé de u :* $\mathsf{sous\text{-}unités}^*(u) \stackrel{\text{def}}{=} \{u' \in U \mid u' \sqsubset u\}$;

— *une unité* feuille *est une unité minimale pour $(U, \sqsubseteq)$, c.-à-d. sans sous-unité.*

Les notions dérivées de la topologie des unités, qui forment un arbre enraciné, sont introduites par la définition ci-dessous.

Définition 3.4 *Soit $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un NUPN.*

— *la* largeur *d'une unité u est le nombre d'unités feuilles dans $\{u\} \cup \mathsf{sous\text{-}unités}^*(u)$;*

— *la* largeur *de N est la largeur de u_0, autrement dit, son nombre d'unités feuilles ;*

— *la* hauteur *d'une unité u est la longueur de la plus longue chaîne d'unités imbriquées $u' \sqsubset \ldots \sqsubset u$, plus un si u est non-vide ; ainsi, une est la hauteur de toute feuille u' ;*

— *la* hauteur *de N est la hauteur maximale de ses unités, c.-à-d. la* hauteur *de u_0 ;*

— *la* profondeur *d'une unité u est le nombre d'unités non-vides dans lesquelles u est récursivement imbriquée ; en particulier, l'unité racine a pour la profondeur zéro.*

Observons que $\forall u \in U, \mathsf{hauteur}\,(u) + \mathsf{profondeur}\,(u) = \mathsf{hauteur}\,(N)$.

La figure 3.2 illustre les valeurs acquises par les prédicats largeur, hauteur et profondeur sur la structure en unités du NUPN de la figure 3.1, qui a une largeur de 3 et une hauteur de 2.

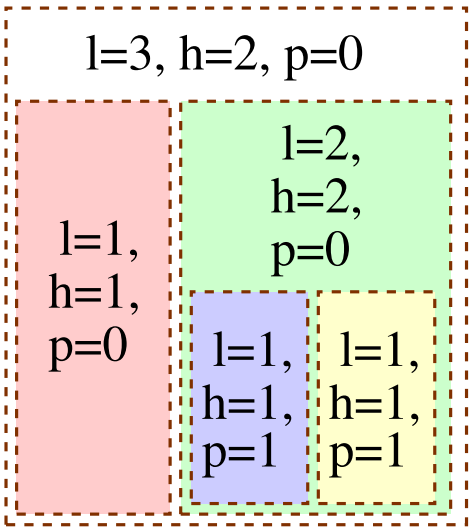

FIGURE 3.2 – Données topologiques de l'arbre d'imbrication des unités de la figure 3.1

3.2 Des réseaux de Petri aux NUPN triviaux

La définition suivante caractérise les NUPN *triviaux*, qui correspondent à des réseaux de Petri n'ayant ni modularité (c.-à-d. de processus concurrents), ni hiérarchie.

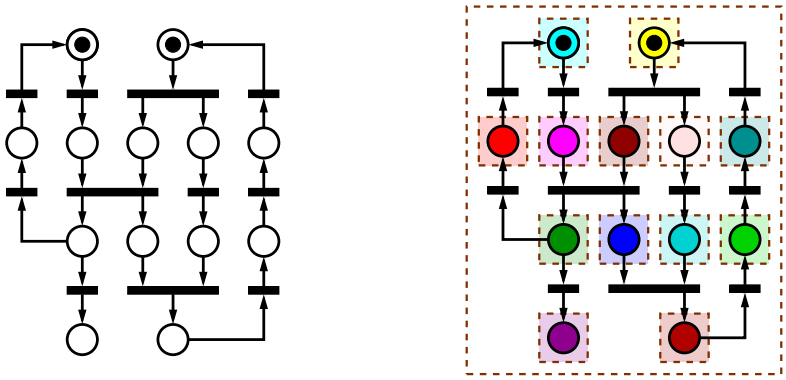

FIGURE 3.3 – Exemple de réseau de Petri (à gauche) et de son NUPN trivial (à droite)

Définition 3.5 *Un NUPN est* trivial *si sa largeur est égale à son nombre de places.*

La proposition suivante montre que tout réseau de Petri ordinaire et sauf peut se traduire en NUPN trivial et sauf.

Proposition 3.1 *Soit* (P, T, F, M_0) *un réseau de Petri ordinaire et sauf. Il existe un* 4*-uplet* $(U, u_0, \sqsubseteq, \mathsf{unité})$ *tel que* $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *soit un NUPN trivial et sauf.*

Preuve. Il suffit de choisir un 4-uplet $(U, u_0, \sqsubseteq, \mathsf{unité})$ tel que :

- $|U| = \mathsf{si}\ |P| > 1\ \mathsf{alors}\ |P| + 1\ \mathsf{sinon}\ 1$;
- $u_0 = \mathsf{max}(U)$;
- $\forall u_1, u_2 \in U, (u_1 \sqsubseteq u_2) \Leftrightarrow (u_2 \in \{u_0, u_1\})$;
- $\forall (p, u) \in P \times U, (\mathsf{unité}(p) = u) \Leftrightarrow (\#_P(p) = \#_U(u))$.

□

Concrètement, un NUPN trivial est un NUPN où chaque unité n'a qu'une seule place locale, sauf l'unité racine qui peut n'en contenir aucune. La figure 3.3 montre un exemple de telle traduction.

Dans le cas général d'un réseau de Petri ordinaire et sauf, ayant zéro place ou au moins deux places, il existe un seul NUPN trivial correspondant, au nommage des unités près. Cette forme correspond à celle produite par la preuve de la proposition 3.1.

Mais, dans le cas particulier où le réseau de Petri sous-jacent ne possède qu'une seule place, il existe une seconde forme de NUPN trivial, obtenue en remplaçant la première contrainte « $|U| = \mathsf{si}\ |P| > 1\ \mathsf{alors}\ |P| + 1\ \mathsf{sinon}\ 1$ » de la preuve susmentionnée par « $|U| = |P| + 1$ ». Une fois cette contrainte remplacée, le NUPN généré n'a non pas une seule unité, mais

deux unités, comme le montre la figure 3.4.

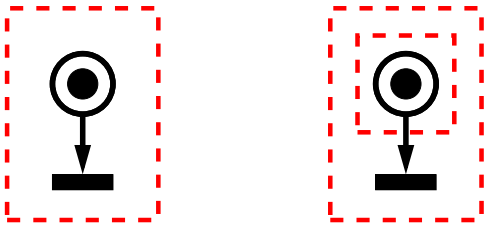

FIGURE 3.4 – Deux formes de NUPN triviaux pour les réseaux à une place

3.3 Des automates communicants aux NUPN plats

La définition suivante caractérise les NUPN *plats*, qui sont des NUPN possédant des informations de modularité (des processus concurrents), mais ne disposant pas de hiérarchie.

Définition 3.6 *Un NUPN est* plat *si sa hauteur est égale à un. Ceci revient à dire que toute unité non-feuille est vide (sachant que, pour rappel, seule l'unité racine u_0 peut être vide), ou encore que ses unités ont toutes une profondeur de zéro.*

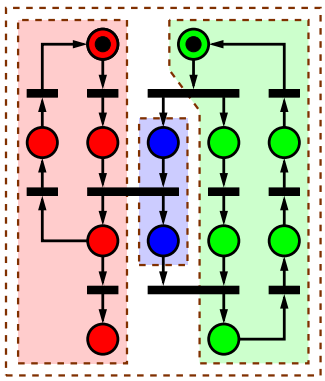

FIGURE 3.5 – Exemple de NUPN plat

Concrètement, un NUPN plat est un NUPN où toute unité non-vide est feuille et où toute unité ayant au moins une sous-unité est une unité vide. Ils correspondent, dans la terminologie NUPN, à un réseau décomposé en automates communicants. Un exemple de NUPN plat est représenté par la figure 3.5.

Puisqu'un NUPN trivial est un NUPN plat, il existe pour tout réseau de Petri ordinaire et sauf, au moins un NUPN plat et sauf correspondant.

La proposition suivante montre qu'il est possible de convertir tout NUPN non-plat et sauf en un NUPN plat et sauf, ayant le même nombre d'unités feuilles. Celui-ci est obtenu depuis le réseau originel en élaguant les unités non-feuilles et non-racines, puis en déplaçant toutes les places vers les unités feuilles.

Proposition 3.2 *Tout NUPN peut être transformé en un NUPN plat, préservant la propriété sauf et conservant la même largeur que le réseau d'origine.*

Preuve. Soit un NUPN $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. Nous définissons alors $N' \stackrel{\text{def}}{=} (P, T, F, M_0, U', u_0, \sqsubseteq', \mathsf{unité}')$, tel que :

— $U' \stackrel{\text{def}}{=} \{u \in U \mid (u = u_0) \vee \mathsf{feuille}(u)\}$;

— $\sqsubseteq' \stackrel{\text{def}}{=} \sqsubseteq \cap (U' \times U')$;

— $\forall p \in P, \mathsf{unité}'(p) \stackrel{\text{def}}{=} \mathsf{choix}(\{u \in U \mid (u \sqsubseteq \mathsf{unité}(p)) \wedge \mathsf{feuille}(u)\})$.

Ainsi, N' est un NUPN plat et possède $\mathsf{largeur}(N)$ unités feuilles. Puisque N et N' ont le même réseau de Petri sous-jacent (P, T, F, M_0), N est sauf *ssi* N' est sauf. □

Le NUPN plat de la figure 3.5 peut être obtenu par application de la proposition 3.2 sur le NUPN en partie gauche de la figure 3.1.

Notons que l'opération de la proposition 3.2 retire la hiérarchie présente dans les unités du NUPN originel en transférant les places des unités parentes dans les unités filles. En outre, ce transfert de places altère les processus du NUPN originel, en confondant les places d'un processus parent avec celles d'un processus enfant.

Remarquons enfin que le NUPN N' produit par la preuve de la proposition 3.2 est identique au NUPN d'entrée N *ssi* N est plat.

3.4 Des réseaux saufs aux NUPN unit-saufs

La définition suivante introduit la notion de réseau *unit-sauf*, généralisation de la notion de réseau sauf pour les NUPN. Elle est particulièrement cruciale, car c'est elle qui justifie l'intérêt de la structure en unités apportée par les NUPN.

Définition 3.7 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. *Alors :*

— *Le prédicat* $\mathsf{disjoint}(u_1, u_2) \stackrel{\text{def}}{=} (u_1 \not\sqsubseteq u_2) \wedge (u_2 \not\sqsubseteq u_1)$ *caractérise les paires d'unités qui ne sont pas en relation dans* $\sqsubseteq$*, c.-à-d., ne sont pas transitivement imbriquées ;*

— *Le marquage* $M \subseteq P$ *est dit* unit-sauf *si les jetons sont localisés dans des unités disjointes :* $(\forall p_1, p_2 \in M)\ (p_1 \neq p_2) \Rightarrow \mathsf{disjoint}(\mathsf{unité}(p_1), \mathsf{unité}(p_2))$ *;*

— *Le NUPN* N *est dit* unit-sauf *si tous ses marquages accessibles sont unit-sauf.*

Dans un NUPN unit-sauf, une unité possède au plus un jeton, pour tout marquage accessible. Ceci correspond à la notion de modularité des processus dans les systèmes concurrents hiérarchiques, où un processus donné ne peut être que dans un seul état à la fois (voir la section 1.5).

De plus, dans un NUPN unit-sauf, deux unités imbriquées, autrement dit en relation parent-enfant, ne peuvent avoir simultanément un jeton dans un marquage accessible. Ceci correspond à la notion de hiérarchie des processus dans les systèmes concurrents

hiérarchiques, où l'exécution d'un processus parent ne peut se dérouler en même temps que l'exécution de l'un de ses enfants (voir la section 1.6).

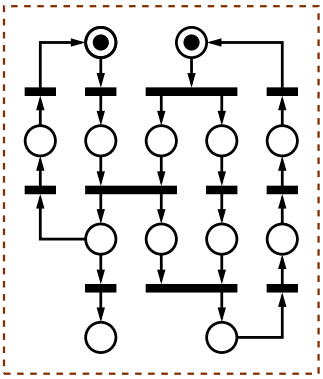

FIGURE 3.6 – Exemple de NUPN sauf, mais pas unit-sauf

Si N est unit-sauf, alors le réseau de Petri sous-jacent (P, T, F, M_0) est sauf — mais la réciproque est fausse. La figure 3.6 illustre un tel contre-exemple : l'unité racine contient deux jetons dans le marquage initial, mais son réseau de Petri sous-jacent est sauf.

L'utilisation de la propriété unit-sauf permet d'accroître l'efficacité des approches de vérification formelle en apportant : (i) Une réduction du nombre de variables requis pour coder chacun des marquages accessibles. Le codage le moins dense consiste à utiliser une variable booléenne par place (il s'agit du codage traditionnel pour les réseaux de Petri saufs, sans structure NUPN). Le codage le plus dense consisterait à explorer d'abord tous les marquages accessibles, puis à les coder en utilisant $\lceil \log_2 (|\{M \mid M_0 \xrightarrow{*} M\}|) \rceil$ variables ; ce codage est bien évidemment irréaliste, l'exploration ne pouvant pas être conduite sans avoir au préalable un codage. Entre ces deux extrêmes, [Gar19, sect. 6] répertorie cinq codages utilisables en pratique et évalue leurs densités respectives. (ii) Une meilleure localité, en rapprochant les variables associées à un même processus [ABD17]. Ces deux points permettent des gains souvent sensibles lors de l'exploration des marquages accessibles d'un réseau au moyen de diagrammes de décision binaires, à la fois en terme de rapidité d'exploration et en terme d'utilisation de la mémoire.

Il existe différentes façons de s'assurer qu'un NUPN soit unit-sauf (ou sauf). Lorsqu'un modèle NUPN présente une directive « `!unit_safe` » (voir la section 3.6), le NUPN est unit-sauf (et ainsi sauf) par construction (autrement dit, garanti unit-sauf par l'outil l'ayant généré). En l'absence d'une telle directive, il est possible d'utiliser les algorithmes de CONCNUPN (voir la section 6.14), qui acceptent une classe vaste de réseaux. Si cet outil ne peut prouver la propriété, nous devons vérifier cette propriété (il s'agit d'un problème PSPACE-complet, d'après la section 6.14) à l'aide d'un outil reposant sur le parcours des marquages accessibles (voir la section 3.7), par exemple CÆSAR.BDD (avec l'option « `-check` ») ou CÆSAR.SDD (avec l'option « `--check` »).

La proposition suivante montre, une fois combinée avec la proposition 3.1, que tout réseau

de Petri ordinaire et sauf peut se traduire en un NUPN trivial et unit-sauf. Ceci rend la notion de NUPN trivial importante sur le plan théorique, car elle implique que la classe des NUPN unit-saufs possède la même expressivité que celle des réseaux de Petri saufs.

Proposition 3.3 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN. Si* N *est trivial et sauf, alors il est également unit-sauf.*

Preuve. Puisque N est trivial, $\forall p_1, p_2 \in P, (p_1 \neq p_2) \Leftrightarrow \mathsf{disjoint}\,(\mathsf{unité}\,(p_1), \mathsf{unité}\,(p_2))$. Ainsi, tout marquage accessible sauf est aussi unit-sauf. D'où le résultat. □

La proposition suivante généralise la proposition 3.2 aux NUPN unit-saufs.

Proposition 3.4 *Tout NUPN unit-sauf peut être transformé en un NUPN plat, unit-sauf et conservant la même largeur que le réseau d'origine.*

Preuve. Soit un NUPN $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ unit-sauf. Soit le NUPN produit par la preuve de la proposition 3.2 : $N' = (P, T, F, M_0, U', u_0, \sqsubseteq', \mathsf{unité}')$. Par définition, nous avons : $\forall p_1, p_2 \in P, (\mathsf{unité}'(p_1) \sqsubseteq' \mathsf{unité}'(p_2)) \Rightarrow (\mathsf{unité}\,(p_1) \sqsubseteq \mathsf{unité}\,(p_2))$. Ainsi, tout marquage unit-sauf dans N' l'est également dans N. Donc le NUPN N' est plat, unit-sauf et $\mathsf{largeur}\,(N) = \mathsf{largeur}\,(N')$. □

En définitive, le formalisme NUPN est une extension stricte des réseaux de Petri ordinaires et saufs. Nous pouvons distinguer trois catégories utiles de structuration en unités :

— Les NUPN triviaux et (unit-)saufs, qui permettent aux NUPN de conserver l'expressivité des réseaux de Petri ordinaires et saufs (voir les propositions 3.1 et 3.3).

— Les NUPN plats, non-triviaux et unit-saufs, qui permettent de représenter la modularité des automates communicants sous la forme de réseaux de Petri.

— Les NUPN non-plats et unit-saufs, appelés NUPN *hiérarchiques*, qui permettent de représenter la modularité et la hiérarchie des unités.

3.5 Panorama des formalismes alternatifs aux NUPN

Nous avons vu que les réseaux de Petri permettent de modéliser fidèlement la concurrence asynchrone (voir la section 1.4), mais qu'ils manquent de structure pour exprimer la modularité et la hiérarchie (voir la section 1.7).

C'est cette lacune qui a conduit à la naissance du formalisme NUPN. Notons que nous nous concentrons sur l'étude des flots de contrôle des systèmes modélisés, et non sur leurs données. Une comparaison détaillée des NUPN avec d'autres extensions proposées pour les réseaux de Petri se retrouve dans [Gar19, Sect. 8]. Parmi celles-ci, trois se rapprochent du modèle NUPN :

— Les extensions apportant aux réseaux de Petri les notions de *composition parallèle* [LAV90] [PBV11] [AD22] ou de *boîtes* [BDH92, BDE93, KEB94], [BDK99, BDK02], [BDK01a, BDK01b] [DKKP02, DKKP03]. Elles définissent des opérateurs prenant

en entrée plusieurs sous-réseaux et produisant un nouveau réseau dont les transitions communes sont synchronisées. Ces réseaux peuvent certes être composés récursivement, mais cela ne conduit pas à l'organisation hiérarchique décrite en section 1.6 : ces extensions ne permettent pas d'exprimer le fait qu'un réseau soit le parent d'un autre.

— Les réseaux de Petri proposés dans [AKW05, Mic05] sont étendus par des *macro-places*, qui autorisent la construction d'un réseau par assemblage (récursif) de sous-réseaux. Une distinction est réalisée entre les macro-places *parallèles* et les macro-places *séquentielles*, qui permet de déclarer si les places récursivement contenues dans une macro-place donnée peuvent ou non posséder simultanément un jeton dans un même marquage accessible. L'utilisation conjointe de ces deux types de macro-places permettrait de décrire à la fois la modularité et la hiérarchie des systèmes concurrents hiérarchiques, en générant deux macro-places par processus s'il contient des sous-processus, et sinon, dans le cas d'un processus feuille, une seule macro-place. Néanmoins, cette distinction entre les macro-places parallèles et séquentielles n'est qu'évoquée informellement.

— Les réseaux de Petri *colorés* [Jen92] permettent la définition de *couleurs*, autrement dit de types de données. À chaque place du réseau est attachée une couleur. Chaque arc possède un fragment d'expression dont le type correspond à celui de la place reliée. Tout jeton possède une valeur, dont le type est celui de la place le contenant. L'attribution d'une couleur par processus permettrait de séparer les places et les jetons appartenant à des processus distincts et de ne pas confondre les places des processus réalisant des transitions synchronisées ; ceci exprimerait la notion de modularité. En définissant un codage des couleurs approprié, il serait même possible de décrire une forme de hiérarchie. Cependant, les réseaux colorés ne permettent pas de déclarer qu'il ne peut exister simultanément deux jetons dans les places d'un même processus et, a fortiori, dans les places de deux processus imbriqués hiérarchiquement. Autrement dit, ils ne possèdent pas de propriété équivalente à unit-sauf. En outre, cette construction décrit la modularité et la hiérarchie dynamiquement, en attachant des données aux jetons. Les NUPN fournissent une structure statique, qui permet de distinguer plus simplement les composants séquentiels d'un système et leur imbrication.

La suite de cette section est consacrée à d'autres formalismes qui ne reposent pas sur les réseaux de Petri. Les deux formalismes suivants permettent de décrire la modularité de processus concurrents, mais ne permettent pas de décrire leur hiérarchie :

— Les automates communicants permettent l'expression de la composition en parallèle d'automates finis, c.-à-d. de composants séquentiels. Il existe de nombreuses variantes d'automates communicants, certaines axées sur les états, d'autres sur les actions, certaines autorisant le lancement ou la terminaison d'automates, tandis que d'autres imposent la création d'états de « repos », certaines autorisant la création d'états partagés entre différents automates, tandis que d'autres l'interdisent, etc. Cependant,

il est difficile d'exprimer avec des automates communicants une succession séquentielle d'actions, dont certaines sont des étapes parallèles, par exemple $a.(b \parallel c).d$. En effet, ceci impose, soit de développer le terme parallèle $b \parallel c$ dans l'automate parent, ce qui conduirait à une explosion combinatoire de la taille des automates produits, soit de recourir à la génération de plusieurs automates à plat, puis à les synchroniser, ce qui n'est guère élégant.

— Le *langage de spécification et de description* SDL (en anglais « Specification and Description Language ») [CCI88], est un langage formel et communément utilisé pour la modélisation des systèmes distribués, asynchrones et éventuellement temps réel. Ce langage normalisé possède à la fois une description textuelle précise et une représentation graphique intuitive. Concrètement, un système modélisé en SDL est constitué de blocs formant un arbre enraciné. Ces blocs peuvent communiquer par passage de messages. Mais, parmi les sommets de cet arbre enraciné, seules les feuilles peuvent contenir des tâches séquentielles (sous une forme d'automates finis étendus). Le formalisme de SDL, bien que hiérarchique, n'est donc pas plus expressif que celui des automates communicants.

Les quatre autres formalismes normalisés Estelle, UML, BPMN et SysML, permettent de décrire à la fois la modularité et la hiérarchie des processus. Mais ces formalismes donnent aux processus imbriqués hiérarchiquement une autre sémantique d'exécution que celle que nous avons décrite à la section 1.6 :

— Le langage *Estelle* [ISO88] est un langage pour la description formelle de systèmes concurrents. Il repose, comme SDL, sur une forme d'automates finis étendus pouvant communiquer par passage de messages. Un système est modélisé en Estelle par un arbre enraciné dont les sommets sont appelés modules. Mais ici, contrairement à SDL, tous les modules de l'arbre contiennent un automate fini, y compris les nœuds non-feuilles. En outre, en Estelle, chaque module possède un type définissant la sémantique d'exécution parallèle de ses modules fils, qui peut être soit synchrone, soit asynchrone. Bien qu'Estelle puisse combiner de manière synchrone des sous-modules asynchrones, ce langage ne permet pas de combiner de manière asynchrone des sous-modules synchrones. Le langage Estelle impose une autre contrainte : au sein d'un module donné, une transition ne peut être exécutée que si aucune des transitions contenues dans les modules parents (par transitivité) n'est exécutable.

— Le *langage de modélisation unifié* UML (en anglais « Unified Modeling Language ») [ISO12] est un langage de modélisation graphique inspiré du paradigme de la programmation par objet. Un très haut niveau d'abstraction est apporté par UML, permettant ainsi de représenter de nombreux systèmes sans devoir se préoccuper des détails d'implantation. Cette norme est ainsi largement utilisée pour documenter des processus existants, planifier et analyser le comportement d'un système avant sa mise en œuvre. La sémantique d'UML est cependant imprécise, dépendante des outils de modélisation, ce qui rend les modèles UML difficilement portables. Parmi les quatorze diagrammes proposés par la norme UML, deux nous intéressent : il

s'agit des diagrammes d'états et diagrammes d'activité. Les diagrammes d'états étendent le concept des automates finis en introduisant la notion d'automates finis hiérarchiquement imbriqués ; c'est pourquoi ces diagrammes sont également appelés « automates finis hiérarchiques ». Un tel automate peut soit s'exécuter lui-même, soit exécuter en parallèle ses sous-automates. Cette modélisation de la concurrence est cependant restrictive, dans la mesure où les automates parallèles sont initialisés et arrêtés en même temps. Les diagrammes d'activité permettent, quant à eux, de modéliser des processus métiers, des processus d'utilisation, des flux de données et des flux de contrôle. Ces diagrammes présentent les mêmes limitations relatives à la concurrence que les diagrammes d'états.

— Cette analyse d'UML vaut également pour le *modèle de procédé d'affaire et notation* (en anglais, BPMN pour « Business Process Model and Notation ») [ISO13] et pour le *langage de modélisation des systèmes* (en anglais SysML, pour « Systems Modeling Language ») [ISO17], tous deux inspirés d'UML.

— Enfin, le formalisme de l'outil *Ptolemy II* [EJL+03] devrait permettre de décrire à la fois la modularité et la hiérarchie des systèmes concurrents hiérarchiques. La hiérarchie est exploitée par Ptolemy II afin de pouvoir regrouper des sous-composants exprimés dans des formalismes hétérogènes et donc des sémantiques d'exécution différentes. Concrètement, le modèle est un arbre enraciné de sous-modèles imbriqués, où chaque nœud possède un domaine, lequel définissant sa sémantique d'exécution. Les nœuds s'exécutent de manière concurrente et peuvent recourir à la communication par messages. Le domaine « machine à états finis » est introduit par [LL98], qui propose d'associer des nœuds dans ce domaine avec des nœuds ayant pour domaine « flux de données synchrone » ou « événements discrets ». Mais ces deux domaines définissent un parallélisme synchrone. Le domaine « rendez-vous », aussi proposé par Ptolemy II, correspondrait davantage à nos besoins : il propose une sémantique asynchrone, qui, combinée avec le domaine « machine à états finis », devrait permettre de décrire la libre composition des comportements séquentiels et parallèles des systèmes concurrents hiérarchiques. Cependant, la faisabilité d'une telle association ne semble pas étudiée.

3.6 Formats de fichiers NUPN et PNML

Il existe deux formats de fichiers principaux pour décrire les NUPN :

— Le format NUPN [Gar19, annexe A][1] est une représentation textuelle, concise et lisible pour un lecteur humain, possédant une syntaxe aisément implantable dans des programmes ou des scripts. Ce format possède une directive « `!unit_safe` » certifiant que le NUPN est unit-sauf, une directive « `!multiple_arcs` » indiquant que le réseau a été obtenu depuis un réseau de Petri non-ordinaire et une directive

1. cadp.inria.fr/man/nupn.html

« `!multiple_initial_tokens` » indiquant que le réseau a été obtenu depuis un modèle non-sauf, le marquage initial contenant plusieurs jetons. Notons que dans ce format, les ensembles d'unités, de places et de transitions sont des intervalles d'entiers et sont donc ordonnés. Il existe un mécanisme d'étiquettes permettant optionnellement d'attacher des chaînes de caractères aux unités, places et transitions.

— Le format PNML (acronyme anglais de « *Petri Net Markup Language* ») [ISO11] est une norme ISO reposant sur XML, permettant de décrire les réseaux de Petri et diverses extensions de ce formalisme ; la majeure partie des outils pour les réseaux de Petri a adopté ce format ; une section « `toolspecific` » [Gar19, annexe B] [2], fondée sur le mécanisme d'extension générique fourni par la norme PNML, permet de décrire les unités et leur imbrication hiérarchique, ainsi qu'une valeur booléenne certifiant, lorsqu'elle est vraie, que le NUPN décrit est unit-sauf.

Suivant leur format, les fichiers doivent se terminer par une extension « `.nupn` » ou « `.pnml` ».

La conversion entre les deux formats est facile : les fichiers NUPN peuvent être traduits en fichiers PNML en invoquant l'outil CÆSAR.BDD [3] (voir la section 3.7) avec l'option « `-pnml` », qui préserve dans la section « `toolspecific` » les informations relatives aux unités, ainsi que la valeur de l'attribut « `!unit_safe` ».

Les fichiers PNML peuvent quant à eux être convertis en fichiers NUPN par l'intermédiaire de l'outil Ndrio, disponible dans la boîte à outils Tina, développée au LAAS-CNRS (Toulouse, France) [4], ou bien de l'outil PNML2NUPN, développé au LIP6 (Paris, France) [5]. Cette conversion écarte les entités PNML qui ne sont pas pertinentes pour produire des NUPN, par exemple les attributs colorés, temporisés ou graphiques. Notons que, si le PNML source n'est pas ordinaire (il dispose d'arcs multivalués) ou s'il est non-sauf de manière évidente (par exemple, si certaines places possèdent plusieurs jetons dans le marquage initial), l'outil PNML2NUPN génère toujours un fichier NUPN, mais annoté avec des directives spéciales « `!multiple_arcs` » ou « `!multiple_initial_tokens` ».

3.7 Outils logiciels existants pour les NUPN

La structure NUPN est facilement exploitable dans les outils logiciels existants, d'où le rapide gain de popularité du formalisme NUPN dans la communauté des réseaux de Petri. À ce jour, au moins dix-neuf outils logiciels produisent des modèles NUPN (voir la table 3.1) ou exploitent l'information (voir la table 3.2) contenue dans de tels modèles.

2. mcc.lip6.fr/nupn.php
3. cadp.inria.fr/man/caesar.bdd.html
4. projects.laas.fr/tina/manuals/ndrio.html, invoqué avec l'option « `-nupn` ».
5. pnml.lip6.fr/pnml2nupn
6. cadp.inria.fr/man/caesar.html
7. cadp.inria.fr/man/nupn_info.html
8. cadp.inria.fr/man/exp.open.html

nom	localisation	référence
CÆSAR.BDD	Grenoble, France	[GLMS13] [3]
NDRIO	Toulouse, France	[BRV04] [4]
PNML2NUPN	Paris, France	[5]
CÆSAR	Grenoble, France	[GLMS13] [6]
NUPN_INFO	Grenoble, France	[GLMS13] [7]
EXP.OPEN	Grenoble, France	[GLMS13] [8]
RERS BENCHMARK GENERATOR	Dortmund, Allemagne et Twente, Pays-Bas	[SJMvdP17]

TABLE 3.1 – Liste d'outils de conversion de formats ou produisant de l'information NUPN

nom	localisation	référence
CÆSAR.BDD	Grenoble, France	[GLMS13] [3]
CÆSAR.SDD	Paris, France	[9]
CONCNUPN	Grenoble, France	section 3.7
KONG	Toulouse, France	[ABG23] [10]
ENPAC	Shanghai, Chine	[11]
GREATSPN-MEDDLY	Turin, Italie	[ABB+16] [12]
ITS-TOOLS	Paris, France	[TM15] [13]
LOLA	Rostock, Allemagne	[Wol18] [14]
LTSMIN	Twente, Pays-Bas	[KLM+15] [15]
PNMC	Paris, France	[Ham16] [16]
SIFT	Toulouse, France	[BRV04] [17]
SMPT	Toulouse, France	[ABD21] [18]
TEDD	Toulouse, France	[BRV04] [19]

TABLE 3.2 – Liste d'outils logiciels dédiés aux NUPN ou exploitant de l'information NUPN

Voici les principaux outils logiciels utilisés dans ce travail (en plus de ceux de la section 3.6) :

— ConcNUPN est un outil prototype écrit en langage Python (environ 730 lignes de code), implantant des algorithmes pour calculer les places mortes (voir la définition 5.2), les transitions mortes (voir la définition 5.4), les places concurrentes (voir la définition 6.1), les unités concurrentes (voir la définition 6.3), les propriétés sauf et unit-sauf (voir la section 6.14), ainsi que d'autres fonctionnalités. Il est voué à l'expérimentation de nouvelles idées dans cette thèse, ainsi qu'à la vérification croisée des résultats de CÆSAR.BDD.

— Kong est un outil développé par Nicolas Amat (LAAS-CNRS, Toulouse, France), visant à faciliter la détermination de la relation des places concurrentes, de l'ensemble des places mortes ou encore de l'accessibilité d'un marquage pour des réseaux de Petri de grande taille, par l'emploi de réductions structurelles. Cet outil sera abordé avec plus de détails à la section 6.13.

— Nupn_Info est un outil écrit en langages Awk et shell (environ 2 400 lignes de code) et disponible en tant que composant de la boîte à outils CADP [GLMS13] [7]. Cet outil implante à ce jour seize opérations de transformation des modèles NUPN, telles que le remplacement de la structure en unités par une structure triviale, les permutations des unités, places ou transitions, etc.

— CÆSAR.BDD est un outil de vérification pour les réseaux de Petri et les NUPN, écrit en langage C (environ 10 400 lignes de code) également disponible au sein de la boîte à outils CADP. Les marquages accessibles sont représentés sous forme de diagrammes de décision binaires, en exploitant la bibliothèque CUDD, ce qui est plus efficace à la fois en temps et en mémoire qu'une représentation explicite de ces marquages.

  Lors de son introduction en 2004, CÆSAR.BDD était un outil auxiliaire pour les réseaux de Petri interprétés générés automatiquement par le compilateur LOTOS [GS90] de CADP. Pour ces réseaux, qui peuvent être de grande taille, CÆSAR.BDD permet la suppression des transitions mortes (voir la définition 5.4) ainsi que la détermination des unités concurrentes (voir la définition 6.3), une notion nécessaire pour effectuer des analyses de flux de données [GS06].

  Depuis 2013, CÆSAR.BDD a été progressivement étendu avec de nouvelles fonc-

9. github.com/ahamez/caesar.sdd
10. github.com/nicolasAmat/Kong
11. github.com/Tj-Cong/EnPAC_2021
12. github.com/greatspn/SOURCES
13. lip6.github.io/ITSTools-web
14. service-technology.org/tools
15. ltsmin.utwente.nl
16. github.com/ahamez/pnmc
17. projects.laas.fr/tina/manuals/sift.html
18. homepages.laas.fr/namat/project/smpt
19. projects.laas.fr/tina/manuals/tedd.html

tionnalités, dont la conversion automatisée des réseaux du format NUPN vers le format normalisé PNML (mentionnée à la section 3.6), ainsi que la détermination d'une vingtaine de propriétés structurelles et comportementales des réseaux de Petri, dont figure la réversibilité, la vivacité, la présence de transitions mortes (voir la définition 5.4), la présence de places mortes (voir la définition 5.2), etc.

CÆSAR.BDD a été ensuite révisé pour enrichir le format de fichier NUPN avec des directives, des étiquettes de places, des étiquettes de transitions, des étiquettes d'unités et des contraintes syntaxiques et sémantiques plus strictes. Beaucoup de nouvelles options ont été ajoutées pour faciliter l'interrogation des modèles NUPN : nombre de places, nombre de transitions, densité des arcs, hauteur de l'arbre des unités, etc.

Les travaux de cette thèse ont donné une impulsion nouvelle au développement de CÆSAR.BDD. Des structures de données creuses sont maintenant employées pour enregistrer les arcs des réseaux, en remplacement de matrices denses, permettant ainsi de traiter plus efficacement des réseaux de grande taille. Le parcours des marquages accessibles a été amélioré par une meilleure gestion des erreurs et des dépassements de délais de CUDD, par une révision des critères activant le réordonnancement dynamique des variables et par un changement de stratégie de réordonnancement des variables du diagramme, qui permute maintenant les variables de la même unité de manière groupée et non plus individuellement. La détermination des places et transitions mortes (voir le chapitre 5), ainsi que le calcul des places concurrentes (voir le chapitre 6) ont été étendus avec de nouveaux algorithmes. Enfin, ont été ajoutées cinq nouvelles options pour résoudre le problème de l'isomorphisme de NUPN (voir le chapitre 4) et deux nouvelles options pour faciliter le problème de décomposition (voir les chapitres 7 et 8).

CÆSAR.BDD comprend à ce jour plus d'une cinquantaine d'options [3].

3.8 Constitution d'un jeu de tests

Le modèle NUPN a été adopté à ce jour par deux compétitions :

— Le MCC (acronyme anglais de « *Model Checking Contest* ») [20] [ABC+19, KHHH+21] est une compétition internationale consacrée à l'évaluation des outils de vérification formelle. À cette fin, cette compétition repose sur les réseaux de Petri en guise de formalisme commun, compris par tous les outils participants. Elle possède une collection de réseaux de Petri s'enrichissant chaque année, par la collecte de divers réseaux (au format normalisé PNML, optionnellement avec une structuration NUPN, voir la section 3.6) exigeants auprès de la communauté de la vérification de modèles. La collection du MCC contient actuellement 128 modèles paramétrés, totalisant 1 628 instances de réseaux, dont 1 387 ne sont pas colorés.

20. mcc.lip6.fr/models.php.

— Le RERS (acronyme anglais de « *Rigorous Examination of Reactive Systems* »)[21] [JMM+19, HJM+21] propose à ses participants de résoudre des problèmes de difficultés croissantes et synthétisés de sorte à exhiber des comportements complexes. Pour décrire les modèles correspondants aux problèmes parallèles de cette compétition, des fichiers au format PNML sont mis à disposition des participants. Ces fichiers PNML contiennent systématiquement une structuration NUPN. La collection du RERS contient actuellement 33 modèles.

Pour réaliser les expériences du présent document, nous avons utilisé une vaste collection comptant 16 200 réseaux de Petri ordinaires et saufs (éventuellement avec une structure NUPN unit-sauf), qui a été patiemment constituée depuis 2013 au centre INRIA de l'Université Grenoble Alpes.

La majeure partie des modèles de notre collection sont dérivés de spécifications « réalistes » (c.-à-d., qu'ils résultent de problèmes industriels décrits par des utilisateurs, dans des langages de spécifications de plus haut-niveau, tels que LOTOS, LNT, AADL, etc., plutôt que de réseaux de Petri générés arbitrairement ou aléatoirement) et sont unit-saufs (ce qui implique saufs) par construction.

Notre collection contient tous les modèles non-colorés et saufs[22] accumulés entre 2011 et 2021 par les éditions successives du MCC, tous les modèles NUPN du RERS, ainsi que tous les réseaux ordinaires et saufs de l'ancienne collection PetriWeb [GvHPvdW06], qui comptait 43 modèles.

À notre connaissance, notre collection est la plus importante jamais rapportée dans les publications relatives aux réseaux de Petri : à titre d'exemple, les résultats expérimentaux de [PC98, PCP99] reposaient sur au plus 15 modèles ; [MOW09] mentionne 1 700 réseaux de flux de travail fournis par IBM Zurich, [EM15] rapporte des expériences sur 1 976 réseaux de Petri et [WWJ19] évoque 202 réseaux de Petri saufs.

propriété	vraie	fausse	propriété	vraie	fausse
pure	66,8 %	33,2 %	connexe	95,6 %	4,4 %
choix libres	39,0 %	61,0 %	fortement connexe	9,4 %	90,6 %
choix libres étendus	40,1 %	59,9 %	conservatif	19,2 %	80,8 %
graphes marqués	1,9 %	98,1 %	sous-conservatif	29,3 %	70,7 %
machine à états	16,3 %	83,7 %	non trivial et unit-sauf	81,3 %	18,7 %

TABLE 3.3 – Propriétés structurelles et comportementales de nos modèles

Une étude statistique confirme la diversité de notre collection de modèles. La table 3.3 donne le pourcentage de modèles satisfaisant diverses propriétés (structurelles et comportementales) des réseaux (voir [Mur89] pour le vocabulaire associé). Le pourcentage de NUPN ayant une

21. www.rers-challenge.org.

22. Éventuellement suite à une transformation des arcs multivalués en arcs ordinaires, de sorte à obtenir des réseaux ordinaires.

structure non-triviale et possédant une directive « `!unit_safe` » (indiquant que le modèle est unit-sauf par construction) figure dans la table 3.3.

caractéristique	valeur min	valeur max	moyenne	médiane	déviation std
# places	1	131 216	345,8	24	2 799
# transitions	0	16 967 720	7 998,1	31	244 569
# arcs	0	146 528 584	71 217,9	88	2 080 604
densité d'arcs	0,0 %	100,0 %	10,2 %	5,9 %	0,1
# unités	1	78 644	123,4	7	1 536
hauteur	1	2 891	4,3	2	38,5
largeur	1	78 643	117,6	5	1 533

TABLE 3.4 – Propriétés numériques de nos modèles

La table 3.4 donne des informations sur la taille des modèles : nombre de places, de transitions et d'arcs, ainsi que la *densité d'arcs*, que nous définissons comme le nombre d'arcs divisé par le double du produit du nombre de places et du nombre de transitions (c'est-à-dire la quantité de mémoire requise pour enregistrer l'ensemble des arcs F sous la forme de deux matrices $P \times T$ et $T \times P$).

Deuxième partie

Équivalence

Chapitre 4

Équivalence de NUPN

Le problème de l'équivalence de NUPN caractérise les ensembles de réseaux identiques moyennant une permutation des places, une permutation des transitions et une permutation des unités. Il s'agit ainsi un d'outil pertinent pour la construction, l'évaluation et la maintenance de collections destinées aux évaluations comparatives et aux compétitions de logiciels pour les réseaux de Petri. Mais ce problème est difficile, car de complexité théorique équivalente à celle de l'isomorphisme de graphes.

Le présent chapitre propose diverses approches, intégrées dans une chaîne d'outils cohérente, destinées à résoudre ce problème de l'équivalence de NUPN. Cette chaîne d'outils a été évaluée sur quatre collections allant de 244 à 241 657 réseaux et contenant soit peu, soit beaucoup de doublons. Nous avons observé des taux de réussite allant de 99 % à 100 % pour la détection de doublons.

4.1 Motivations

Nous voyons se constituer de vastes collections de réseaux de Petri formées pour des tests de non-régression ou des compétitions de logiciels. La constitution, l'évaluation et la maintenance sur le long terme de telles collections rassemblant plusieurs centaines à plusieurs milliers de réseaux, exigent pour être réalisées adéquatement, un travail considérable et méticuleux à bien des égards.

Parmi les problèmes les plus courants figure la présence de modèles « doublons », c.-à-d. de multiples occurrences de réseaux identiques ou quasi-identiques. Ces modèles peuvent être présents pour trois raisons :

— La collection est constituée de modèles proposés par différents contributeurs.

— La collection est gérée par plusieurs personnes, qui pourraient indépendamment ajouter le même modèle.

— Les doublons peuvent provenir de transformations appliquées à des modèles existants,

par exemple, la production de réseaux de Petri depuis des spécifications de haut-niveau, la conversion de réseaux de Petri colorés en réseaux non-colorés, la suppression de places ou de transitions mortes (voir le chapitre 5), et ainsi de suite.

En pratique, ces « doublons » sont indésirables pour au moins quatre raisons :

— Ils entraînent un gaspillage d'espace disque et de ressources de sauvegarde, en particulier lorsque les réseaux sont exprimés dans des formats particulièrement verbeux, à l'instar du format normalisé PNML (voir la section 3.6) qui hérite des atouts et des faiblesses du langage XML sur lequel il est fondé.

— Ils conduisent à des calculs redondants, pouvant s'avérer particulièrement coûteux en temps processeur et en espace mémoire, notamment lorsque le recours à l'exploration des marquages accessibles est requis, en raison du problème de l'explosion de l'espace d'états.

— Ils peuvent introduire des biais insidieux dans les expériences d'évaluation comparative et les compétitions de logiciels, en augmentant indûment l'importance de certains réseaux.

— Leur présence soulève maintes fois des questions et des débats fastidieux (mais légitimes) entre les utilisateurs et les administrateurs de collections de réseaux.

Le présent chapitre aborde ce problème en proposant des approches, implantées dans des outils, pour détecter les doublons susceptibles d'être présents dans les collections existantes, ou introduits lors de l'élargissement de collections en cours de constitution. Ces approches et outils permettent également d'extraire, à partir d'une large collection possédant éventuellement de nombreux doublons, un sous-ensemble de modèles « uniques ».

La suite de ce chapitre s'articule comme suit. La section 4.2 introduit formellement ce problème. La section 4.3 esquisse un inventaire de l'état de l'art et la section 4.4 analyse sa complexité théorique.

Les quatre sections suivantes élaborent quatre approches complémentaires pour la détection des réseaux isomorphes ou non-isomorphes, ordonnées par complexité croissante : la section 4.5 présente la notion de signatures de réseaux ; la section 4.6 aborde le principe de canonisation de réseaux ; la section 4.7 expose une technique permettant de réduire le problème en un problème d'isomorphisme de graphes ; la section 4.8 explique comment le problème peut s'exprimer sous la forme de formules logiques. La section 4.9 présente l'intégration de toutes ces approches dans une chaîne d'outils.

La section 4.10 définit les métriques utiles pour un utilisateur final et expose comment une implantation efficace peut les utiliser. La section 4.11 donne les résultats expérimentaux obtenus sur quatre collections de réseaux. La section 4.12 aborde les démarches entreprises pour la validation des résultats.

Ce chapitre est clôs par la section 4.13, qui formule quelques remarques conclusives et prospectives.

4.2 Énoncé du problème

Cette notion de recherche de modèles « doublons » correspond au problème de recherche de NUPN isomorphes, que nous allons décrire formellement dans cette section.

4.2.1 Isomorphisme de réseaux de Petri

Dans un premier temps, nous définissons le concept d'*isomorphisme de réseaux de Petri*, autrement dit, les conditions pour que deux réseaux de Petri soient considérés isomorphes.

Intuitivement, nous considérons que deux réseaux de Petri sont des doublons *ssi* il existe une bijection entre leurs places et une bijection entre leurs transitions ; cette correspondance doit préserver les arcs et les marquages initiaux. Une telle définition étend la notion classique d'isomorphisme de graphes aux réseaux de Petri, en gardant à l'esprit que les réseaux de Petri sont des graphes bipartis orientés.

Définition 4.1 *Soit un réseau de Petri* $N = (P, T, F, M_0)$ *et soit* $\pi_P : P \to P'$ *une bijection. On note* $\overline{\pi_P}(N) \stackrel{\text{def}}{=} (P', T, F', M'_0)$ *le réseau de Petri obtenu en renommant les places de* N *suivant* π_P*, autrement dit, avec* $F' \subseteq (P' \times T) \cup (T \times P')$ *et* $M'_0 \subseteq P'$ *tels que :*

- $\forall (p,t) \in P \times T, (\pi_P(p), t) \in F' \Leftrightarrow (p,t) \in F$;
- $\forall (t,p) \in T \times P, (t, \pi_P(p)) \in F' \Leftrightarrow (t,p) \in F$;
- $\forall p \in P, \pi_P(p) \in M'_0 \Leftrightarrow p \in M_0$.

Notons que, puisque π_P est bijective, $|P| = |P'|$, $|F| = |F'|$ et $|M_0| = |M'_0|$.

Définition 4.2 *Soit un réseau de Petri* $N = (P, T, F, M_0)$ *et soit* $\pi_T : T \to T'$ *une bijection. On note* $\overline{\pi_T}(N) \stackrel{\text{def}}{=} (P, T', F', M_0)$ *le réseau de Petri obtenu en renommant les transitions de* N *suivant* π_T*, autrement dit, avec* $F' \subseteq (P \times T') \cup (T' \times P)$ *tel que :*

- $\forall (p,t) \in P \times T, (p, \pi_T(t)) \in F' \Leftrightarrow (p,t) \in F$;
- $\forall (t,p) \in T \times P, (\pi_T(t), p) \in F' \Leftrightarrow (t,p) \in F$.

Notons que, puisque π_T est bijective, $|T| = |T'|$ et $|F| = |F'|$.

Remarquons que pour toutes bijections π_P et π_T, les fonctions $\overline{\pi_P}$ et $\overline{\pi_T}$ sont commutatives, autrement dit, on a : $(\overline{\pi_T} \circ \overline{\pi_P}) = (\overline{\pi_P} \circ \overline{\pi_T})$.

Définition 4.3 *Soient deux réseaux de Petri* N *et* N' *tels que* $N = (P, T, F, M_0)$ *et* $N' = (P', T', F', M'_0)$*. Ces réseaux sont dits* isomorphes ssi *il existe deux bijections* $\pi_P : P \to P'$ *et* $\pi_T : T \to T'$ *telles que :* $N' = (\overline{\pi_T} \circ \overline{\pi_P})(N)$.

Intuitivement, pour la définition 4.3, deux réseaux de Petri sont isomorphes s'ils sont identiques à la permutation de leurs noms de places et leurs noms de transitions près.

4.2.2 Isomorphisme de NUPN

Nous étendons ensuite cette définition aux paires de NUPN, afin d'établir le concept d'*isomorphisme de NUPN*.

Intuitivement, nous définissons cette notion de modèles doublons ainsi : deux NUPN sont des doublons *ssi* il existe une bijection entre leurs places, une bijection entre leurs transitions et une bijection entre leurs unités ; cette correspondance doit préserver les arcs, les marquages initiaux, les unités racines, l'imbrication entre les unités (c.-à-d. la hiérarchie des NUPN) et la répartition des places dans les unités (c.-à-d. la modularité des NUPN).

Définition 4.4 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *et soit* $\pi_P : P \to P'$ *une bijection. On note* $\overline{\pi_P}(N) \stackrel{\text{def}}{=} (P', T, F', M_0', U, u_0, \sqsubseteq, \mathsf{unité}')$ *le NUPN obtenu en renommant les places de* N *suivant* π_P*, autrement dit, avec* $F' \subseteq (P' \times T) \cup (T \times P')$*,* $M_0' \subseteq P'$ *et* $\mathsf{unité}' : P' \to U$ *tels que :*

- $\forall (p,t) \in P \times T, (\pi_P(p), t) \in F' \Leftrightarrow (p,t) \in F$;
- $\forall (t,p) \in T \times P, (t, \pi_P(p)) \in F' \Leftrightarrow (t,p) \in F$;
- $\forall p \in P, \pi_P(p) \in M_0' \Leftrightarrow p \in M_0$;
- $\mathsf{unité}' = \mathsf{unité} \circ (\pi_P^{-1})$.

Notons que, puisque π_P est bijective, $|P| = |P'|$, $|F| = |F'|$ et $|M_0| = |M_0'|$.

Définition 4.5 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *et soit* $\pi_T : T \to T'$ *une bijection. On note* $\overline{\pi_T}(N) \stackrel{\text{def}}{=} (P, T', F', M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *le NUPN obtenu en renommant les transitions de* N *suivant* π_T*, autrement dit, avec* $F' \subseteq (P \times T') \cup (T' \times P)$ *tel que :*

- $\forall (p,t) \in P \times T, (p, \pi_T(t)) \in F' \Leftrightarrow (p,t) \in F$;
- $\forall (t,p) \in T \times P, (\pi_T(t), p) \in F' \Leftrightarrow (t,p) \in F$.

Notons que, puisque π_T est bijective, $|T| = |T'|$ et $|F| = |F'|$.

Définition 4.6 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *et soit* $\pi_U : U \to U'$ *une bijection. On note* $\overline{\pi_U}(N) \stackrel{\text{def}}{=} (P, T, F, M_0, U', u_0', \sqsubseteq', \mathsf{unité}')$ *le NUPN obtenu en renommant les unités de* N *suivant* π_U*, c.-à-d., avec* u_0'*,* $(\sqsubseteq') \subseteq U' \times U'$ *et* $\mathsf{unité}' : P \to U'$ *tels que :*

- $u_0' = \pi_U(u_0)$;
- $\forall u_1, u_2 \in U, \pi_U(u_1) \sqsubseteq' \pi_U(u_2) \Leftrightarrow u_1 \sqsubseteq u_2$;
- $\mathsf{unité}' = \pi_U \circ \mathsf{unité}$.

Notons que, puisque π_U est bijective, $|\sqsubseteq| = |\sqsubseteq'|$.

Remarquons que pour toutes bijections π_P, π_T et π_U, les fonctions $\overline{\pi_P}$, $\overline{\pi_T}$ et $\overline{\pi_U}$ sont commutatives.

Définition 4.7 *Soient deux NUPN* N *et* N' *tels que* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *et* $N' = (P', T', F', M_0', U', u_0', \sqsubseteq', \mathsf{unité}')$*. Ces réseaux sont dits* isomorphes ssi *il existe trois bijections* $\pi_P : P \to P'$*,* $\pi_T : T \to T'$ *et* $\pi_U : U \to U'$ *telles que* $N' = (\overline{\pi_U} \circ \overline{\pi_T} \circ \overline{\pi_P})(N)$.

D'après la définition 4.7, deux NUPN sont isomorphes si leurs réseaux de Petri sous-jacents sont identiques à la permutation de leurs noms de places et de transitions près et qu'ils disposent de la même information NUPN.

Notons que si deux NUPN sont isomorphes, alors leurs réseaux de Petri sous-jacents sont isomorphes, mais que la réciproque est fausse : par exemple, les NUPN des figures 3.1 (à gauche), 3.3 (à droite) et 3.5 sont deux à deux non-isomorphes, alors que leurs réseaux de Petri sous-jacents sont deux à deux isomorphes.

La définition 4.7 n'est pas une définition « comportementale », car elle ne repose pas sur l'exploration des marquages accessibles des réseaux : seuls les marquages initiaux sont considérés. Cette définition, qui ne tient compte que des structures des réseaux en entrée, est donc purement « structurelle ». Les trois raisons suivantes conduisent à ce choix :

— En s'affranchissant du parcours des marquages accessibles qui, en général, ne peut être exhaustif sur les modèles de grande taille, une définition structurelle « bien choisie » est a priori moins coûteuse qu'une définition comportementale.

— Deux réseaux peuvent avoir des ensembles de marquages accessibles identiques ou isomorphes, mais présenter des topologies distinctes avec, par exemple, des nombres différents d'unités, de places ou de transitions.

— Avec la définition adoptée ici, deux réseaux isomorphes ont des graphes des marquages accessibles isomorphes : il n'est pas donc pas nécessaire de déterminer leurs graphes respectifs pour le prouver.

La proposition suivante caractérise la relation d'isomorphisme, en vue de l'introduction du problème traité dans ce chapitre.

Proposition 4.1 *L'isomorphisme de NUPN est une relation d'équivalence.*

Preuve. Une fonction identité ayant pour co-domaine son domaine est bijective, l'inverse f^{-1} d'une bijection f est bijective sur son domaine de définition et la composée $f \circ g$ de bijections est une bijection si le co-domaine de g est égal au domaine de f. De ce fait, les démonstrations de la réflexivité, de la symétrie et de la transitivité de la relation d'isomorphisme de NUPN sont immédiates. □

En outre, cette proposition implique que la négation de la relation d'isomorphisme de réseaux est anti-réflexive et symétrique (mais elle n'est ni transitive, ni anti-transitive).

De cette relation d'isomorphisme, le problème traité dans ce chapitre, c.-à-d. celui de l'identification des réseaux doublons au sein d'une collection de NUPN, est introduit formellement comme suit.

Définition 4.8 *Le* problème des réseaux isomorphes *consiste en la détection de toutes les paires de réseaux isomorphes au sein d'une collection de NUPN. De manière équivalente, puisque la relation d'isomorphisme est une relation d'équivalence, ce problème consiste en la détermination de toutes les classes d'équivalence de la collection, c.-à-d. une partition de la collection en ensembles maximaux (au sens de l'inclusion) de réseaux mutuellement*

isomorphes.

4.3 Travaux voisins

À notre connaissance, il existe peu de travaux antérieurs relatifs au problème abordé dans ce chapitre, consistant au partitionnement efficace d'un jeu de tests composé de NUPN (ou de réseaux de Petri), selon une relation d'isomorphisme. Ceci est vraisemblablement lié au fait que la constitution de vastes collections de réseaux, comprenant plusieurs milliers de modèles, est un phénomène relativement récent.

Toutefois, des définitions de l'isomorphisme de réseaux de Petri existent dans certaines publications, parfois différentes de celle de la définition 4.3. Dans [RE96, Def. 27] se trouve une définition de l'isomorphisme prenant en considération les places, les transitions, les arcs, ainsi que le marquage initial. La définition donnée dans [HK98] prend en considération les places, les étiquettes de places, les transitions, les étiquettes de transitions et les poids des arcs, mais pas le marquage initial. La définition des réseaux « isomorphes enracinés » de [Gor17] prend en considération les places, les transitions, les étiquettes de transitions et le marquage initial. Une définition moins générale de l'isomorphisme de réseaux est donnée par [DS18, Dev21], dans laquelle seules les places sont renommées et les transitions sont inchangées : elle revient à utiliser la définition 4.3, avec les restrictions $T = T'$ et $\pi_T = \mathsf{Id}$.

Pour l'isomorphisme de réseaux de Petri, outre les définitions « structurelles » citées dans le paragraphe ci-dessus, il existe des définitions « comportementales ». Les trois notions d'équivalences de réseaux de Petri élémentaires proposées dans [RE96, Def. 28, 31 et 32], reposent sur les marquages accessibles. Dans [DR96], [BD96] et [BD15], l'isomorphisme est une relation binaire entre les graphes des marquages accessibles des réseaux de Petri et les systèmes de transition d'états. Dans [PCR01], l'isomorphisme est une relation binaire entre les graphes des marquages accessibles des réseaux de Petri et les algèbres de Boole. Rappelons que ces définitions comportementales sont impliquées par la définition 4.3, qui est structurelle.

Notons enfin, de manière plus distante de notre problématique, l'isomorphisme est défini dans [Cap11] entre les marquages d'un réseau non-coloré (dit de « niveau de base ») et ceux d'un réseau coloré (dit de « niveau méta »).

4.4 Complexité du problème

Cette section vise à établir les complexités théoriques des problèmes de l'isomorphisme de réseaux de Petri et de l'isomorphisme de NUPN.

Proposition 4.2 *Le problème de l'isomorphisme de graphes se ramène au problème d'isomorphisme de réseaux de Petri.*

Preuve. Soit une fonction N associant à tout graphe (V, E) un réseau de Petri $(V, T, F, \varnothing)$ tel que : (i) $|T| = |E|$; (ii) $F \subseteq V \times T$; (iii) $\forall \{v_1, v_2\} \in E, \exists t \in T \mid {}^\bullet t = \{v_1, v_2\}$ et

(iv) $\forall t_1, t_2 \in T, ({}^\bullet t_1 \cap {}^\bullet t_2 \neq \varnothing) \Rightarrow (t_1 = t_2)$. On montre facilement que les graphes G et G' sont isomorphes *ssi* les réseaux de Petri $N(G)$ et $N(G')$ sont isomorphes. □

Proposition 4.3 *Le problème de l'isomorphisme de réseaux de Petri se ramène au problème d'isomorphisme de NUPN.*

Preuve. On montre facilement que deux réseaux de Petri sont isomorphes *ssi* leurs NUPN triviaux correspondants selon la preuve de la proposition 3.1 sont isomorphes. □

Proposition 4.4 *Le problème de l'isomorphisme de NUPN se ramène au problème d'isomorphisme de graphes orientés et étiquetés.*

Preuve. Soit une fonction G associant à tout NUPN $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un graphe orienté et étiqueté (V, E, c) tel que :

- $V \stackrel{\text{def}}{=} P \cup T \cup U$
- $E \stackrel{\text{def}}{=} F \cup \{(p, \mathsf{unité}\,(p)) \in P \times U\} \cup \{(u_1, u_2) \in U \times U \mid u_1 \in \mathsf{sous\text{-}unités}\,(u_2)\}$
- $\forall v \in V, \mathsf{c}\,(v) \stackrel{\text{def}}{=} \begin{cases} \mathsf{si}\ v \in M_0\ \mathsf{alors}\ 0 \\ \mathsf{si}\ v \in P \setminus M_0\ \mathsf{alors}\ 1 \\ \mathsf{si}\ v \in T\ \mathsf{alors}\ 2 \\ \mathsf{si}\ v \in U\ \mathsf{alors}\ 3 \end{cases}$

Dans le graphe, les étiquettes permettent de distinguer trois types de sommets (P, T et U) et de reconnaître ceux correspondant à des places initiales. Les arcs du graphe permettent de décrire les arcs du réseau de Petri sous-jacent, association des places vers les unités et l'arbre des unités.
On montre facilement que les NUPN N et N' sont isomorphes *ssi* les graphes orientés et étiquetés $G(N)$ et $G(N')$ sont isomorphes. □

Ces trois propositions ci-dessus (combinées au fait que problème d'isomorphisme de graphes orientés et étiquetés est de même complexité théorique que le problème d'isomorphisme de graphes non-orientés et non-étiquetés [Mil79] [BC79]) permettent de mettre en évidence que les problèmes suivants sont, d'un point de vue théorique, équivalents : (i) l'isomorphisme de réseaux de Petri ; (ii) l'isomorphisme de NUPN ; (iii) et l'isomorphisme de graphes étiquetés (ou non) et orientés (ou non).

Maintenant que nous avons établi que l'isomorphisme de NUPN et que l'isomorphisme de réseaux de Petri sont deux problèmes équivalents à celui de l'isomorphisme de graphes (c.-à-d. GI-complets, GI faisant référence à la classe de complexité « Graph Isomorphism »), remarquons que le meilleur algorithme connu à l'heure actuelle pour l'isomorphisme de graphes présente un coût en temps quasi-polynomial [Bab16, Bab19]. Mais, à notre connaissance, les outils disponibles ont tous un coût en temps exponentiel dans le pire des cas [JK07] [Gre09] [KSM12] [LCA13] [MP14].

Nous allons maintenant présenter quatre approches pour la détection de l'isomorphisme ou du non-isomorphisme de NUPN. Ces approches, conçues pour les NUPN, englobent les

réseaux de Petri en tant que cas particulier, à savoir en tant que NUPN triviaux.

4.5 Signatures de réseaux

La première approche repose sur l'idée de *signature de réseaux*, puisant ses fondements dans les notions de signatures, d'empreintes, et de fonctions de hachage.

Définition 4.9 *Une* signature de réseau *(ou en abrégé,* signature*) est une fonction* sig *définie sur les NUPN, telle que, pour deux réseaux quelconques N et N', si N et N' sont isomorphes, alors* $\mathsf{sig}(N) \stackrel{\text{def}}{=} \mathsf{sig}(N')$.

En pratique, la contraposition de cette expression est utilisée : deux réseaux ayant des signatures différentes ne sont pas isomorphes. La réciproque de cette expression est fausse : deux réseaux ayant la même signature ne sont pas nécessairement isomorphes (il peut subsister un risque de collision de leurs signatures). Conséquemment, il n'est pas souhaitable de considérer deux réseaux isomorphes par la seule comparaison de leurs signatures.

Proposition 4.5 *Si* sig *est une signature, alors pour tout réseau N et pour toute permutation π_x de places, de transitions ou d'unités,* $(\mathsf{sig} \circ \overline{\pi_x})(N) = \mathsf{sig}(N)$.

Pour être effective (c.-à-d. pour permettre la discrimination d'autant de réseaux que possible), une signature doit être suffisamment exhaustive pour couvrir une grande partie (voire la quasi-totalité) des informations invariantes par permutations, mais elle doit également se montrer suffisamment rapide à calculer. Dans ce qui suit, nous proposons une fonction sig définie sur les NUPN.

Définition 4.10 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN. Nous définissons* $\mathsf{sig}(N)$ *comme étant un 32-uplet, de taille fixe, constitué de nombres naturels ou d'n-uplets de nombres naturels calculés sur N. Les constituants de cet ensemble se classent en trois parties : les éléments reposant sur les places, les éléments reposant sur les transitions, et les éléments reposant sur les unités de N. Ces parties, respectivement notées $(h_1, \ldots)$, $(k_1, \ldots)$ et $(l_1, \ldots)$, sont décrites dans les définitions 4.16, 4.17 et 4.18, ci-dessous.*

4.5.1 Multi-ensembles et fonctions de hachage

Cette sous-section présente des notions essentielles pour la suite de cette section, à savoir les notions de *multi-ensembles*, (qui sont notés $\{\!| \ldots |\!\}$ pour les distinguer des ensembles (normaux), notés $\{\ldots\}$), de *fonctions de hachage*, ainsi que le type $\mathbb{D}$.

Soit $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un NUPN : deux exemples de multi-ensembles sont $\{\!| |{}^\bullet t| \mid t \in T |\!\}$ et $\{\!| \mathsf{hauteur}(u) \mid u \in U |\!\}$. Pour vérifier l'égalité de ces multi-ensembles, dont les dimensions peuvent être fixes ou variables, nous adoptons une approche reposant sur le *hachage*, qui convertit chaque multi-ensemble en une structure de type $\mathbb{D}$ (de taille fixe), de sorte que l'égalité de deux multi-ensembles implique l'égalité des deux champs $\mathbb{D}$ correspondants. La fonction de hachage choisie n'est donc pas censée être parfaite (il peut y avoir des collisions de hachage).

Cependant, il est souhaitable que cette fonction renvoie un résultat indépendant de l'ordre des éléments (les multi-ensembles ne sont pas ordonnés). Une solution simple consisterait à trier les éléments d'un multi-ensemble et à les concaténer pour former une chaîne de bits, sur laquelle une fonction de hachage standard (cryptographique ou non) serait appliquée. Cependant, cette approche est lente (au moins en raison du tri) et produit un résultat ne pouvant pas être compris par les humains. Nous adoptons donc une approche alternative basée sur la définition suivante.

Définition 4.11 *Concernant les* fonctions de hachage*, nous définissons :*

- *le type* $\mathbb{D} \stackrel{\text{def}}{=} \mathbb{N}^5$ *est un* 5*-uplet d'entiers naturels ;*
- *les éléments de tout* 5*-uplet* $d \in \mathbb{D}$ *sont dénommés ainsi :* $d.\mathsf{card} \stackrel{\text{def}}{=} d[1]$*,* $d.\mathsf{min} \stackrel{\text{def}}{=} d[2]$*,* $d.\mathsf{max} \stackrel{\text{def}}{=} d[3]$*,* $d.\mathsf{sum} \stackrel{\text{def}}{=} d[4]$ *et* $d.\mathsf{prod} \stackrel{\text{def}}{=} d[5]$ *;*
- $\mathcal{H} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{D}$ *est une fonction de* hachage*, définie sur les multi-ensembles d'entiers naturels, telle que*[1] *:*
  $\mathcal{H}(M).\mathsf{card} \stackrel{\text{def}}{=} \sum_{x \in \mathrm{dom}_M} M(x)$,
  $\mathcal{H}(M).\mathsf{min} \stackrel{\text{def}}{=} \mathsf{min}\,(\mathrm{dom}_M)$,
  $\mathcal{H}(M).\mathsf{max} \stackrel{\text{def}}{=} \mathsf{max}\,(\mathrm{dom}_M)$,
  $\mathcal{H}(M).\mathsf{sum} \stackrel{\text{def}}{=} \sum_{x \in \mathrm{dom}_M} M(x) \times x$,
  $\mathcal{H}(M).\mathsf{prod} \stackrel{\text{def}}{=}$ si $\mathrm{dom}_M \neq \varnothing$ alors $(\prod_{x \in \mathrm{dom}_M} (2 \times x + 2\ 654\ 435\ 769)^{M(x)})/2$ sinon 1 *;*
- $\mathcal{M} : (\mathbb{D} \to \mathbb{N}) \to \mathbb{D}$ *est une fonction de « hachage de hachages », définie sur les multi-ensembles de* $\mathbb{D}$*, telle que :*
  $\mathcal{M}(M).\mathsf{card} \stackrel{\text{def}}{=} \sum_{x \in \mathrm{dom}_M} M(x) \times x.\mathsf{card}$,
  $\mathcal{M}(M).\mathsf{min} \stackrel{\text{def}}{=} \mathsf{min}\,(\{x.\mathsf{min} \mid x \in \mathrm{dom}_M\})$,
  $\mathcal{M}(M).\mathsf{max} \stackrel{\text{def}}{=} \mathsf{max}\,(\{x.\mathsf{max} \mid x \in \mathrm{dom}_M\})$,
  $\mathcal{M}(M).\mathsf{sum} \stackrel{\text{def}}{=} \sum_{x \in \mathrm{dom}_M} M(x) \times x.\mathsf{sum}$,
  $\mathcal{M}(M).\mathsf{prod} \stackrel{\text{def}}{=}$ si $\mathrm{dom}_M \neq \varnothing$ alors $(\prod_{x \in \mathrm{dom}_M} (2 \times x.\mathsf{prod} + 1)^{M(x)})/2$ sinon 1.

La fonction $\mathcal{H}$ traite les multi-ensembles d'entiers naturels, tandis que la fonction $\mathcal{M}$, traite les multi-ensembles de $\mathbb{D}$ à un niveau supérieur (« hachage de hachages »). Toutes les opérations impliquées dans $\mathcal{H}$ et $\mathcal{M}$ étant commutatives et associatives, nul besoin de procéder à un tri préliminaire des multi-ensembles d'entrée : les deux fonctions peuvent être calculées par induction sur la taille n de leurs entrées En pratique, les champs du type $\mathbb{D}$ sont implantés en utilisant des entiers. machine, de sorte que tous les calculs arithmétiques sont effectués modulo, par exemple, 2^{32} ou 2^{64}. Tous les constituants du type $\mathbb{D}$, à l'exception notable de prod, sont lisibles par les humains et expriment des propriétés signifiantes du multi-ensemble correspondant. En revanche, prod utilise une forme de hachage multiplicatif[2] visant à améliorer la dispersion pour les grands multi-ensembles.

1. En particulier, nous avons : $\mathcal{H}(\{\!|\ |\!\}) = \mathcal{H}(x \mapsto 0) = (0, +\infty, -\infty, 0, 1)$.
2. stackoverflow.com/questions/1536393/good-hash-function-for-permutations/1537189#1537189

Chaque facteur de prod étant impair, leurs produits ne peuvent jamais s'annuler, y compris en arithmétique modulaire ; en conséquence, leur bit de poids faible étant toujours égal à un, il est systématiquement éliminé par une division par deux. Ceci permet de gagner, avec une représentation bornée des nombres entiers, un bit de poids fort.

4.5.2 Attributs pour les places et les transitions

La définition de la fonction de signature repose sur divers attributs calculés pour chaque place et chaque transition du réseau. Ces attributs contiennent des informations qui permettent de différencier les places entres elles et les transitions entre elles. De prime abord, toutes les places se ressemblent (sauf celles du marquage initial), mais celles-ci peuvent être distinguées à l'aide d'informations locales (par exemple, le nombre d'arcs et de transitions qui leurs sont connectés) ainsi que d'informations globales (par exemple, leurs distances par rapport à d'autres places, notoires : places initiales, places puits, et ainsi de suite.)

Définition 4.12 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN. Un ensemble de places* $p_0, \ldots, p_n$ *et un ensemble de transitions* $t_1, \ldots, t_n$ *sont dits d'être une* chaîne de longueur n depuis p_0 vers p_n ssi $\forall i \in \{1 : n\},\ (p_{i-1}, t_i) \in F \wedge (t_i, p_i) \in F$. *Étant donné deux places* p *et* p', *la* distance de p à p' *est définie comme la longueur de la plus courte chaîne possible allant de* p *à* p' *; s'il n'existe pas de telle chaîne, cette distance est égale à* $|P| + 1$ [3].

Définition 4.13 *À chaque place* p, *les trois attributs entiers suivants sont associés :*

- $\mathsf{distance1}\,(p)$ *est définie comme étant la distance minimale de* p *à toute place du marquage initial* M_0 *;*
- $\mathsf{distance2}\,(p)$ *est définie comme étant la distance minimale de toute place du marquage initial* M_0 *à* p *;*
- $\mathsf{distance3}\,(p)$ *est définie comme étant la distance minimale de* p *à toute place puits (c.-à-d., toute place* p' *telle que* $p'^{\bullet} = \varnothing$*).*

Définition 4.14 *À chaque transition* t, *les trois attributs booléens et les six attributs de type* $\mathbb{D}$ *suivants sont associés :*

- $\mathsf{décroissante}\,(t) \stackrel{\text{def}}{=} |{}^{\bullet}t| > |t^{\bullet}|$ *;*
- $\mathsf{conservatrice}\,(t) \stackrel{\text{def}}{=} |{}^{\bullet}t| = |t^{\bullet}|$ *;*
- $\mathsf{croissante}\,(t) \stackrel{\text{def}}{=} |{}^{\bullet}t| < |t^{\bullet}|$ *;*
- *pour* i *allant de un à trois :* $\mathsf{distance\_d'entrée}\,i(t) \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{distance}\,i(p) \mid p \in {}^{\bullet}t \,|\!\})$ *;*
- *pour* i *allant de un à trois :* $\mathsf{distance\_de\_sortie}\,i(t) \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{distance}\,i(p) \mid p \in t^{\bullet} \,|\!\})$.

Nous pouvons constater que les prédicats booléens « décroissante », « conservatrice » et « croissante » forment en réalité une partition des transitions du réseau : ils sont mu-

3. La valeur $|P| + 1$ est à considérer comme une valeur infinie.

tuellement exclusifs et leur somme est égale au nombre de transitions. De plus, pour tout i allant de un à trois et pour toute transition t, $\mathsf{distance\_d'entrée}\, i(t).\mathsf{card} = |{}^\bullet t|$ et $\mathsf{distance\_de\_sortie}\, i(t).\mathsf{card} = |t^\bullet|$.

Définition 4.15 *À chaque place p sont associés sept attributs de type entiers naturels et seize attributs de type $\mathbb{D}$:*

- $\mathsf{nb\_de\_boucles}\,(p) \stackrel{\text{def}}{=} |{}^\bullet p \cap p^\bullet|$;
- $\mathsf{nb\_de\_transitions\_d'entrée\_décroissantes}\,(p) \stackrel{\text{def}}{=} |\{t \in {}^\bullet p \mid \mathsf{décroissante}(t)\}|$;
- $\mathsf{nb\_de\_transitions\_d'entrée\_conservatrices}\,(p) \stackrel{\text{def}}{=} |\{t \in {}^\bullet p \mid \mathsf{conservatrice}(t)\}|$;
- $\mathsf{nb\_de\_transitions\_d'entrée\_croissantes}\,(p) \stackrel{\text{def}}{=} |\{t \in {}^\bullet p \mid \mathsf{croissante}(t)\}|$;
- $\mathsf{nb\_de\_transitions\_de\_sortie\_décroissantes}\,(p) \stackrel{\text{def}}{=} |\{t \in p^\bullet \mid \mathsf{décroissante}(t)\}|$;
- $\mathsf{nb\_de\_transitions\_de\_sortie\_conservatrices}\,(p) \stackrel{\text{def}}{=} |\{t \in p^\bullet \mid \mathsf{conservatrice}(t)\}|$;
- $\mathsf{nb\_de\_transitions\_de\_sortie\_croissantes}\,(p) \stackrel{\text{def}}{=} |\{t \in p^\bullet \mid \mathsf{croissante}(t)\}|$;
- $\mathsf{nb\_de\_places\_d'entrée\_pred}\,(p) \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |{}^\bullet t| \mid t \in {}^\bullet p \,|\!\})$;
- $\mathsf{nb\_de\_places\_de\_sortie\_pred}\,(p) \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |t^\bullet| \mid t \in {}^\bullet p \,|\!\})$;
- $\mathsf{nb\_de\_places\_d'entrée\_succ}\,(p) \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |{}^\bullet t| \mid t \in p^\bullet \,|\!\})$;
- $\mathsf{nb\_de\_places\_de\_sortie\_succ}\,(p) \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |t^\bullet| \mid t \in p^\bullet \,|\!\})$;
- *pour i allant de un à trois :*
$$\mathsf{distance\_d'entrée\_pred}\, i(p) \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{distance\_d'entrée}\, i(t) \mid t \in {}^\bullet p \,|\!\}) \quad \textit{et}$$
$$\mathsf{distance\_de\_sortie\_pred}\, i(p) \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{distance\_de\_sortie}\, i(t) \mid t \in {}^\bullet p \,|\!\})\ ;$$
- *pour i allant de un à trois :*
$$\mathsf{distance\_d'entrée\_succ}\, i(p) \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{distance\_d'entrée}\, i(t) \mid t \in p^\bullet \,|\!\}) \quad \textit{et}$$
$$\mathsf{distance\_de\_sortie\_succ}\, i(p) \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{distance\_de\_sortie}\, i(t) \mid t \in p^\bullet \,|\!\}).$$

Notons $F^-(p) \stackrel{\text{def}}{=} (T \times \{p\}) \cap F$ les arcs entrant vers p et $F^+(p) \stackrel{\text{def}}{=} (\{p\} \times T) \cap F$ les arcs sortant de p. Pour toute place p et pour i allant de un à trois, nous avons :

- $|F^-(p)| = \mathsf{nb\_de\_places\_d'entrée\_pred}\, i(p).\mathsf{card}$
$= \mathsf{nb\_de\_places\_de\_sortie\_pred}\, i(p).\mathsf{card}$
$= \mathsf{nb\_de\_transitions\_d'entrée\_décroissantes}\,(p) +$
$\mathsf{nb\_de\_transitions\_d'entrée\_conservatrices}\,(p) +$
$\mathsf{nb\_de\_transitions\_d'entrée\_croissantes}\,(p)$;
- $|F^+(p)| = \mathsf{nb\_de\_places\_d'entrée\_succ}\, i(p).\mathsf{card}$
$= \mathsf{nb\_de\_places\_de\_sortie\_succ}\, i(p).\mathsf{card}$
$= \mathsf{nb\_de\_transitions\_de\_sortie\_décroissantes}\,(p) +$
$\mathsf{nb\_de\_transitions\_de\_sortie\_conservatrices}\,(p) +$
$\mathsf{nb\_de\_transitions\_de\_sortie\_croissantes}\,(p)$;
- $\mathsf{distance\_d'entrée\_pred}\, i(p).\mathsf{card} = \mathsf{nb\_de\_places\_d'entrée\_pred}\,(p).\mathsf{sum}$;
- $\mathsf{distance\_de\_sortie\_pred}\, i(p).\mathsf{card} = \mathsf{nb\_de\_places\_de\_sortie\_pred}\,(p).\mathsf{sum}$;
- $\mathsf{distance\_d'entrée\_succ}\, i(p).\mathsf{card} = \mathsf{nb\_de\_places\_d'entrée\_succ}\,(p).\mathsf{sum}$;

— $\mathsf{distance\_de\_sortie\_succ}\, i(p).\mathsf{card} = \mathsf{nb\_de\_places\_de\_sortie\_succ}\,(p).\mathsf{sum}$.

4.5.3 Part de la signature fondée sur les places

La première partie de notre fonction de signature concerne les places et est définie comme indiqué ci-après.

Définition 4.16 *Soit $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un NUPN. Dans la définition 4.10, la* partie reposant sur les places *de la fonction* $\mathsf{sig}\,(N)$ *est un* 16*-uplet* $(h_1, \ldots, h_{16})$ *constitué de nombres naturels ou de valeurs de type* $\mathbb{D}$ *calculées sur* N*, tel que :*

— $h_1 \stackrel{\text{def}}{=} |P|$, *autrement dit, le nombre de places ;*
— $h_2 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{distance1}\,(p) \mid p \in P \,|\!\})$;
— $h_3 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{distance2}\,(p) \mid p \in P \,|\!\})$;
— $h_4 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{distance3}\,(p) \mid p \in P \,|\!\})$;
— $h_5 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_boucles}\,(p) \mid p \in P \,|\!\})$;
— $h_6 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_de\_transitions\_d'entrée\_décroissantes}\,(p) \mid p \in P \,|\!\})$;
— $h_7 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_de\_transitions\_d'entrée\_conservatrices}\,(p) \mid p \in P \,|\!\})$;
— $h_8 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_de\_transitions\_d'entrée\_croissantes}\,(p) \mid p \in P \,|\!\})$ [4] ;
— $h_9 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_de\_transitions\_de\_sortie\_décroissantes}\,(p) \mid p \in P \,|\!\})$;
— $h_{10} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_de\_transitions\_de\_sortie\_conservatrices}\,(p) \mid p \in P \,|\!\})$;
— $h_{11} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{nb\_de\_transitions\_de\_sortie\_croissantes}\,(p) \mid p \in P \,|\!\})$ [5] ;
— $h_{12} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{c}_2\,(|{}^\bullet p|, |p^\bullet|) \mid p \in P \,|\!\})$, *où* $\mathsf{c}_2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *est une bijection réalisant une fonction de couplage ;*
— $h_{13} \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{nb\_de\_places\_d'entrée\_pred}\,(p) \mid p \in P \,|\!\})$;
— $h_{14} \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{nb\_de\_places\_de\_sortie\_pred}\,(p) \mid p \in P \,|\!\})$;
— $h_{15} \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{nb\_de\_places\_d'entrée\_succ}\,(p) \mid p \in P \,|\!\})$;
— $h_{16} \stackrel{\text{def}}{=} \mathcal{M}(\{\!|\, \mathsf{nb\_de\_places\_de\_sortie\_succ}\,(p) \mid p \in P \,|\!\})$.

Les composants suivants sont exclus de la partie de la signature basée sur les places : $h_2.\mathsf{card}$, $h_3.\mathsf{card}$, *...,* $h_{12}.\mathsf{card}$ *(puisque tous égaux à* h_1*) ;* $h_{13}.\mathsf{card}$ *et* $h_{14}.\mathsf{card}$ *(qui sont égaux à* $|F^-|$*) ;* $h_{15}.\mathsf{card}$ *et* $h_{16}.\mathsf{card}$ *(qui sont égaux à* $|F^+|$*) ; et* $h_{12}.\mathsf{sum}$ *(qui est une combinaison linéaire de* $h_6.\mathsf{sum}$, *...,* $h_{11}.\mathsf{sum}$*).*

4.5.4 Part de la signature fondée sur les transitions

La seconde partie de notre fonction de signature concerne les transitions et est définie comme indiqué ci-après.

4. À noter que : $h_6.\mathsf{sum} + h_7.\mathsf{sum} + h_8.\mathsf{sum} = |F^-|$.
5. À noter que : $h_9.\mathsf{sum} + h_{10}.\mathsf{sum} + h_{11}.\mathsf{sum} = |F^+|$.

Définition 4.17 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN. Dans la définition 4.10, la* partie reposant sur les transitions *de la fonction* $\mathsf{sig}\,(N)$ *est un triplet* (k_1, k_2, k_3) *constitué de nombres naturels ou de valeurs de type* $\mathbb{D}$ *calculées sur* N*, tel que :*

- $k_1 \stackrel{\text{def}}{=} |T|$*, c.-à-d., le nombre de transitions ;*
- $k_2 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |{}^\bullet t| \mid t \in T \,|\!\})$ *;*
- $k_3 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |t^\bullet| \mid t \in T \,|\!\})$*.*

Les éléments suivants sont exclus de la partie liée au transitions de la signature : $k_2.\mathsf{card}$ *et* $k_3.\mathsf{card}$ *(car égaux à* k_1*) ;* $k_2.\mathsf{sum}$ *(égal à* $|F^+|$*, c.-à-d. à* $h_9.\mathsf{sum} + h_{10}.\mathsf{sum} + h_{11}.\mathsf{sum}$*) ; et* $k_3.\mathsf{sum}$ *(égal à* $|F^-|$*, c.-à-d. à* $h_6.\mathsf{sum} + h_7.\mathsf{sum} + h_8.\mathsf{sum}$*).*

4.5.5 Part de la signature fondée sur les unités

La troisième partie de notre fonction de signature concerne les unités et est définie comme indiqué ci-après.

Définition 4.18 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN. Dans la définition 4.10, la* partie reposant sur les unités *de la fonction* $\mathsf{sig}\,(N)$ *est un* 13*-uplet* $(l_1, \ldots, l_{13})$*, constitué de nombres naturels ou de valeurs de type* $\mathbb{D}$ *calculées sur* N*, tel que :*

- $l_1 \stackrel{\text{def}}{=} |U|$*, c.-à-d., le nombre d'unités ;*
- $l_2 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |\mathsf{sous\text{-}unités}\,(u)| \mid u \in U \,|\!\})$ *;*
- $l_3 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |\mathsf{sous\text{-}unités}^*(u)| \mid u \in U \,|\!\})$ *;*
- $l_4 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |\mathsf{places}\,(u)| \mid u \in U \,|\!\})$ *;*
- $l_5 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |\mathsf{places}^*(u)| \mid u \in U \,|\!\})$ *;*
- $l_6 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |\mathsf{places}\,(u) \cap M_0| \mid u \in U \,|\!\})$ *;*
- $l_7 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, |\mathsf{places}^*(u) \cap M_0| \mid u \in U \,|\!\})$ *;*
- $l_8 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{profondeur}\,(u) \mid u \in U \,|\!\})$ *;*
- $l_9 \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{hauteur}\,(u) \mid u \in U \,|\!\})$ *;*
- $l_{10} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{largeur}\,(u) \mid u \in U \,|\!\})$ [6] *;*
- $l_{11} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{entrée}\,(u) \mid u \in U \,|\!\})$*, où* $\mathsf{entrée}\,(u) \stackrel{\text{def}}{=} \sum_{t \in T} |{}^\bullet t \cap \mathsf{places}\,(u)|$ *;*
- $l_{12} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{sortie}\,(u) \mid u \in U \,|\!\})$*, où* $\mathsf{sortie}\,(u) \stackrel{\text{def}}{=} \sum_{t \in T} |t^\bullet \cap \mathsf{places}\,(u)|$ *;*
- $l_{13} \stackrel{\text{def}}{=} \mathcal{H}(\{\!|\, \mathsf{c}_{11}\,(|\mathsf{sous\text{-}unités}\,(u)|, |\mathsf{sous\text{-}unités}^*(u)|, |\mathsf{places}\,(u)|, |\mathsf{places}^*(u)|, |\mathsf{places}\,(u) \cap M_0|, |\mathsf{places}^*(u) \cap M_0|, \mathsf{profondeur}\,(u), \mathsf{hauteur}\,(u), \mathsf{largeur}\,(u), \mathsf{entrée}\,(u), \mathsf{sortie}\,(u)) \,|\!\})$*, où* $\mathsf{c}_{11} : \mathbb{N}^{11} \to \mathbb{N}$ *est une fonction de couplage généralisée.*

Les composants suivants sont exclus de la partie de la signature basée sur les unités car leur présence serait redondante : $l_2.\mathsf{card} = l_1$*,* $l_2.\mathsf{min} = 0$*,* $l_2.\mathsf{sum} = l_1 - 1$*,* $l_3.\mathsf{card} = l_1$*,* $l_3.\mathsf{min} = 0$*,* $l_3.\mathsf{max} = l_1 - 1$*,* $l_4.\mathsf{card} = l_1$*,* $l_4.\mathsf{sum} = |P|$*,* $l_5.\mathsf{card} = l_1$*,* $l_6.\mathsf{card} = l_1$*,* $l_6.\mathsf{min} \leq 1$

6. À noter que $l10.\mathsf{max} = \mathsf{largeur}\,(u_0)$

et $l_6.\mathsf{max} \leq 1$ *si* N *est unit-sauf,* $l_7.\mathsf{card} = l_1$, $l_7.\mathsf{min} \leq 1$ *si* N *est unit-sauf,* $l_7.\mathsf{max} = |M_0|$, $l_8.\mathsf{card} = l_1$, $l_8.\mathsf{min} = 0$, $l_9.\mathsf{card} = l_1$, $l_9.\mathsf{min} = 1$, $l_9.\mathsf{max} = l_8.\mathsf{max} + 1$, $l_{10}.\mathsf{card} = l_1$, $l_{10}.\mathsf{min} = 1$, $l_{11}.\mathsf{card} = l_1$, $l_{12}.\mathsf{card} = l_1$, $l_{13}.\mathsf{card} = l_1$, $l_{11}.\mathsf{sum} = |F^+|$ *et* $l_{12}.\mathsf{sum} = |F^-|$.

4.6 Canonisation de réseaux

La seconde approche repose sur l'idée de *canonisation de réseaux*, puisant ses racines dans la notion de « forme normale ».

Définition 4.19 *Une* canonisation de réseaux *(où, en abrégé,* canonisation*) est une fonction* can *définie sur les NUPN, telle que si deux réseaux* N *et* N' *vérifient* $\mathsf{can}\,(N) = \mathsf{can}(N')$, *alors* N *et* N' *sont isomorphes.*

Nous souhaitons que cette approche pour la canonisation de réseaux soit efficiente : l'objectif est d'être à même d'identifier un maximum de réseaux isomorphes, avec des temps de calcul raisonnables. C'est pourquoi l'implication réciproque de la définition 4.19 n'est pas requise : deux réseaux isomorphes n'ont pas nécessairement la même image par canonisation.

Dans ce qui suit, nous proposons une fonction particulière, intitulée can, définie sur les NUPN et produisant des NUPN. Les attributs des définitions 4.13 à 4.15, représentant des informations concrètes sur les réseaux et lisibles par des êtres humains, seront réutilisés pour définir trois permutations successives d'unités, de places et de transitions. La fonction can est alors caractérisée par l'application successive de ces trois permutations ; les unités sont permutées en premier, car dans un NUPN non-trivial, il y a moins d'unités que de places (dans un NUPN trivial, $|P| \leq |U| \leq |P| + 1$) ; les transitions sont permutées en dernier, car il y a d'ordinaire davantage de transitions que de places dans un réseau.

Définition 4.20 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$.

- *Soit* $\pi_{U[N]} : U \to U$ *un automorphisme désignant une permutation des unités de* N *; la définition précise de* $\pi_{U[N]}$ *est donnée ci-après dans la définition 4.22 ;*
- *soit* $\pi_{P[N]} : P \to P$ *un automorphisme désignant une permutation des places de* N *; la définition précise de* $\pi_{P[N]}$ *est donnée ci-après dans la définition 4.25 ;*
- *soit* $\pi_{T[N]} : T \to T$ *un automorphisme désignant une permutation des transitions de* N *; la définition précise de* $\pi_{T[N]}$ *est donnée ci-après dans la définition 4.27.*

Finalement, nous définissons la fonction de canonisation, intitulée can*, par la composition suivante :* $\mathsf{can}\,(N) \stackrel{\text{def}}{=} (\overline{\pi_{T[N]}} \circ \overline{\pi_{P[N]}} \circ \overline{\pi_{U[N]}})\,(N)$. *Dit autrement, il s'agit du NUPN obtenu suite aux permutations successives des unités selon* $\pi_{U[N]}$*, puis des places selon* $\pi_{P[N]}$ *et enfin des transitions selon* $\pi_{T[N]}$.

4.6.1 Classement et tri des unités

La fonction $\pi_{U[N]}$ de permutation des unités, mentionnée au sein de la définition 4.20 est définie ci-dessous.

Définition 4.21 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$*. Pour toute unité* u*, un* 35*-uplet* $m(u) \stackrel{\text{def}}{=} (m_1(u), \ldots, m_{35}(u))$ *est construit, constitué de nombres naturels ou de valeurs de type* $\mathbb{D}$ *calculées sur* N*, tel que :*

- $m_1(u) \stackrel{\text{def}}{=} \mathsf{profondeur}(u)$;
- $m_2(u) \stackrel{\text{def}}{=} |\mathsf{sous\text{-}unités}(u)|$;
- $m_3(u) \stackrel{\text{def}}{=} |\mathsf{places}(u)|$;
- $m_4(u) \stackrel{\text{def}}{=} |\mathsf{places}(u) \cap M_0|$;
- $m_5(u) \stackrel{\text{def}}{=} |\mathsf{sous\text{-}unités}^*(u)|$;
- $m_6(u) \stackrel{\text{def}}{=} |\mathsf{places}^*(u)|$;
- $m_7(u) \stackrel{\text{def}}{=} |\mathsf{places}^*(u) \cap M_0|$;
- $m_8(u) \stackrel{\text{def}}{=} \mathsf{hauteur}(u)$;
- $m_9(u) \stackrel{\text{def}}{=} \mathsf{largeur}(u)$;
- $m_{10}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{distance1}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{11}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{distance2}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{12}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{distance3}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{13}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_boucles}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{14}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_de\_transitions\_d'entrée\_décroissantes}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{15}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_de\_transitions\_d'entrée\_conservatrices}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{16}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_de\_transitions\_d'entrée\_croissantes}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{17}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_de\_transitions\_de\_sortie\_décroissantes}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{18}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_de\_transitions\_de\_sortie\_conservatrices}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{19}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{H}}(\{\!|\, \mathsf{nb\_de\_transitions\_de\_sortie\_croissantes}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{20}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{nb\_de\_places\_d'entrée\_pred}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{21}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{nb\_de\_places\_de\_sortie\_pred}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{22}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{nb\_de\_places\_d'entrée\_succ}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{23}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{nb\_de\_places\_de\_sortie\_succ}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{24}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_d'entrée\_pred\,1}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{25}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_de\_sortie\_pred\,1}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{26}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_d'entrée\_succ\,1}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{27}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_de\_sortie\_succ\,1}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{28}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_d'entrée\_pred\,2}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{29}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_de\_sortie\_pred\,2}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{30}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_d'entrée\_succ\,2}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{31}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_de\_sortie\_succ\,2}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;
- $m_{32}(u) \stackrel{\text{def}}{=} \tilde{\mathcal{M}}(\{\!|\, \mathsf{distance\_d'entrée\_pred\,3}(p) \mid p \in \mathsf{places}(u) \,|\!\})$;

— $m_{33}(u) \stackrel{\text{def}}{=} \check{\mathcal{M}}(\{\!|\, \mathsf{distance\_de\_sortie\_pred}\, 3\,(p) \mid p \in \mathsf{places}\,(u)\,|\!\})$;
— $m_{34}(u) \stackrel{\text{def}}{=} \check{\mathcal{M}}(\{\!|\, \mathsf{distance\_d'entrée\_succ}\, 3\,(p) \mid p \in \mathsf{places}\,(u)\,|\!\})$;
— $m_{35}(u) \stackrel{\text{def}}{=} \check{\mathcal{M}}(\{\!|\, \mathsf{distance\_de\_sortie\_succ}\, 3\,(p) \mid p \in \mathsf{places}\,(u)\,|\!\})$.

Les composants suivants sont exclus de $m\,(u)$ *:* $m_{10}(u).\mathsf{card}$, $m_{11}(u).\mathsf{card}$, *...,* $m_{18}(u).\mathsf{card}$ *(puisque tous égaux à* $m_3(u)$*);* $m_{20}(u).\mathsf{card}$ *et* $m_{21}(u).\mathsf{card}$ *(égaux à* $m_{14}(u).\mathsf{sum} + m_{15}(u).\mathsf{sum} + m_{16}(u).\mathsf{sum}$*);* $m_{22}(u).\mathsf{card}$ *et* $m_{23}(u).\mathsf{card}$ *(égaux à* $m_{17}(u).\mathsf{sum} + m_{18}(u).\mathsf{sum} + m_{19}(u).\mathsf{sum}$*);* $m_{24}(u).\mathsf{card}$, $m_{28}(u).\mathsf{card}$ *et* $m_{32}(u).\mathsf{card}$ *(égaux à* $m_{20}(u).\mathsf{sum}$*);* $m_{25}(u).\mathsf{card}$, $m_{29}(u).\mathsf{card}$ *et* $m_{33}(u).\mathsf{card}$ *(égaux à* $m_{21}(u).\mathsf{sum}$*);* $m_{26}(u).\mathsf{card}$, $m_{30}(u).\mathsf{card}$ *et* $m_{34}(u).\mathsf{card}$ *(égaux à* $m_{22}(u).\mathsf{sum}$*); et* $m_{27}(u).\mathsf{card}$, $m_{31}(u).\mathsf{card}$ *et* $m_{35}(u).\mathsf{card}$ *(égaux à* $m_{23}(u).\mathsf{sum}$*).*

Il convient de souligner que les éléments des n-uplets m ne dépendent ni de l'ordre des unités, ni de l'ordre des places, ni de l'ordre des transitions.

Définition 4.22 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$*. La fonction* $\pi_{U[N]}$ *de la définition 4.20 est définie comme étant toute permutation* $\pi_U : U \to U$*, telle que :*

— $\forall u, u' \in U, (m\,(u) \preceq m\,(u')) \Rightarrow (\#\,(\pi_U(u)) \leq \#\,(\pi_U(u')))$*, avec* $\preceq$ *la relation désignant l'ordre lexicographique des n-uplets ;*
— $\forall u, u' \in U, (m\,(u) = m\,(u')) \wedge (\#\,(u) < \#\,(u')) \Rightarrow (\#\,(\pi_U(u)) < \#\,(\pi_U(u')))$.

Autrement dit, π_U *réalise un tri stable des unités de* U *par ordre croissant selon* $u \mapsto m\,(u)$.

4.6.2 Classement et tri des places

La fonction $\pi_{P[N]}$ de permutation des places, mentionnée au sein de la définition 4.20 est définie ci-dessous.

Définition 4.23 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$*. Pour toute place* p*, un 27-uplet* $n\,(p) \stackrel{\text{def}}{=} (n_1(p), \ldots, n_{27}(p))$ *est construit, constitué de nombres naturels, de valeurs de type* $\mathbb{D}$*, ou de vecteurs, calculées sur* N*, tel que :*

— $n_1(p) \stackrel{\text{def}}{=} m\,(\mathsf{unité}\,(p))$;
— $n_2(p) \stackrel{\text{def}}{=} \mathsf{distance1}\,(p)$;
— $n_3(p) \stackrel{\text{def}}{=} \mathsf{distance2}\,(p)$;
— $n_4(p) \stackrel{\text{def}}{=} \mathsf{distance3}\,(p)$;
— $n_5(p) \stackrel{\text{def}}{=} \mathsf{nb\_boucles}\,(p)$;
— $n_6(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_transitions\_d'entrée\_décroissantes}\,(p)$;
— $n_7(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_transitions\_d'entrée\_conservatrices}\,(p)$;
— $n_8(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_transitions\_d'entrée\_croissantes}\,(p)$;
— $n_9(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_transitions\_de\_sortie\_décroissantes}\,(p)$;
— $n_{10}(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_transitions\_de\_sortie\_conservatrices}\,(p)$;
— $n_{11}(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_transitions\_de\_sortie\_croissantes}\,(p)$;

— $n_{12}(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_places\_d'entrée\_pred}\,(p)$;
— $n_{13}(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_places\_de\_sortie\_pred}\,(p)$;
— $n_{14}(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_places\_d'entrée\_succ}\,(p)$;
— $n_{15}(p) \stackrel{\text{def}}{=} \mathsf{nb\_de\_places\_de\_sortie\_succ}\,(p)$;
— $n_{16}(p) \stackrel{\text{def}}{=} \mathsf{distance\_d'entrée\_pred}\,1\,(p)$;
— $n_{17}(p) \stackrel{\text{def}}{=} \mathsf{distance\_de\_sortie\_pred}\,1\,(p)$;
— $n_{18}(p) \stackrel{\text{def}}{=} \mathsf{distance\_d'entrée\_succ}\,1\,(p)$;
— $n_{19}(p) \stackrel{\text{def}}{=} \mathsf{distance\_de\_sortie\_succ}\,1\,(p)$;
— $n_{20}(p) \stackrel{\text{def}}{=} \mathsf{distance\_d'entrée\_pred}\,2\,(p)$;
— $n_{21}(p) \stackrel{\text{def}}{=} \mathsf{distance\_de\_sortie\_pred}\,2\,(p)$;
— $n_{22}(p) \stackrel{\text{def}}{=} \mathsf{distance\_d'entrée\_succ}\,2\,(p)$;
— $n_{23}(p) \stackrel{\text{def}}{=} \mathsf{distance\_de\_sortie\_succ}\,2\,(p)$;
— $n_{24}(p) \stackrel{\text{def}}{=} \mathsf{distance\_d'entrée\_pred}\,3\,(p)$;
— $n_{25}(p) \stackrel{\text{def}}{=} \mathsf{distance\_de\_sortie\_pred}\,3\,(p)$;
— $n_{26}(p) \stackrel{\text{def}}{=} \mathsf{distance\_d'entrée\_succ}\,3\,(p)$;
— $n_{27}(p) \stackrel{\text{def}}{=} \mathsf{distance\_de\_sortie\_succ}\,3\,(p)$.

Les composants suivants sont exclus de $n\,(p)$ *:* $n_{12}(p).\mathsf{card}$ *et* $n_{13}(p).\mathsf{card}$ *(puisque tous égaux à* $n_6(p)+n_7(p)+n_8(p)$*);* $n_{14}(p).\mathsf{card}$ *et* $n_{15}(p).\mathsf{card}$ *(égaux à* $n_9(p)+n_{10}(p)+n_{11}(p)$*);* $n_{16}(p).\mathsf{card}$*,* $n_{20}(p).\mathsf{card}$ *et* $n_{24}(p).\mathsf{card}$ *(égaux à* $n_{12}(p).\mathsf{sum}$*);* $n_{17}(p).\mathsf{card}$*,* $n_{21}(p).\mathsf{card}$ *et* $n_{25}(p).\mathsf{card}$ *(égaux à* $n_{13}(p).\mathsf{sum}$*);* $n_{18}(p).\mathsf{card}$*,* $n_{22}(p).\mathsf{card}$ *et* $n_{26}(p).\mathsf{card}$ *(égaux à* $n_{14}(p).\mathsf{sum}$*); et enfin* $n_{19}(p).\mathsf{card}$*,* $n_{23}(p).\mathsf{card}$ *et* $n_{27}(p).\mathsf{card}$ *(égaux à* $n_{15}(p).\mathsf{sum}$*).*

Il est important de préciser que les attributs des n-uplets n ne dépendent ni de l'ordre des unités, ni de l'ordre des places, ni de l'ordre des transitions du réseau.

Dans le format de fichier NUPN (voir la section 3.6), l'ensemble des places locales d'une unité doit former un intervalle, cette contrainte se traduisant pour toutes bijections des unités π_U et des places π_P sous la forme de l'équation suivante : $\forall p_1, p_2 \in P, (\pi_P\,(p_1) \leq \pi_P\,(p_2)) \Leftrightarrow (\pi_U\,(\mathsf{unité}\,(\pi_P\,(p_1))) \leq \pi_U\,(\mathsf{unité}\,(\pi_P\,(p_2))))$.

Puisque dans la définition 4.20, nous appliquons au préalable une permutation des unités, avant d'appliquer une permutation des places, nous ne pouvons pas utiliser (en l'état) n pour définir la fonction π_P. Nous construisons alors de nouveaux n-uplets $\overline{n}$ préservant l'ordre déjà appliqué sur les transitions.

Définition 4.24 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$*. Pour toute place* p*, un* 2*-uplet* $\overline{n}\,(p) \stackrel{\text{def}}{=} (\#\,(\mathsf{unité}\,(p)), n\,(p))$ *est construit.*

Le seul attribut de $\overline{n}$ dépendant de l'ordre des unités est $\overline{n}[1]$, et nous avons bien : pour toute paire de places p_1 et p_2, $(\pi_U\,(\mathsf{unité}\,(p_1)) \leq \pi_U\,(\mathsf{unité}\,(p_2))) \Rightarrow (\overline{n}\,(p_1) \leq \overline{n}\,(p_2))$. Notons par ailleurs que $(n\,(p_1) < n\,(p_2)) \Rightarrow (\overline{n}\,(p_1) < \overline{n}\,(p_2))$ et que $(\overline{n}\,(p_1) \leq \overline{n}\,(p_2)) \Rightarrow (n\,(p_1) \leq n\,(p_2))$.

Définition 4.25 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. *La fonction* $\pi_{P[N]}$ *de la définition 4.20 est définie comme étant toute permutation* $\pi_P : P \to P$, *telle que :*

- $\forall p, p' \in P, (\overline{n}(p) \preceq \overline{n}(p')) \Rightarrow (\#(\pi_P(p)) \leq \#(\pi_P(p')))$, *avec* $\preceq$ *la relation désignant l'ordre lexicographique des n-uplets ;*
- $\forall p, p' \in P, (\overline{n}(p) = \overline{n}(p')) \wedge (\#(p) < \#(p')) \Rightarrow (\#(\pi_P(p)) < \#(\pi_P(p')))$.

Ainsi définie, π_P *réalise un tri stable des places de* P *par ordre croissant selon* $p \mapsto \overline{n}(p)$.

4.6.3 Classement et tri des transitions

La fonction $\pi_{T[N]}$ de permutation des transitions, mentionnée au sein de la définition 4.20 est définie ci-dessous.

Définition 4.26 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. *Pour toute transition* t, *un 2-uplet* $o(t) \stackrel{\text{def}}{=} (o_1(t), o_2(t))$ *est construit, constitué de nombres naturels ou de valeurs de type* $\mathbb{D}$ *calculées sur* N, *tel que :*

- $o_1(t) \stackrel{\text{def}}{=} \vec{\mathcal{H}}(\{\!| \#p \mid p \in {}^\bullet t |\!\})$, *en remarquant que* $o_1(t).\mathsf{card} = |{}^\bullet t|$;
- $o_2(t) \stackrel{\text{def}}{=} \vec{\mathcal{H}}(\{\!| \#p \mid p \in t^\bullet |\!\})$, *en remarquant que* $o_2(t).\mathsf{card} = |t^\bullet|$.

Précisons que le classement et le tri des transitions repose entièrement sur l'ordre des places (et *a fortiori* des unités), qui ont déjà été permutées. D'où la taille réduite des n-uplets o.

Définition 4.27 *Soit un NUPN* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. *La fonction* $\pi_{T[N]}$ *de la définition 4.20 est définie comme étant toute permutation* $\pi_T : T \to T$, *telle que :*

- $\forall t, t' \in T, (o(t) \preceq o(t')) \Rightarrow (\#(\pi_T(t)) \leq \#(\pi_T(t')))$, *avec* $\preceq$ *la relation désignant l'ordre lexicographique des n-uplets ;*
- $\forall t, t' \in T, (o(t) = o(t')) \wedge (\#(t) < \#(t')) \Rightarrow (\#(\pi_T(t)) < \#(\pi_T(t')))$.

En d'autres termes, π_T *réalise un tri stable des transitions de* T *par ordre croissant selon* $t \mapsto o(t)$.

4.6.4 Unicité de la canonisation

Lorsqu'un réseau possède, indépendamment de toute permutation préalable de ses unités, places et transitions, une unique forme canonique, alors il est non-isomorphe avec tout réseau ayant une forme canonique distincte. La proposition suivante caractérise un ensemble de réseaux ayant une canonisation unique.

Proposition 4.6 *Soit* N *un NUPN tel que* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. *Si nous avons* $|\{m(u) \mid u \in U\}| = |U|$ *(voir la définition 4.21),* $|\{\overline{n}(p) \mid p \in P\}| = |P|$ *(voir la définition 4.24) et* $|\{o(t) \mid t \in T\}| = |T|$ *(voir la définition 4.26), alors il y a unicité de la canonisation du réseau* N *: pour toutes permutations* π_x *d'unités, de places ou de transitions,* $(\mathsf{can} \circ \overline{\pi_x})(N) = \mathsf{can}(N)$.

En pratique, nous n'avons pas besoin d'un résultat aussi fort pour montrer que deux réseaux

canonisés ne sont pas isomorphes : il suffit de comparer les n-uplets associés aux unités et aux places des deux réseaux canonisés, comme le montre la proposition ci-dessous.

Proposition 4.7 *Soient N et N' deux NUPN, tels que $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$, $N' = (P', T', F', M'_0, U', u'_0, \sqsubseteq', \mathsf{unité}')$, avec $\mathsf{sig}(N) = \mathsf{sig}(N')$ et $\mathsf{can}(N) \neq \mathsf{can}(N')$. Si nous avons $\{\!|\, n(p) \mid p \in P \,|\!\} \neq \{\!|\, n(p') \mid p' \in P' \,|\!\}$, alors les réseaux N et N' ne sont pas isomorphes.*

Cette proposition réalise en fait une seconde signature, en réutilisant les résultats de la canonisation à moindre coût. Notons l'absence des n-uplets m et o, qui sont subsumés par les informations contenues dans sig et n.

4.7 Réduction en termes d'isomorphisme de graphes

La troisième approche est vouée aux exemples complexes ne pouvant être prouvés isomorphes ou non-isomorphes à l'aide des signatures et des canonisations de NUPN. Elle exprime l'isomorphisme des NUPN en termes d'isomorphisme de graphes, un problème pour lequel des outils logiciels sont disponibles, notamment Bliss [JK07], Nishe [Gre09], Saucy [KSM12], Conauto [LCA13], Nauty [MP14] et Traces [MP14].

Le schéma de traduction proposé par la preuve de la proposition 4.4 permet certes d'exprimer succinctement le problème d'isomorphisme de NUPN en termes d'isomorphisme de graphes, mais n'est pas efficace, en particulier pour les NUPN possédant un arbre des unités $(U, \sqsupseteq)$ profond ou large, ce qui inclut notamment, mais pas exclusivement, les NUPN hiérarchiques et les NUPN triviaux.

Nous avons conçu cinq schémas de réduction différents du problème de l'isomorphisme de NUPN en isomorphisme de graphes, avec des boucles ou non, des arcs orientés ou non, des étiquettes ou non. Ces réductions produisent toutes, dans le pire des cas, des graphes de taille linéaire ou quadratique par rapport aux NUPN en entrée et sont de complexité équivalente d'un point de vue théorique. Nous présentons ci-après la traduction produisant les meilleurs résultats en pratique avec l'outil choisi.

Définition 4.28 *Soit $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un NUPN. La fonction G associe à N un graphe étiqueté $\mathsf{G}(N) \stackrel{\text{def}}{=} (V, E, \mathsf{c})$, où :*

1. *L'ensemble V des sommets est défini comme suit :*
   - *V contient un sommet v_0 ;*
   - *pour toute $p \in P$, V contient deux sommets notés v_p et v'_p ;*
   - *pour toute $t \in T$, V contient un sommet noté v_t ;*
   - *tous les sommets susmentionnés sont distincts, ainsi $|V| = 2\,|P| + |T| + 1$.*
2. *L'ensemble des arêtes E est défini comme suit :*
   - *pour toute $p \in P$, E possède une arête $\{v_p, v'_p\}$ exprimant que v_p et v'_p sont tous deux en lien avec la même place p ;*

— *pour toute* $p \in M_0$, *E possède une arête* $\{v_0, v'_p\}$ *exprimant que p est une place du marquage initial ;*

— *pour tout arc d'entrée* $(t,p) \in F \cap (P \times T)$, *E possède une arête* $\{v_p, v_t\}$ *le représentant ;*

— *pour tout arc de sortie* $(t,p) \in F \cap (T \times P)$, *E possède une arête* $\{v_t, v'_p\}$ *le représentant ;*

— *pour toute* $p \in \mathsf{places}(u_0)$, *E possède une arête* $\{v_0, v_p\}$ *exprimant que p est une place de l'unité racine, si tel est le cas,* u_0 *n'est pas vide ;*

— *pour toute paire* $(p_1, p_2) \in P^2$ *telle que* $\mathsf{unité}(p_1) \in (\mathsf{sous\text{-}unités} \circ \mathsf{unité})(p_2)$, *E possède une arête* $\{v_{p_1}, v_{p_2}\}$, *exprimant le fait que* p_1 *et* p_2 *sont les places locales d'unités directement en relation mère-fille.*

3. *La fonction d'étiquetage* c *est définie comme suit :*

— $\mathsf{c}(v_0) = 0$;

— *pour toute* $p \in P$, $\mathsf{c}(v_p) \stackrel{\text{def}}{=} 2 \times \#(n(p)) + 1$; — *voir la def. 4.23 pour* $p \mapsto n(p)$

— *pour toute* $p \in P$, $\mathsf{c}(v'_p) \stackrel{\text{def}}{=} \mathsf{c}(v_p) + 1$;

— *pour toute* $t \in T$, $\mathsf{c}(v_t) \stackrel{\text{def}}{=} \mathsf{max}(\{\mathsf{c}(v'_p) \mid p \in P\}) + 1$.

La figure 4.1 présente un NUPN (à gauche) et son graphe étiqueté (à droite) correspondant, produit par application de la fonction G (voir la définition 4.28).

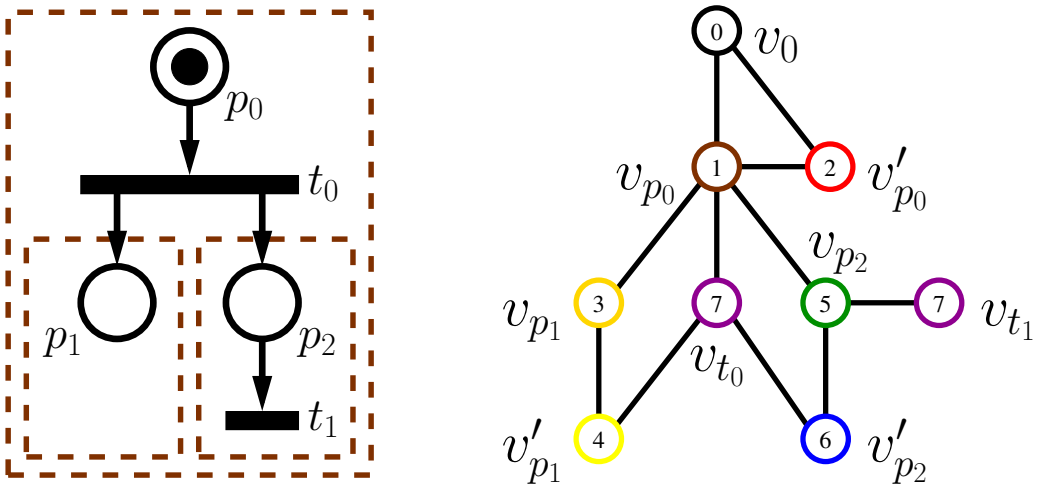

FIGURE 4.1 – Un NUPN N (à gauche) et son graphe $\mathsf{G}(N)$ correspondant (à droite)

La proposition suivante énonce quelques propriétés utiles de la fonction G.

Proposition 4.8 *Soit un NUPN N et son graphe correspondant* $\mathsf{G}(N)$. *Nous définissons quatre « types » de sommets* V_0, V_P, V'_P *et* V_T, *qui correspondent respectivement à* $\{v_0\}$, $\{v_p \mid p \in P\}$, $\{v'_p \mid p \in P\}$ *et* $\{v_t \mid t \in T\}$. *Le graphe possède les propriétés suivantes :*

(i) $\mathsf{G}(N)$ *ne possède pas de boucle et ses arcs ne sont pas orientés ;*

(ii) *le sommet appartenant à* V_0 *est unique et ne peut être relié qu'à des sommets de* $V_P \cup V'_P$ *;*

(iii) *les sommets* V_P *peuvent être reliés aux sommets de* $V = V_0 \cup V_P \cup V'_P \cup V_T$ *;*

(iv) *les sommets* V'_P *ne peuvent être reliés qu'à des sommets de* $V_0 \cup V_P \cup V_T$ *;*

(v) *les sommets* V_T *ne peuvent être reliés qu'à des sommets de* $V_P \cup V'_P$ *;*

(vi) *lorsque* N *est un NUPN trivial, aucune paire de sommets de* V_P *ne peuvent être reliés par une arête, le graphe* $\mathsf{G}(N)$ *est ainsi quadriparti ;*

(vii) *lorsque* N *est un NUPN trivial ayant une unité racine vide,* $\mathsf{G}(N)$ *ne contient aucune arête relative aux unités ;*

(viii) *l'étiquetage des sommets de* $\mathsf{G}(N)$ *est invariant des permutations des unités, des places et des transitions de* N *;*

(ix) *pour toute* $p \in P$ *et tout* $t \in T$*, nous avons* $0 = \mathsf{c}(v_0) < \mathsf{c}(v_p) < \mathsf{c}(v'_p) < \mathsf{c}(v_t)$ *;*

(x) *pour toutes* $p_1, p_2 \in P$ *telles que* $(\mathsf{profondeur} \circ \mathsf{unité})(p_1) < (\mathsf{profondeur} \circ \mathsf{unité})(p_2)$*, nous avons* $\mathsf{c}(v_{p_1}) < \mathsf{c}(v'_{p_1}) < \mathsf{c}(v_{p_2}) < \mathsf{c}(v'_{p_2})$ *(l'attribution initiale des étiquettes des sommets de* V_P *et* V'_P *se faisant par ordre lexicographique suivant* $p \mapsto n(p)$ *; elle se fait également par ordre lexicographique suivant* $\mathsf{profondeur} \circ \mathsf{unité}$*).*

Les graphes produits par la fonction G sont traités par des outils d'isomorphisme de graphes. Plutôt que de décider directement de l'isomorphisme de deux graphes en entrée, certains outils (notamment Bliss [JK07], Nishe [Gre09], Nauty [MP14] et Traces [MP14]) peuvent prendre en entrée un graphe, classer les sommets de celui-ci selon un ordre invariant de toute permutation préalables de ses sommets, afin de produire en sortie un graphe dit *canonique*. Conséquemment, déterminer si deux graphes sont isomorphes (ou non) revient à vérifier si leurs graphes canoniques sont identiques (ou distincts). Traiter les graphes individuellement plutôt que par paires permet d'obtenir un coût linéaire en fonction du nombre de graphes, plutôt que quadratique : nous optons pour la canonisation de graphes.

Définition 4.29 *Un outil de canonisation de graphes est une fonction* iso*, prenant en entrée un graphe étiqueté* $G = (V, E, \mathsf{c})$ *et produisant en sortie un graphe (non-étiqueté)* $\mathsf{iso}_G(G) = (V', E')$ *et une fonction bijective* $\mathsf{iso}_\pi(G) : V \to V'$*, tels que :*

— $E' = \{\{\mathsf{iso}_\pi(G)(v_1), \mathsf{iso}_\pi(G)(v_2)\} \mid \{v_1, v_2\} \in E\}$ *;*

— $\forall v_1, v_2 \in V, \mathsf{c}(v_1) < \mathsf{c}(v_2) \Rightarrow \mathsf{iso}_\pi(G)(v_1) < \mathsf{iso}_\pi(G)(v_2)$ *;*

— *deux graphes étiquetés sont isomorphes* ssi *ils possèdent la même image par les fonctions* $(V, E, \mathsf{c}) \mapsto \mathsf{iso}_{(V,E,\mathsf{c})}$ *et* $(V, E, \mathsf{c}) \mapsto c \circ (\mathsf{iso}_\pi^{-1}(V, E, \mathsf{c}))$*.*

Concrètement, un outil de canonisation de graphes fonctionne par raffinements successifs de la fonction d'étiquetage du graphe G donnée en entrée[7]. Il produit une fonction $\mathsf{iso}_\pi(G)$

7. Lorsque le graphe en entrée est non-étiqueté, il est converti en graphe étiqueté en attribuant à tous les sommets la même étiquette, c.-à-d. en définissant c comme étant une fonction constante.

bijective sur les sommets de G, préservant l'ordre des sommets de G selon ses étiquettes et invariante de toute permutation préalable des sommets de G. Cette fonction $\mathsf{iso}_\pi(G)$ est finalement utilisée pour produire le graphe « canonique » $\mathsf{iso}_G(G)$. La fonction d'étiquetage du graphe en entrée a ainsi deux rôles pour la procédure de la définition 4.29 : (i) elle permet de préserver un ordre existant sur les sommets ; (ii) elle permet (potentiellement) d'accélérer la procédure de canonisation : plus le co-domaine de la fonction d'étiquetage est grand, moins il reste de sommets à distinguer.

Notons que dans la définition 4.28, nous aurions également pu donner aux transitions des étiquettes distinctes, par exemple par l'intermédiaire d'une fonction $t \mapsto (|{}^\bullet t|, |t^\bullet|)$ [8]. Cependant, ces informations sont retrouvées par les outils d'isomorphisme de graphe en une seule itération, par l'intermédiaire des étiquettes des places : de ce fait, ajouter ces informations augmenterait la taille des graphes, sans gain de rapidité observable en pratique pour les outils.

Les deux propositions suivantes montrent que le problème de l'isomorphisme (resp. du non-isomorphisme) de NUPN revient à déterminer s'ils ont la même image (resp. des images distinctes) par la fonction composée $\mathsf{iso}_G \circ \mathsf{G}$.

Proposition 4.9 *Deux NUPN isomorphes ont la même image par la fonction* $\mathsf{iso}_G \circ \mathsf{G}$.

Preuve. Dans la mesure où la fonction $p \mapsto n(p)$ est invariante des permutations d'unités, des permutations de places et des permutations de transitions, la fonction G de la définition 4.28 l'est également, au nommage près des sommets de V. Ainsi, par construction, à deux NUPN N et N' isomorphes sont associés deux graphes $\mathsf{G}(N)$ et $\mathsf{G}(N')$ isomorphes : N et N' ont la même image par $\mathsf{iso}_G \circ \mathsf{G}$. □

Maintenant que l'approche est prouvée complète (elle préserve l'isomorphisme des NUPN en entrée), il reste à montrer qu'elle soit correcte (deux NUPN ayant la même image par $\mathsf{iso}_G \circ \mathsf{G}$ sont bien isomorphes).

Proposition 4.10 *Deux NUPN ayant la même image par la fonction composée* $\mathsf{iso}_G \circ \mathsf{G}$ *(issue des définitions 4.28 et 4.29) sont isomorphes.*

Preuve. Montrons qu'à partir de tout graphe $(\mathsf{iso}_G \circ \mathsf{G})(N) = (V, E)$, il est possible de reconstruire sans ambiguïté le NUPN N, au nommage des places, des transitions et des unités près. Sans restreindre la généralité, supposons que $V \cap \mathbb{N} = \varnothing$.

(i) D'après la définition 4.29, le graphe canonique issu de la composée $\mathsf{iso}_G \circ \mathsf{G}$ est sans étiquette, mais les sommets de ce graphe restent ordonnés selon l'ordre décrit par les étiquettes de la fonction G de la définition 4.28.

(ii) Le sommet v_0 existe et est unique : d'après (ix) de la proposition 4.8, il s'agit du plus petit élément de V, c.-à-d. de celui tel que $v_0 = \mathsf{min}(V)$.

(iii) Les sommets de $\{v_1 \in V \mid (v_2 \in V \setminus V_0) \wedge (\{v_1, v_2\} \in E) \Rightarrow (v_1 <_V v_2)\}$ appartiennent à V_P, d'après (ii …v) et (ix) de la proposition 4.8. De plus, cet ensemble est celui des

8. Ici, les n-uplets o de la définition 4.26 ne sont pas utilisables, car ils dépendent de l'ordre des places.

places de profondeur minimale, d'après (x) de la proposition 4.8.

(iv) Tout sommet $v_p \in V_P$ est relié à un unique sommet de V'_P, dont l'étiquette est strictement plus grande que celle de v_p ; plus précisément, il s'agit du sommet $v'_p \in V \mid v'_p = \mathsf{min}_V(\{v \in V \mid (\{v, v_p\} \in E) \wedge (v_p <_V v)\})$, d'après (iii) et (ix) de la proposition 4.8.

(v) Les sommets V_T ne sont reliés qu'à des sommets strictement plus petits, d'après (v) et (ix) de la proposition 4.8.

(vi) Les points (iv) et (v) de cette présente preuve permettent de déterminer si un sommet $v \in V_P \cup V_T$ appartient à V_P ou V_T.

(vii) Pour tout sommet de V, il est possible de reconnaître sans ambiguïté s'il est de type V_0, V_P, V'_P ou V_T : ceci s'effectue en partant des sommets du point (iii) de cette preuve, c.-à-d. des sommets de V_P de profondeur minimale, puis, en itérant sur les sommets de V_P et V'_P de profondeurs croissantes, jusqu'à saturation, c.-à-d. jusqu'à atteindre les places de profondeur maximale, en exploitant les points (iv) et (vi) de cette preuve.

(viii) L'ensemble $\{p \in V_P \mid \{v_0, v_p\} \in E\}$ est celui des places directement contenues dans l'unité racine.

Nous reconstruisons ainsi le NUPN $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$, sans ambiguïté, au nommage des places, des transitions et des unités près :

— soit $P \stackrel{\text{def}}{=} V_P$;

— soit $T \stackrel{\text{def}}{=} V_T$;

— soit $M_0 \stackrel{\text{def}}{=} \{p \in P \mid \{v_p, v_0\} \in E\}$;

— soit $F \stackrel{\text{def}}{=} \{(p,t) \in (P \times T) \mid \{v_p, v_t\} \in E\} \cup \{(t,p) \in (T \times P) \mid \{v'_p, v_t\} \in E\}$;

— $u_0 \stackrel{\text{def}}{=}$ *si* $(\{u_0\} \otimes P) \cap E \neq \varnothing$ *alors* $\mathsf{min}\,(\{p \in P \mid \{v_0, p\} \in E\})$ *sinon* 0 ;

— $\mathsf{unité}\,(p_1) \stackrel{\text{def}}{=} \mathsf{min}\,(\{p_2 \in P \mid \forall p_3 \in P, (\{p_1, p_3\} \in E) \Leftrightarrow (\{p_2, p_3\} \in E)\})$;

— $U \stackrel{\text{def}}{=} \{u_0\} \cup \{\mathsf{unité}\,(p) \mid p \in P\}$;

— $\sqsubseteq \stackrel{\text{def}}{=} \mathsf{f\_trans}\,(\{(\mathsf{unité}\,(p_1), \mathsf{unité}\,(p_2)) \mid (p_1, p_2 \in P) \wedge (\{p_1, p_2\} \in E)\} \cup (\{u_0\} \times U))$ où $\mathsf{f\_trans}$ est la fonction de calcul de la fermeture transitive d'un ensemble.

Ainsi, des NUPN ayant la même image par $\mathsf{iso}_G \circ \mathsf{G}$ sont identiques au nommage des places, des transitions et des unités près, c.-à-d. sont des NUPN isomorphes. □

Lorsque deux NUPN N et N' sont déclarés isomorphes par l'approche de cette section (c.-à-d. ils ont la même image par $\mathsf{iso}_G \circ \mathsf{G}$), nous voulons déterminer les bijections des places π_P, des transitions π_T et des unités π_U permettant de satisfaire la définition 4.7 et ainsi avoir $(\overline{\pi_T} \circ \overline{\pi_P} \circ \overline{\pi_U})\,(N) = N'$. Supposons que nous ayons déjà la connaissance des bijections des places π_P et des transitions π_T, mais pas de la bijection des unités π_U. La proposition suivante montre que nous pouvons retrouver π_U depuis π_P.

Proposition 4.11 *Soient deux NUPN isomorphes N et N' et soient trois bijections correspondantes de places $\pi_P : P \to P'$, de transitions $\pi_T : T \to T'$ et d'unités $\pi_U : U \to U'$. Alors $\pi_U = u \mapsto$ si $u = u_0$ alors u'_0 sinon $(\mathsf{unité}_{N'} \circ \pi_P \circ \mathsf{choix} \circ \mathsf{places}_N)(u)$.*

Preuve. Par définition, toute place $p \in P$ est place locale de l'unité $\mathsf{unité}_N(p) \in U$. La place p de N est associée à la place $\pi_P(p) \in P'$ de N'. Ainsi, l'unité $\mathsf{unité}_N(p)$ de N est associée à l'unité $\mathsf{unité}_{N'}(\pi_P(p))$ de N'. Or, toutes les unités non-racines de N et de N' ont au moins une place, et l'unité racine u_0 de N est, par définition, associée à l'unité racine u'_0 de N'. D'où le résultat. □

Lorsque deux NUPN sont prouvés isomorphes, les outils d'isomorphisme génèrent les bijections des sommets menant au graphe canonique. La proposition suivante montre qu'il est possible d'extraire de ces bijections sur les sommets les trois bijections des places, des transitions et des unités permettant de passer d'un NUPN à l'autre.

Proposition 4.12 *Soit deux NUPN N et N' isomorphes. Les fonctions $\mathsf{iso}_\pi(N)$ et $\mathsf{iso}_\pi(N')$ permettent de définir les bijections des places π_P, des transitions π_T et des unités π_U de N vers N'.*

Preuve. Soit N et N' deux NUPN isomorphes. La proposition 4.9 s'applique et les graphes $\mathsf{G}(N)$ et $\mathsf{G}(N')$ sont isomorphes. La fonction suivante est définie et bijective des sommets de $\mathsf{G}(N)$ vers ceux de $\mathsf{G}(N')$: $\pi_V \stackrel{\text{def}}{=} \mathsf{iso}_\pi^{-1}(N') \circ \mathsf{iso}_\pi(N)$. Cette fonction permet de définir la bijection des places $\pi_P \stackrel{\text{def}}{=} p \mapsto \mathsf{choix}(\{p' \in P' \mid v_{p'} = \pi_V(v_p)\})$ ainsi que la bijection des transitions $\pi_T \stackrel{\text{def}}{=} t \mapsto \mathsf{choix}(\{t' \in T' \mid v_{t'} = \pi_V(v_t)\})$. La bijection des unités π_U découle, quant à elle, de la proposition 4.11. □

4.8 Réduction en termes de validité booléenne

À l'instar de l'approche de la section 4.7, la quatrième approche est destinée aux exemples complexes ne pouvant être prouvés isomorphes ou non-isomorphes par les autres approches. Elle exprime l'isomorphisme d'une paire de NUPN par une formule logique, qui peut être évaluée à l'aide d'un résolveur de formules logiques du premier ordre (dit autrement, une formule SAT, qui est ensuite donnée à un résolveur SAT).

Nous définissons au préalable trois fonctions auxiliaires donnant, pour chaque unité, place ou transition de l'un des NUPN, un sur-ensemble d'unités, de places ou de transitions pouvant lui être associées dans l'autre réseau.

Définition 4.30 *Soient N et N' deux NUPN tels que $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ et $N' = (P', T', F', M'_0, U', u'_0, \sqsubseteq', \mathsf{unité}')$. Nous définissons assoc_U (resp. assoc_P, assoc_T) comme étant une fonction mettant chaque unité (resp. place, transition) de N en correspondance avec un sous-ensemble d'unités (resp. de places, de transitions) de N' :*

- $\mathsf{assoc}_U(u) \stackrel{\text{def}}{=} \{u' \in U' \mid \mathsf{profondeur}(u) = \mathsf{profondeur}(u')\}$;
- $\mathsf{assoc}_P(p) \stackrel{\text{def}}{=}$ si $p \in M_0$ alors M'_0 sinon $P' \setminus M'_0$;

— $\mathsf{assoc}_T(t) \stackrel{\text{def}}{=} \{t' \in T' \mid (|{}^\bullet t| = |{}^\bullet t'|) \wedge (|t^\bullet| = |t'^\bullet|)\}$.

Nous définissons ensuite une formule logique du premier ordre utilisant le fragment de logique QF_IDL [9] : ce fragment correspond à la logique *quantifier-free integer-difference*, qui apporte les variables entières et les contraintes arithmétiques portant sur la différence entre deux variables.

La formule logique $\varphi(N, N')$ décrite dans ce fragment est définie *ssi* les NUPN N et N' ont les mêmes nombres de places et de transitions, et vraie *ssi* N et N' sont isomorphes.

Définition 4.31 *Soient N et N' deux NUPN tels que $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$, $N' = (P', T', F', M'_0, U', u'_0, \sqsubseteq', \mathsf{unité}')$, $|P| = |P'|$ et $|T| = |T'|$. Nous définissons $\varphi(N, N')$ comme étant une formule logique ayant $|P| + |T|$ variables :*

— *pour toute $p \in P$, $\varphi(N, N')$ contient une variable x_p ayant pour domaine P' ; toutes ensembles, ces variables x_p représentent une correspondance $P \to P'$;*

— *pour toute $t \in T$, $\varphi(N, N')$ contient une variable x_t ayant pour domaine T' ; toutes ensembles, ces variables x_t représentent une correspondance $T \to T'$.*

Les contraintes logiques exprimées par $\varphi(N, N')$ sont les suivantes :

— *pour chaque $p \in P$, une contrainte $x_p \in \mathsf{assoc}_P(p)$ limite les valeurs possibles de x_p ;*

— *pour chaque $t \in T$, une contrainte $x_t \in \mathsf{assoc}_T(t)$ limite les valeurs possibles de x_t ;*

— *pour chaque paire $p_1, p_2 \in P$ telle que $p_1 <_P p_2$, une contrainte $x_{p_1} \neq x_{p_2}$ permet d'imposer que la correspondance susmentionnée « $P \to P'$ » représente une bijection ;*

— *pour chaque paire $t_1, t_2 \in T$ telle que $t_1 <_T t_2$, une contrainte $x_{t_1} \neq x_{t_2}$ permet d'imposer que la correspondance susmentionnée « $T \to T'$ » représente une bijection ;*

— *pour chaque $t \in T$ et chaque $t' \in \mathsf{assoc}_T(t)$, une contrainte*

$$x_t = t' \Rightarrow \bigwedge_{p \in {}^\bullet t} \left(x_p \in {}^\bullet t' \cap \mathsf{assoc}_P(p)\right) \wedge \bigwedge_{p \in t^\bullet} \left(x_p \in t'^\bullet \cap \mathsf{assoc}_P(p)\right)$$

exprime le fait que les arcs de F correspondent aux arcs de F' ;

— *pour chaque $u \in U$ et chaque $u' \in \mathsf{assoc}_U(u)$, une contrainte*

$$\bigvee_{p_1 \in \mathsf{places}(u)} \left(x_{p_1} \in \mathsf{assoc}_P(p_1) \cap \mathsf{places}(u')\right) \Rightarrow \bigwedge_{p_2 \in \mathsf{places}(u)} \left(x_{p_2} \in \mathsf{assoc}_P(p_2) \cap \mathsf{places}(u')\right)$$

exprime le fait que les places locales de u correspondent aux places locales de l'unité u' correspondante ;

— *nous imposons à l'arbre des unités $(U, \sqsupseteq)$ de coïncider avec $(U', \sqsupseteq')$: plus précisément, pour chaque $u_1 \in U$, $u_2 \in \{u \in U \mid u_1 \in \mathsf{sous\text{-}unités}(u) \wedge |\mathsf{assoc}_U(u)| > 1\}$, $u'_1 \in \mathsf{assoc}_U(u_1)$ et $u'_2 \in \{u' \in \mathsf{assoc}_U(u_2) \mid u'_1 \in \mathsf{sous\text{-}unités}(u')\}$, une contrainte*

$$\left(x_{(\mathsf{choix} \circ \mathsf{places})(u_1)} \in \mathsf{places}(u'_1)\right) \Rightarrow \left(x_{(\mathsf{choix} \circ \mathsf{places})(u_2)} \in \mathsf{places}(u'_2)\right)$$

9. smtlib.cs.uiowa.edu/logics.shtml

exprime le fait qu'à deux unités imbriquées $u_1 \sqsubset u_2$, correspondent (par l'intermédiaire de leurs places locales respectives) deux unités U' imbriquées, c.-à-d. telles que $u'_1 \sqsubset' u'_2$.

Notons que $\varphi(N, N')$ ne contient aucune variable relative aux unités, autrement dit, de variable x_u avec $u \in U$. En effet, chaque unité u non-vide est représentée dans $\varphi(N, N')$ par des contraintes sur les variables x_p de ses places locales $p \in \mathsf{places}(u)$. Or, dans le formalisme NUPN, seule l'unité racine peut être vide, c.-à-d. ne posséder aucune place locale. Comme cette unité racine est unique, nous savons que u_0 dans N doit être associée à u'_0 dans N'.

Les deux propositions suivantes mettent en évidence que deux réseaux N et N' ayant les mêmes nombres de places et de transitions sont isomorphes *ssi* la formule $\varphi(N, N')$ est vraie. De plus, si tel est le cas, l'affectation des variables nous permet de construire les trois bijections de places, transitions et unités permettant de passer de N à N'.

Proposition 4.13 *Si N et N' sont des NUPN isomorphes, $\varphi(N, N')$ est définie et vraie.*

Preuve. Soit $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un NUPN. Par construction, la formule $\varphi(N, N)$ est définie et vraie avec l'affectation des variables $\{x_y = y \mid \forall y \in P \cup T\}$. En outre, les définitions des fonctions assoc_U, assoc_P, assoc_T et φ sont invariantes par toutes permutations d'unités π_U, de places π_P et de transitions π_T. Ainsi, pour tout NUPN $N' = (\overline{\pi_T} \circ \overline{\pi_P} \circ \overline{\pi_U})(N)$, la formule $\varphi(N, N')$ est définie (N et N' ont les mêmes nombres de places et de transitions) et est vraie avec l'affectation des variables suivante $\{x_p = \pi_P(p) \mid \forall p \in P\} \cup \{x_t = \pi_T(t) \mid \forall t \in T\}$. □

Proposition 4.14 *Soient deux NUPN N et N' ayant les mêmes nombres de places et de transitions, autrement dit, dont la formule $\varphi(N, N')$ est définie. S'il existe une affectation des variables $\{x_p, \ldots, x_t, \ldots\}$ telle que $\varphi(N, N')$ soit vraie, alors N et N' sont isomorphes.*

Preuve. Soient N et N' deux NUPN, tels que la formule $\varphi(N, N')$ soit définie, avec $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ et $N' = (P', T', F', M'_0, U', u'_0, \sqsubseteq', \mathsf{unité}')$. Supposons qu'il existe une affectation des variables $\{x_p, \ldots, x_t, \ldots\}$ telle que la formule $\varphi(N, N')$ soit vraie. Cette affectation permet de définir les fonctions $\pi_P : p \mapsto x_p$ et $\pi_T : t \mapsto x_t$, qui sont bijectives d'après la définition 4.31. Définissons également, par application de la proposition 4.11, la fonction $\pi_U : u \mapsto$ *si* $u = u_0$ *alors* u'_0 *sinon* $(\mathsf{unité} \circ \pi_P \circ \mathsf{choix} \circ \mathsf{places})(u)$. Comme la formule $\varphi(N, N')$ est invariante par toute permutation de places de P situées dans la même unité de U et que π_P est bijective, la fonction π_U est bijective. Soit $N'' = (\overline{\pi_T} \circ \overline{\pi_P} \circ \overline{\pi_U})(N) = (P'', T'', F'', M''_0, U'', u''_0, \sqsubseteq'', \mathsf{unité}'')$. Montrons que $N' = N''$.

Nous avons, d'après les définitions 4.30 et 4.31, une correspondance entre les arcs F' et F''. D'après la définition 4.30, nous avons $\forall p \in P, p \in M_0 \Leftrightarrow \pi_P(p) \in M''_0$, c.-à-d. $M'_0 = M''_0$. Ainsi, les réseaux de Petri sous-jacents de N' et N'' sont identiques.

D'après la définition 4.30, deux unités associées par la fonction π_U possèdent la même profondeur. Par définition, $u''_0 = \pi_U(u_0) = u'_0$. Le formalisme NUPN impose que toutes les autres unités de U' et U'' possèdent au moins une place locale. Or, d'après la définition 4.31,

si une place $p \in P$ d'une unité $u \in U$ est associée à une place locale d'une unité $u' \in U'$, alors toutes les places locales de u deviennent des places locales de u', ainsi, les unités sont préservées par la permutation des places et les fonctions $\mathsf{unité}'$ et $\mathsf{unité}''$ sont identiques. Toujours d'après la définition 4.31, l'imbrication des unités est préservée, ainsi, les arbres $(U', \sqsubseteq')$ et $(U'', \sqsubseteq'')$ sont identiques. Donc $N' = N''$ et, conséquemment, la validité d'une formule $\varphi(N, N')$ implique que N et N' sont isomorphes. □

Quoique les trois prédicats de la définition 4.30 soient suffisants pour la définition 4.31 et garantissent les propositions 4.13 et 4.14, nous présentons ci-dessous un ensemble de règles permettant d'affiner ces trois prédicats, pour diminuer (parfois drastiquement) la taille des formules $\varphi(N, N')$ générées et accélérer le résolveur. Ces règles sont appliquées jusqu'à saturation, avant la génération de chaque formule.

Définition 4.32 *Soient N et N' deux NUPN tels que $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$, $N' = (P', T', F', M'_0, U', u'_0, \sqsubseteq', \mathsf{unité}')$. Les co-domaines des trois fonctions assoc_U, assoc_P et assoc_T de la définition 4.30 peuvent être restreints par l'application jusqu'à saturation des règles suivantes. Pour toutes $u \in U$, $p \in P$, $t \in T$, $u' \in U'$, $p' \in P'$, $t' \in T'$ et pour toute fonction $f \in \{x \to {}^\bullet x \setminus x^\bullet, x \to {}^\bullet x \cap x^\bullet, x \to x^\bullet \setminus {}^\bullet x\}$:*

- $\mathsf{assoc}_U(u) = \{u' \in \mathsf{assoc}_U(u) \mid m(u) = m(u')\}$; — *voir la def. 4.21 pour* $u \mapsto m(u)$
- $\mathsf{assoc}_P(p) = \{p' \in \mathsf{assoc}_P(p) \mid n(p) = n(p')\}$; — *voir la def. 4.23 pour* $p \mapsto n(p)$
- *si* $\mathsf{assoc}_U(u) = \{u'\}$ *alors* $(\forall u_2 \in U \setminus \{u\})\ u' \notin \mathsf{assoc}_U(u_2)$;
- *si* $\mathsf{assoc}_P(p) = \{p'\}$ *alors* $(\forall p_2 \in P \setminus \{p\})\ p' \notin \mathsf{assoc}_P(p_2)$;
- *si* $\mathsf{assoc}_U(t) = \{t'\}$ *alors* $(\forall t_2 \in T \setminus \{t\})\ t' \notin \mathsf{assoc}_T(t_2)$;
- *si* $\mathsf{assoc}_U(u) = \{u'\} \wedge (p \in \mathsf{places}(u))$ *alors* $\mathsf{assoc}_P(p) \subseteq \mathsf{unité}(u')$;
- *si* $\mathsf{assoc}_U(u) = \{u'\}$ *alors* $\bigcup_{u_2 \in \mathsf{sous\text{-}unités}^*(u)} \mathsf{assoc}_U(u_2) \subseteq \mathsf{sous\text{-}unités}^*(u')$;
- *si* $\mathsf{assoc}_P(p) = \{p'\} \wedge (t \in f(p)) \wedge (t' \notin f(p'))$ *alors* $t' \notin \mathsf{assoc}_T(t)$;
- *si* $\mathsf{assoc}_T(t) = \{t'\} \wedge (p \in f(t)) \wedge (p' \notin f(t'))$ *alors* $p' \notin \mathsf{assoc}_P(p)$;
- *si* $\mathsf{assoc}_P(p) = \{p'\} \wedge (p \in \mathsf{unité}(u)) \wedge (p' \in \mathsf{unité}(u'))$ *alors* $\mathsf{assoc}_U(u) = \{u'\}$;
- *si* $p' \in \mathsf{assoc}_P(p) \wedge \left(f(p') \notin \bigcup_{t \in f(p)} \mathsf{assoc}_T(t)\right)$ *alors* $p' \notin \mathsf{assoc}_P(p)$;
- *si* $t' \in \mathsf{assoc}_T(t) \wedge \left(f(t') \notin \bigcup_{p \in f(t)} \mathsf{assoc}_P(p)\right)$ *alors* $t' \notin \mathsf{assoc}_T(t)$;
- *si* $p' \in \mathsf{assoc}_P(p) \wedge (|f(p')| \neq |f(p)|)$ *alors* $p' \notin \mathsf{assoc}_P(p)$;
- *si* $t' \in \mathsf{assoc}_T(t) \wedge (|f(t')| \neq |f(t)|)$ *alors* $t' \notin \mathsf{assoc}_T(t)$.

4.9 Implantation logicielle et formats de fichiers

Nous avons implanté ces idées dans une chaîne d'outils combinant : des outils spécialement développés, des outils existants au centre INRIA de l'Université Grenoble Alpes que nous avons étendus et des outils tiers que nous avons réutilisés tels quels.

Notre chaîne d'outils prend en entrée une collection de réseaux au format NUPN ou au format normalisé PNML, puis détermine ceux qui sont isomorphes et ceux qui ne le sont pas. Elle implante toutes les approches présentées dans les sections 4.5 à 4.8.

Les sous-sections suivantes expliquent comment sont implantées les différentes approches de notre chaîne d'outils, en suivant leur ordre d'exécution, qui correspond à une complexité croissante.

4.9.1 Déduplication de fichiers

La première façon de rechercher des doublons dans une collection de réseaux (et la plus simple) est de repérer les fichiers identiques. Parmi les nombreux outils qui existent à cet effet, nous avons choisi JDUPES [10], un outil en ligne de commande fiable et efficace.

De toute évidence, cette approche est très limitée. Par exemple, l'insertion d'un espace supplémentaire dans un fichier PNML peut empêcher la détection de deux réseaux isomorphes par cette méthode. Il y a donc un compromis évident entre la flexibilité d'un format de réseau et la capacité de détecter les doublons en utilisant une simple comparaison de fichiers. À cet égard, le format NUPN est préférable au format PNML parce qu'il est plus contraignant : les places, les transitions et les unités sont identifiées à l'aide de nombres naturels formant des intervalles au lieu d'identificateurs alphanumériques ; les lexèmes doivent être séparés par un seul espace ; les lignes vides sont interdites, ainsi que les espaces avant la fin des lignes, etc. Pour cette raison, notre chaîne d'outils utilise le format NUPN plutôt que PNML.

En pratique, la traduction de PNML en NUPN, suivie d'une invocation de JDUPES, est souvent suffisante pour détecter les doublons n'impliquant aucune permutation.

4.9.2 Pré-canonisation de réseaux

Bien que le format NUPN soit plus contraignant que le format PNML, il offre tout de même un certain niveau de flexibilité permettant d'exprimer un même réseau sous différentes formes, même en l'absence de toute permutation de places, de transitions ou d'unités. Pour résoudre ce problème, l'outil NUPN_INFO (voir la section 3.7) a été étendu avec une option « `-precanonical-nupn` » prenant en entrée un réseau au format NUPN et produisant en sortie le même réseau dans lequel : (i) toutes les places (resp. transitions, unités) sont renumérotées à partir de zéro ; (ii) toutes les listes de places (resp. transitions, unités) sont triées par ordre croissant ; (iii) toutes les étiquettes de places (resp. transitions, unités) sont supprimées ; (iv) enfin, toutes les directives sont supprimées.

Après avoir mis tous les réseaux sous cette forme pré-canonique, JDUPES est (de nouveau) invoqué pour détecter les fichiers doublons.

4.9.3 Signatures de réseaux

L'outil CÆSAR.BDD (voir la section 3.7) a été étendu en lui ajoutant deux options « `-signature` » et « `-signature-multiple` » permettant de calculer la signature (telle

10. github.com/jbruchon/jdupes

que définie dans la section 4.5) d'un réseau donné au format « `.nupn` ». L'option « `-signature-multiple` » exploite également le contenu des directives « `!multiple_arcs` » et « `!multiple_initial_tokens` » (voir la section 3.6) pour mieux différencier les réseaux non-ordinaires issus de la traduction du format PNML vers NUPN. Écrit en C, le calcul est rapide (0,12 seconde par réseau en moyenne) et réussit toujours.

Un script shell identifie les classes de réseaux ayant les mêmes signatures.

4.9.4 Canonisation de réseaux

L'outil CÆSAR.BDD a été étendu avec trois nouvelles options (« `-unit-order` », « `-place-order` », et « `-transition-order` ») afin de calculer, pour un réseau donné au format NUPN, tous les n-uplets des fonctions $u \mapsto m(u)$, $p \mapsto \overline{n}(p)$ et $t \mapsto o(t)$ définies dans la section 4.6. L'outil CÆSAR.BDD invoque ensuite la commande Unix SORT afin de trier ces n-uplets selon un ordre lexicographique croissant et effectue, uniquement pour l'option « `-unit-order` », des calculs supplémentaires susceptibles de permettre la distinction d'unités ayant la même image par la fonction m, en considérant également les valeurs attribuées aux unités mères, filles et sœurs dans l'arbre des unités.

Un script Awk est ensuite invoqué pour transformer ces résultats en permutations d'unités, de places ou de transitions. Le réseau en entrée et les trois permutations sont ensuite donnés à l'outil NUPN_INFO (avec les options « `-unit-permute` », « `-place-permute` », et « `-transition-permute` »), lequel produit en sortie un réseau canonisé. Une nouvelle option « `-canonical-nupn` », automatisant toutes ces étapes, y compris les trois invocations de CÆSAR.BDD, a été ajoutée à NUPN_INFO. Écrite en langages C et Awk, la canonisation est assez rapide (2,3 secondes par réseau en moyenne).

Enfin, JDUPES est invoqué pour détecter les doublons de fichiers parmi l'ensemble des réseaux canonisés. La proposition 4.7 est quant à elle implantée par un script Awk, lisant les sorties de CÆSAR.BDD et produisant un nouvel ensemble de signatures.

4.9.5 Isomorphisme de graphes

Parmi les différents outils dédiés à l'isomorphisme de graphes, nous avons sélectionné le logiciel TRACES[11] [MP14] en raison de sa notoriété en termes d'efficacité. Cet outil, qui fournit à la fois une API (en langage C) et une interface en ligne de commande, est capable de mettre un graphe sous forme canonique (deux graphes sont isomorphes ssi leurs graphes canoniques respectifs sont identiques) et aussi de décider si deux graphes sont isomorphes.

Nous avons développé un script Python permettant de convertir un réseau au format NUPN en un graphe étiqueté (selon l'algorithme donné à la section 4.7), puis d'invoquer TRACES pour mettre ce graphe sous forme canonique. Pour certains très grands graphes, l'exécution de TRACES peut échouer ou s'arrêter au bout d'un certain temps. Pour traiter une collection,

11. `pallini.di.uniroma1.it` (version 2.8.6, avec tous les correctifs publiés le 13 janvier 2023).

notre script est d'abord invoqué sur chaque réseau de la collection. Enfin, l'outil JDUPES est utilisé pour détecter les doublons parmi les graphes canonisés.

4.9.6 Formules logiques

Pour implanter notre approche pour l'isomorphisme de réseaux fondée sur les formules logiques, nous avons développé un script Python qui prend en entrée deux réseaux au format NUPN ayant les mêmes nombres de places et de transitions, et qui produit une formule employant le fragment de logique QF_IDL du format SMT-LIB 2.6 [12] (selon l'algorithme donné à la section 4.8).

Le traducteur invoque en interne CÆSAR.BDD sur les deux réseaux afin d'obtenir les n-uplets apportés par les fonctions $u \mapsto m(u)$ et $p \mapsto n(p)$ définies dans la section 4.6, afin d'optimiser la formule de la définition 4.30, par application des règles de la définition 4.32.

La formule étant au format standard SMT-LIB, tout résolveur de formules logiques du premier ordre capable de traiter le fragment QF_IDL peut être utilisé. Pour nos expériences, nous avons choisi le résolveur Z3 [dMB08] [13], qui communique avec notre script par deux tubes Unix. Si le résolveur répond que la formule est fausse, les deux réseaux ne sont pas isomorphes. Sinon, s'il répond que la formule est vraie, les deux réseaux sont isomorphes. Il peut également renvoyer un résultat indéfini (abandon ou dépassement de délai), dans la mesure où la complexité de la résolution de formules logiques est exponentielle dans le pire des cas.

4.10 Mesures pour l'évaluation des résultats

4.10.1 Classement en trois catégories

Nous proposons de classer les modèles selon trois catégories mutuellement exclusives, permettant de définir des statistiques simples et pertinentes pour caractériser des jeux de tests, et d'évaluer nos approches pour la détection des NUPN isomorphes ou non-isomorphes :

— les modèles *uniques*, qui forment le plus grand ensemble de modèles que nous pourrions extraire du jeu de tests considéré, tout en ayant la garantie qu'ils soient deux à deux non-isomorphes ;

— les modèles *doublons* qui peuvent être effacés du jeu de tests sans craindre de perdre de modèle unique ;

— les modèles restants, dits *inconnus*.

Ainsi, tout modèle du jeu de tests considéré doit appartenir à l'une de ces trois catégories. En outre, dès lors qu'un modèle est étiqueté unique ou doublon, il doit le rester, indépendamment

12. smtlib.cs.uiowa.edu
13. github.com/Z3Prover/z3

des approches appliquées ultérieurement : seul un modèle inconnu doit pouvoir changer de catégorie, afin de devenir un modèle unique ou doublon.

4.10.2 Représentation de la relation calculée

D'après la proposition 4.1, l'isomorphisme est une relation d'équivalence. Dans une telle relation, une classe d'équivalence est un ensemble maximal (au sens de l'inclusion) de modèles mutuellement isomorphes. Cette relation étant réflexive, symétrique et transitive, l'ensemble des classes d'équivalence forme une partition des modèles du jeu de tests. Nous introduisons alors la fonction $N \mapsto [N]$, associant à tout modèle N sa classe d'équivalence associée $[N]$. Concrètement, lorsque $[N_1] = [N_2]$, les modèles N_1 et N_2 sont isomorphes.

Les classes d'équivalences permettent de décrire les paires de NUPN isomorphes, mais elles ne permettent pas de représenter les paires de NUPN non-isomorphes. Nous englobons alors les classes d'équivalence au sein d'éléments intitulés *ensembles potentiels*, qui forment une partition de l'ensemble des classes d'équivalences de la collection. Deux NUPN localisés dans des ensembles potentiels distincts sont non-isomorphes.

Toute classe d'équivalence possède un NUPN la représentant. En pratique, nous choisissons le NUPN ayant le plus petit contenu suivant l'ordre lexicographique, ou en cas de non-unicité, celui ayant le plus petit nom de fichier, selon l'ordre lexicographique.

Par définition, il existe un NUPN unique par ensemble potentiel. Au sein de chaque ensemble potentiel, nous choisissons systématiquement le plus petit NUPN, en reprenant les critères du paragraphe ci-dessus. Ce NUPN unique est ainsi nécessairement le représentant d'une classe d'équivalence.

L'ensemble des NUPN doublons est l'ensemble des NUPN qui ne sont pas leur propre représentant au sein de leur classe d'équivalence : de ce fait, il y en a autant que le nombre total de NUPN de la collection, moins le nombre de classes d'équivalence.

Les modèles restants, c'est-à-dire les modèles qui sont représentants d'une classe d'équivalence, mais qui ne sont pas les plus petits au sein de leur ensemble potentiel, sont les modèles inconnus : il y en a autant qu'il y a de nombre de classes d'équivalence, moins le nombre de classes potentielles.

4.10.3 Opérations sur la représentation

Initialement, nous ne savons pas si deux NUPN distincts sont isomorphes, ou non, mais la relation étant réflexive, nous savons que tout NUPN est isomorphe à lui-même. De ce fait, la partition initiale d'une collection E est constituée par un unique ensemble potentiel, contenant toutes les classes d'équivalence, lesquelles sont toutes une taille une : $\{\{[N_i] \mid N_i \in E\}\}$.

La figure 4.2 illustre cette partition initiale sur une collection (fictive) de 12 NUPN, où les classes d'équivalence sont représentées en pointillées et sont englobées dans le même ensemble potentiel. Les NUPN sont représentés en violet, blanc ou vert, selon le fait qu'ils

sont classés respectivement uniques, doublons ou inconnus.

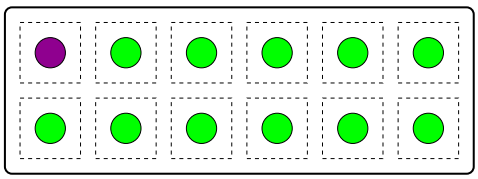

FIGURE 4.2 – Représentation initiale d'une collection de 12 NUPN

Cette partition fait par la suite l'objet d'itérations successives, permettant de découvrir davantage de NUPN uniques ou doublons.

Lorsque deux NUPN N_1 et N_2 appartenant à des classes d'équivalences distinctes sont détectés isomorphes par l'une des approches, nous savons, par transitivité, que $[N_1] \cup [N_2]$ constitue un ensemble de modèles mutuellement isomorphes. Si tel est le cas, les classes d'équivalence $[N_1]$ et $[N_2]$, qui appartiennent au même ensemble potentiel (dans le cas contraire, N_1 et N_2 ne pourraient être isomorphes), sont alors fusionnées, formant ainsi une nouvelle classe d'équivalence $[N_1] \cup [N_2]$. La figure 4.3 illustre, à partir de la figure 4.2, une telle opération.

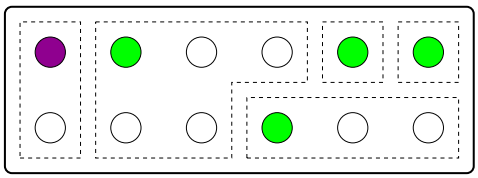

FIGURE 4.3 – Fusion de classes d'équivalence

Les approches visant la détection de modèles non-isomorphes partitionnent individuellement chacun des ensembles potentiels, pour les remplacer ensuite par leurs éléments, lesquels devenant alors les nouveaux ensembles potentiels. Par exemple, si l'une des approches est parvenue à déterminer, pour l'ensemble potentiel $\{[N_1], \ldots, [N_k]\}$, que toutes les paires de NUPN de $\{N_1, \ldots, N_i\} \times \{N_{i+1}, \ldots, N_k\}$ sont mutuellement non-isomorphes, alors la partition de la collection $\{.^{a}_{?}.\} \cup \{\{[N_1], \ldots, [N_k]\}\}$ devient $\{.^{a}_{?}.\} \cup \{\{[N_1], \ldots, [N_i]\}, \{[N_{i+1}], \ldots, [N_k]\}\}$. La figure 4.4 illustre, à partir de la figure 4.3, une telle opération, qui produit deux ensembles potentiels.

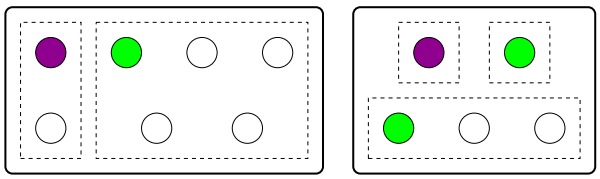

FIGURE 4.4 – Raffinement de la partition en ensembles potentiels

Cette classification des modèles en trois catégories possède également des avantages pour une implantation efficace, en effectuant un minimum de calculs sur les modèles, tout en restant simple. Elle autorise en effet trois raccourcis, que nous avons implantés dans notre chaîne d'outils :

— Dès lors qu'un modèle est reconnu doublon, il n'est plus comparé aux autres modèles du jeu de tests par la chaîne d'outils : il est remplacé par son représentant, au sein de sa classe d'équivalence.

— Lorsqu'un ensemble potentiel ne contient plus de modèles inconnus, autrement dit, qu'il ne contient plus qu'une seule classe d'équivalence, nous savons que le représentant de cette classe d'équivalence est non-isomorphe avec tous les réseaux hors de sa classe d'équivalence. Conséquemment, la chaîne d'outils n'a plus besoin de traiter aucun des modèles de cet ensemble potentiel.

— Lorsqu'il n'existe plus de modèle marqué inconnu, la relation d'isomorphisme est totalement connue, pour l'intégralité du jeu de tests : la chaîne d'outils peut s'arrêter.

4.11 Résultats expérimentaux

4.11.1 Présentation des collections

Nous avons évalué notre chaîne d'outils sur quatre collections de réseaux synthétisées dans la table 4.1 ci-dessous :

— *collection 1* : 244 réseaux de Petri (sous la forme de NUPN triviaux) mis à disposition par l'Université de Zielona Góra [14] ;

— *collection 2* : 1 387 réseaux de Petri obtenus en prenant tous les modèles (non colorés) utilisés pour l'édition 2022 du MCC [15] ; 44 % de ces réseaux ont des places initiales avec plus d'un jeton ou des arcs avec une multiplicité strictement supérieure à un, tandis que 50 % de ces réseaux sont des NUPN non-triviaux et unit-saufs ;

— *collection 3* : il s'agit de la collection décrite dans la section 3.8, constituée de 16 200 NUPN unit-saufs d'origines diverses et contenant très peu de doublons ;

— *collection 4* : 241 657 NUPN unit-saufs (occupant 135 Go d'espace disque) produits au centre INRIA de l'Université Grenoble Alpes en étendant la collection 3 avec des NUPN additionnels, puis en appliquant de nombreuses permutations à tous ces réseaux, ce qui a permis de constituer un ensemble de 840 838 NUPN ; l'outil Jdupes a été ensuite utilisé pour retrancher tous les fichiers identiques : en conséquence, la collection 4 contient de nombreux NUPN isomorphes difficiles à détecter, car ayant des fichiers distincts.

14. www.hippo.iie.uz.zgora.pl (consulté le 23 janvier 2023)
15. mcc.lip6.fr/2022

	collection 1		collection 2	
	moyenne	maximum	moyenne	maximum
# places	15,4	200	2 801,5	537 708
# transitions	11,8	51	10 798	1 070 836
# arcs	34,2	400	83 384,7	25 615 632
# unités	—	—	1 970	537 709
hauteur	—	—	15,4	2 891
largeur	—	—	1 959,1	537 708

	collection 3		collection 4	
	moyenne	maximum	moyenne	maximum
# places	345,8	131 216	740,8	131 216
# transitions	7 998,1	16 967 720	15 645	16 967 720
# arcs	71 217,9	146 528 584	113 102,9	146 528 584
# unités	123,4	78 644	270,4	78 644
hauteur	4,3	2 891	6,3	2 891
largeur	117,6	78 643	259,9	78 643

TABLE 4.1 – Statistiques numériques sur les quatre collections de réseaux

4.11.2 Choix d'une métrique

Après chaque étape de la chaîne d'outils, nous donnons les trois chiffres définies dans la section 4.10 : (i) *doublons* est le pourcentage de réseaux pouvant être supprimés, car ils ont été trouvés isomorphes à d'autres réseaux qui seront conservés dans la collection ; (ii) *uniques* est le pourcentage de réseaux déterminés uniques dans la collection après avoir supprimé tous les doublons ; (iii) enfin, *inconnus* est le pourcentage restant de réseaux dont le statut n'est pas encore déterminé. Le principal résultat est que presque tous les réseaux dupliqués ou uniques des quatre collections sont découverts par notre chaîne d'outils : elle a été concluante pour 99 % à 100 % de chaque collection.

La relation d'isomorphisme de réseaux étant une relation binaire, nous aurions pu, afin de pouvoir mesurer les résultats obtenus, opter (naïvement) pour trois pourcentages sur les paires de réseaux : celui de paires que nous savons isomorphes, celui de paires que nous savons non-isomorphes et celui de paires restant encore inconnues. Mais bien que conceptuellement très simple, nous ne pensons pas qu'une telle métrique serait pertinente, au moins pour les trois raisons suivantes :

— Cette notion de nombre de paires isomorphes et de paires non-isomorphes découvertes est intrinsèquement ambiguë : une fois avoir appliqué l'une des approches de la chaîne d'outils, devrions-nous, avant de compter le nombre de paires découvertes par cette dernière, appliquer une phase de fermeture transitive ? Si tel est le cas, cela voudrait dire que l'on « récompenserait » différemment la découverte de paires de réseaux isomorphes, en fonction des paires pré-existantes, lesquelles dépendent des approches

préalablement appliquées ; sinon, cela voudrait dire que nous devrions appliquer les approches sur toutes les paires inconnues (alors qu'elles sont originellement en nombre quadratique), sans tenir compte du fait que l'isomorphisme est une relation d'équivalence (voir la proposition 4.1), ce qui représenterait un gaspillage de ressources évident.

— Le nombre de paires isomorphes à détecter pour un modèle donné est linéaire en fonction de la taille de sa classe d'équivalence, alors que le nombre total de paires à déterminer pour tout modèle est linéaire en fonction de la taille du jeu de tests : il serait donc périlleux d'interpréter des statistiques obtenues à partir de ces chiffres.

— Ces pourcentages abstraits de paires ne permettent pas une réponse convaincante aux problématiques réelles d'un utilisateur, telles que celles exposées dans la section 4.1 : il peut vouloir constituer, pour les besoin d'une compétition, un sous-ensemble de modèles en étant certain de ne pas introduire de doublon, ceci afin de garantir une bonne diversité ; il peut également souhaiter constituer la plus grande collection possible, en éliminant les doublons connus, mais sans perdre de modèle de grande taille, qui pourrait éventuellement être traité à l'avenir ; enfin, il peut simplement vouloir évaluer la diversité d'une collection déjà constituée et utilisée.

4.11.3 Résultats obtenus et discussions

Pour traiter ces quatre collections, nous avons utilisé des serveurs de la grille de calcul Grid'5000, équipés de processeurs Intel Xeon Gold 5220 (2,2 GHz), de 96 Go de RAM et la distribution Linux Debian 11. La totalité des fichiers était stockée sur un serveur NFS partagé.

Les résultats de l'application de notre chaîne d'outils à ces quatre collections sont présentés dans la table 4.2 ci-dessous.

De plus amples remarques peuvent être formulées :

— La simple application de Jdupes permet de détecter 10 fichiers doublons dans la collection 1.

— La pré-canonisation a détecté 54 018 doublons dans la collection 4 ; la pré-canonisation n'a pas été appliquée à la collection 2, afin de préserver les directives du format de fichier NUPN (voir la section 3.6) donnant des informations sur les arcs multiples (« `!multiple_arcs` ») et les multiples jetons initiaux (« `!multiple_initial_tokens` »).

— Les signatures ont identifié beaucoup de réseaux uniques dans les collections 1 à 3 (représentant 92,21 % à 99,06 % des modèles de ces collections), mais n'ont pas eu le même effet sur la collection 4 (6,49 %), dans laquelle chaque réseau comporte au moins un autre modèle isomorphe.

— La procédure de canonisation de NUPN s'est révélée particulièrement efficace sur les quatre collections, que ce soit pour identifier les réseaux uniques ou les doublons :

	collection 1			collection 2		
	doublons	uniques	inconnus	doublons	uniques	inconnus
fichiers identiques	4,10 %	0,41 %	95,49 %	0	0,07 %	99,93 %
pré-canonisation	=	=	=	—	—	—
signatures	=	92,62 %	3,28 %	=	99,13 %	0,87 %
canonisation	5,74 %	93,03 %	1,23 %	0,58 %	99,42 %	0
isomorph. graphes	6,97 %	=	0	—	—	—
formules logiques	—	—	—	—	—	—
après canonisation	*6,97 %*	=	*0*	—	—	—

	collection 3			collection 4		
	doublons	uniques	inconnus	doublons	uniques	inconnus
fichiers identiques	0	0,01 %	99,99 %	0	0,00 %	100,0 %
pré-canonisation	0,17 %	=	99,83 %	22,35 %	=	77,65 %
signatures	=	96,24 %	3,59 %	=	6,49 %	71,16 %
canonisation	2,26 %	97,10 %	0,64 %	79,44 %	7,58 %	12,98 %
isomorph. graphes	2,78 %	97,22 %	0,01 %	90,05 %	9,09 %	0,86 %
formules logiques	=	=	=	90,06 %	=	0,85 %
après canonisation	*2,76 %*	*97,21 %*	*0,03 %*	*84,11 %*	*7,58 %*	*8,31 %*

Table 4.2 – Résultats obtenus par notre chaîne d'outils sur les quatre collections de réseaux

elle permet en particulier de déterminer intégralement la collection 2.

— Le recours à l'isomorphisme de graphes a permis d'élucider la majeure partie des réseaux dont le statut était resté inconnu après la canonisation : elle a permis de déterminer entièrement la collection 1, ne laissant plus qu'un seul modèle inconnu dans la collection 3 (il s'agit d'un modèle ayant 3 404 places et 189 196 transitions, trop grand pour être traité par les outils d'isomorphisme de graphes et les résolveurs de formules logiques).

— L'application de l'approche reposant sur les formules logiques après l'approche basée sur l'isomorphisme des graphes a diminué le nombre de modèles inconnus de 25 de la collection 4, mais n'a apporté aucun nouveau résultat sur la collection 3 (comme indiqué par le signe "=" dans la table 4.2) ; de plus, lorsqu'elle était appliquée immédiatement après l'étape de canonisation, la résolution de formules logiques était globalement moins efficace que l'isomorphisme de graphes, comme le montre la dernière ligne de la table 4.2 (en italique). Ceci explique pourquoi nous n'avons pas essayé de formuler le problème d'isomorphisme de graphes en termes de formules logiques : les outils dédiés à l'isomorphisme de graphes semblent mieux adaptés.

— Fait intéressant, notre chaîne d'outils a détecté 17 doublons dans la collection 1 et huit doublons au sein de la collection 2 ; pour cette dernière, un doublon est certain et sept sont très probables (leurs réseaux correspondants ne sont pas ordinaires et saufs).

4.12 Validation des résultats

Toutes les étapes de cette chaîne d'outils ont été méticuleusement validées en utilisant diverses techniques, dont les grandes lignes sont expliquées ci-dessous.

4.12.1 Signatures de réseaux

Pour vérifier l'exactitude des signatures, nous avons développé un script Python qui applique des (dizaines de millions de) permutations aléatoires de places, de transitions et d'unités à un NUPN donné et qui vérifie que les signatures des réseaux originaux et permutés soient bien identiques.

4.12.2 Canonisation de réseaux

Afin d'augmenter la confiance dans notre implémentation de la canonisation, nous avons vérifié : (i) que la canonisation est idempotente, c.-à-d. que deux invocations successives de Nupn_Info avec son option « `-canonical-nupn` » produisent le même résultat que celui résultant d'une seule invocation ; de plus, pour chaque réseau, nous avons calculé des dizaines de millions de bijections aléatoires pour nous assurer (ii) que les fonctions $u \mapsto m(u)$ demeurent invariantes pour toute bijection d'unités, de places et de transitions ; (iii) que les fonctions $p \mapsto \overline{n}(p)$ demeurent invariantes pour toute bijection de places et de transitions ; (iv) et enfin que les fonctions $t \mapsto o(t)$ demeurent invariantes pour toute bijection de transitions.

4.12.3 Isomorphisme de graphes

Pour tout graphe sur lequel l'outil d'isomorphisme de graphes a pu calculer une fonction sur ses sommets, nous avions vérifié quatre conditions nécessaires pour que cette fonction soit correcte : (i) la fonction est bien une bijection des sommets du graphe ; (ii) le sommet v_0 reste le premier sommet du graphe (c.-à-d. celui ayant la plus petite étiquette) ; (iii) à chaque sommet de V_P (resp. de V'_P, de V_T) est associé un sommet de V_P (resp. de V'_P, de V_T) ; (iv) enfin, pour toute paire de places p_1 et p_2, si le sommet $v_{p_1} \in V_P$ est associé au sommet $v_{p_2} \in V_P$, alors $v'_{p_1} \in V'_P$ est associé au sommet $v'_{p_2} \in V'_P$.

Proposition 4.15 *Soient deux NUPN isomorphes N et N' et soient leurs bijections de places $\pi_P : P \to P'$ et d'unités $\pi_U : U \to U'$ correspondantes. Soit $f_T \stackrel{\text{def}}{=} t_1 \mapsto |\{t_2 \in T \mid ({}^\bullet t_1 = {}^\bullet t_2) \wedge (t_1{}^\bullet = t_2{}^\bullet) \wedge (\#_T(t_1) \leq \#_T(t_2))\}|$ la fonction auxiliaire donnant à chaque transition $t \in T$ son rang, selon l'ordre $\#_T$, parmi toutes les transitions de T ayant les mêmes places d'entrée et de sortie. Soit $\pi'_T \stackrel{\text{def}}{=} t \mapsto \mathsf{choix}(\{t' \in T' \mid ({}^\bullet t' = \{\pi_P(p) \mid p \in {}^\bullet t\}) \wedge (t'{}^\bullet = \{\pi_P(p) \mid p \in t^\bullet\}) \wedge (f_T(t) = f_{T'}(t'))\})$ une bijection des transitions de T vers celles de T'. Alors, la permutation des places, transitions et unités de N suivant π_P, π'_T et π_U donne N'.*

Preuve. Par définition de l'isomorphisme de NUPN, toute fonction associant les transitions

T de N aux transitions T' de N' doit respecter les deux contraintes suivantes : (i) elle doit être une bijection $T \to T'$; (ii) elle doit associer à toute transition $t \in T$ de N une transition $t' \in T'$ de N' telle que : ${}^\bullet t' = \{\pi_P(p) \mid p \in {}^\bullet t\}$ et $t'^\bullet = \{\pi_P(p) \mid p \in t^\bullet\}$. Ceci est le cas de la fonction π'_t construite. □

Pour garantir l'absence de faux positifs, lorsque deux graphes $\mathsf{G}(N)$ et $\mathsf{G}(N')$ ont la même forme canonique $G'_N = G'_{N'}$, nous vérifions qu'ils ont été issus de deux réseaux isomorphes N et N', en générant à partir des permutations de sommets $\pi_v : \mathsf{G}(N) \to G'_N$ et $\pi_{v'} : \mathsf{G}(N') \to G'_{N'}$ la bijection des places de N vers les places de N', puis, à partir de cette dernière et au moyen des proposition 4.11 et 4.15, les bijections des unités et des transitions (avec le codage des graphes de la définition 4.28, il aurait été possible d'extraire la bijection des transitions depuis les permutations des sommets, à l'instar de ce qui a été fait pour les places, cependant, cette solution n'a pas été retenue car elle n'était pas réalisable avec tous les schémas de réduction vers les graphes expérimentés). Enfin, nous vérifions à l'aide de Nupn_Info que N soit bien identique à N', après permutation de ses unités, places et transitions.

Bien qu'un seul schéma de réduction soit décrit dans la section 4.7 (il s'agit du plus efficace en pratique), nous avons aussi validé les quatre autres schémas.

4.12.4 Formules logiques

Nous nous sommes d'abord assuré que, pour tout NUPN N, la formule $\varphi(N, N)$ est déclarée valide par le résolveur, quand celui-ci n'abandonne pas et ne dépasse pas le délai imparti.

Comme pour l'approche fondée sur l'isomorphisme de graphes, lorsqu'une formule $\varphi(N, N')$ est déclarée valide, pour garantir l'absence de faux positifs, nous vérifions à l'aide de Nupn_Info et au moyen des propositions 4.11 et 4.15 que N soit bien identique à N' après permutation de ses unités, places et transitions selon l'affectation des variables de places fournie par le résolveur (comme pour l'approche reposant sur l'isomorphisme de graphes, certains codages expérimentés pour les formules logiques ne permettaient pas l'extraction de la bijection des transitions depuis l'affectation des variables, d'où l'utilisation de la proposition 4.15).

4.12.5 Vérifications croisées

Nous avons réalisé des vérifications croisées de nos approches, en nous assurant que :

- les NUPN ayant des signatures différentes ont des formes canoniques distinctes ;
- les NUPN ayant des signatures différentes ont des graphes non-isomorphes ;
- les NUPN ayant des formes canoniques identiques engendrent des graphes isomorphes ;
- deux NUPN ayant des formes canoniques identiques engendrent une formule φ valide, lorsque le résolveur n'abandonne pas et ne dépasse pas le délai imparti. ;

— deux NUPN ayant des graphes isomorphes engendrent une formule φ valide, lorsque le résolveur n'abandonne pas et ne dépasse pas le délai imparti ;

— deux NUPN ayant des graphes non-isomorphes engendrent une formule φ non-valide, lorsque le résolveur n'abandonne pas et ne dépasse pas le délai imparti.

4.13 Bilan et perspectives

Pour le problème concret de la recherche de modèles doublons dans de grandes collections de réseaux de Petri ou de NUPN, nous avons conçu quatre approches complémentaires permettant de détecter des réseaux isomorphes : la signature de réseaux (sur-approximation), la canonisation de réseaux (sous-approximation), la réduction à un problème d'isomorphisme de graphes et la réduction à un problème de validité de formule logique du premier ordre.

Ces approches, qui découlent de l'examen minutieux de plusieurs dizaines de milliers de réseaux concrets, ont été entièrement mises en œuvre dans une chaîne d'outils efficace, dont les étapes successives sont classées suivant un ordre croissant de complexité. Pour traiter de grandes collections de réseaux, les calculs peuvent facilement être répartis sur des grilles d'ordinateurs, dans la mesure où la majeure partie des étapes porte sur des réseaux individuels ou, dans le cas de l'approche basée sur la logique, sur des paires de réseaux à comparer. Seule la détection de fichiers identiques ne se prête pas facilement à la parallélisation, mais n'a pas provoqué de goulots d'étranglement, car l'outil Jdupes que nous avons retenu est suffisamment rapide.

Cette chaîne d'outils a été expérimentée sur quatre collections de réseaux, la plus grande étant constituée de plus de 241 000 NUPN d'origines diverses, de tailles variées et possédant de nombreux modèles isomorphes. Sur ces quatre collections, notre chaîne d'outils a obtenu des taux de réussite allant de 99 % à 100 %. En particulier, elle a mis en évidence que la collection de réseaux du Model Checking Contest n'est pas exempte de doublons.

Ce travail pourrait être poursuivi dans plusieurs directions, dont voici les principales :

— Nous pourrions essayer de restreindre le nombre de composants figurant dans les signatures et dans les n-uplets de la canonisation afin de ne conserver que les composants les plus pertinents en pratique.

— Nous pourrions renforcer les signatures en ajoutant de nouveaux champs : dans la mesure où l'isomorphisme de réseaux implique l'équivalence forte de leurs marquages accessibles, certaines propriétés sur ces marquages et invariantes par permutation des places, transitions et unités pourraient être utilisées, par exemple en adoptant certaines notions et algorithmes des chapitres 5 et 6.

— Actuellement, les attributs pour les approches de signature et de canonisation sont des champs de taille fixe, pouvant être de type $\mathbb{N}$, $\mathbb{D}$, ou des n-uplets ; afin d'augmenter la précision des signatures et des canonisations, il serait possible d'opter pour des

champs de taille variable (par exemple en utilisant des arbres [16]), au détriment certes de la lisibilité de ces champs.

— D'après les propositions 4.11 et 4.15, il serait possible de générer une bijection des places, et à partir de celle-ci, de dériver automatiquement les bijections des unités et des transitions : ceci permettrait d'obtenir des formes canoniques plus précises (le tri des transitions deviendrait exact) et plus rapidement (il existe actuellement un recoupement concernant certains attributs portant sur les places et d'autres portant sur les unités).

— Pour les réseaux ayant un statut inconnu malgré l'exécution de toutes les approches actuelles de la chaîne d'outils, une bisimulation des marquages accessibles (à profondeur d'exploration bornée) permettrait de raffiner le tri des places, des transitions et des unités et, ainsi, de détecter de nouveaux réseaux uniques ou doublons.

— Nous pourrions étendre les approches proposées pour prendre en charge des classes plus étendues de réseaux, tels que les réseaux de Petri non-ordinaires ou ayant un marquage initial non-sauf ; pour le moment, ceux-ci sont traités à l'aide de sur-approximations, mais notre chaîne d'outils pourrait aisément être rendue exacte pour ces réseaux, par exemple en étiquetant les arcs non-ordinaires et les places initiales non-ordinaires.

— De même, il pourrait être utile de considérer l'isomorphisme des réseaux colorés, mais il s'agit d'un problème plus difficile.

— Les solutions proposées aux sections 4.5, 4.6 et (éventuellement) 4.8 pourraient aider à résoudre plus efficacement le problème de l'isomorphisme pour certaines classes de graphes, possédant des formes similaires à celles des réseaux de Petri ou des NUPN.

— D'autres schémas de réduction vers des formules logiques pourraient être expérimentés, en utilisant éventuellement un fragment de logique différent.

— Les graphes et formules générés par les approches d'isomorphisme de graphes et de formules logiques pourraient être réutilisés comme jeux de tests dans les compétitions de logiciels.

16. En notant que ces derniers peuvent être canonisés de manière exacte en temps polynomial.

Troisième partie

Analyse

Chapitre 5

Détermination des places mortes et des transitions mortes

Ce chapitre se concentre sur deux problèmes, qui caractérisent les parties d'un réseau qui ne sont jamais actives : la recherche de toutes les places qui ne peuvent jamais être marquées (problème des *places mortes*) et la recherche de toutes les transitions qui ne peuvent jamais être tirées (problème des *transitions mortes*). Leur détection est un outil pertinent pour la simplification de réseaux de Petri complexes, en particulier ceux générés automatiquement à partir de formalismes de plus haut niveau, tels que les algèbres de processus. Le présent chapitre propose divers algorithmes pour déterminer simultanément les places mortes et les transitions mortes.

5.1 Motivations

En pratique, ces deux problèmes, qui correspondent dans les réseaux de Petri au problème de la détection de code mort, sont pertinents pour plusieurs raisons.

Premièrement, ces problèmes représentent de manière concrète la détection de *code mort* pour les réseaux de Petri. Le problème de la détection de code mort a une importance majeure dans l'ingénierie logicielle, puisqu'il est conventionnellement admis que le code mort nuit à la lisibilité, à la maintenance et aux performances des logiciels, ce qui conduit la plupart des processus de développement logiciel à recommander la détection et la suppression pro-active du code mort. À ce sujet, deux langages de programmation pour les automates industriels, se basant sur une représentation graphique et une sémantique inspirée des réseaux de Petri, le *Grafcet* [IEC13a] et les *diagrammes de fonctions séquentiels* [IEC13b], interdisent les branches « non-accessibles », autrement dit, les réseaux de Petri contenant des places ou transitions mortes.

Deuxièmement, dans le contexte de la vérification de modèles, les places mortes augmentent la taille nécessaire à la représentation des marquages accessibles ; de plus, les places mortes,

comme les transitions mortes, augmentent les coûts de stockage et d'accès à la relation de transition (autrement dit, de la fonction donnant, à partir d'un ensemble de marquages, les marquages résultant du franchissement des transitions du réseau). De ce fait, les places et transitions mortes conduisent à une augmentation potentielle des coûts de vérification, à la fois en mémoire et en temps.

En particulier, ces places et transitions mortes apparaissent naturellement lors de la traduction automatisée de modèles écrits dans des langages de haut-niveau, tels que les algèbres de processus, en réseaux de Petri. Par exemple, le réseau de Petri généré depuis le terme « A; A; A || A; A; A » comporte 3×3 transitions synchronisées « A » ; l'élimination des transitions mortes permet de réduire ce nombre à trois. Le gain potentiel est accru lorsque des données sont échangées : le réseau résultant de « A!1; A!2; A!3 || A?x; A?y; A?z » présente toujours 3×3 transitions, mais l'élimination des six transitions mortes permet également à un compilateur de détecter que les variables x, y et z sont en réalité les constantes 1, 2 et 3. Ceci permet ainsi d'ôter trois variables du vecteur d'état, ce qui représente un gain important en mémoire. La boîte à outils CADP implante cette technique d'optimisation lors de la compilation de spécifications LOTOS en réseaux de Petri interprétés et l'exploite avec succès afin de pouvoir traiter des modèles industriels ou académiques de (plus) grande taille.

Lorsque de multiples propriétés doivent être évaluées sur le même modèle, les surcoûts engendrés par les places et les transitions mortes sont cumulés, ce qui rend d'autant plus importantes la détection et l'élimination efficace de telles places et transitions pour les modèles de grande taille. Ceci est le cas pour les utilisateurs industriels, qui ont besoin, en règle générale, d'évaluer (au moins) plusieurs dizaines de formules en logique temporelle, sur des systèmes complexes.

Troisièmement, de nombreuses propriétés globales sur les réseaux de Petri peuvent basculer de vraies à fausses (ou inversement) en ajoutant ou supprimant une seule place ou transition morte. Pour un utilisateur, l'interprétation pertinente des résultats de ces propriétés devient alors délicate. Pour un outil de vérification, certaines étapes d'analyse ou de transformation de réseaux exploitant des propriétés « souhaitables » des modèles (telles que l'appartenance à la classe des réseaux *à choix libres*) pourraient devenir moins efficaces, fournir des résultats moins précis, voire même se retrouver inapplicables.

Pour toutes ces raisons, on doit être en mesure de détecter efficacement, et d'éliminer les places et transitions mortes des réseaux de Petri, afin de ne considérer ou de ne préserver que la partie véritablement « utile » de ces réseaux.

La suite de ce chapitre s'articule ainsi. La section 5.2 introduit formellement les problèmes des places mortes et des transitions mortes. La section 5.3 dresse un inventaire de l'état de l'art et explique en quoi les vérificateurs de modèles conventionnels ne sont pas adaptés pour résoudre ce type de problèmes. La section 5.4 analyse la complexité théorique de ces problèmes. La section 5.5 explique pourquoi il est nécessaire d'avoir des solutions dites « incomplètes » car pouvant présenter des valeurs inconnues.

Les sections 5.6 à 5.10 décrivent ensuite des algorithmes dédiés, qui sont implantés dans des outils logiciels spécialisés, présentés dans la section 5.11. Les sections 5.12 et 5.13 font le point sur les performances de la solution proposée, une fois appliquée à des ensembles complets de modèles. La section 5.14 aborde les démarches entreprises pour la validation des résultats.

Finalement, la section 5.15 formule des observations conclusives et évoque les perspectives ouvertes.

5.2 Énoncé des problèmes

La définition suivante présente la notion de place morte au sein d'un réseau de Petri, en utilisant le prédicat $\Re$ de couverture de marquages de la définition 2.21.

Définition 5.1 *Soit* (P, T, F, M_0) *un réseau de Petri. Une place* p *est* morte ssi *il n'existe pas de marquage accessible ayant un jeton dans* p*, autrement dit* ssi $\neg\Re(\{p\})$.

Le problème de la détermination des places mortes est alors introduit comme suit.

Définition 5.2 *Soit un réseau de réseau de Petri* (P, T, F, M_0) *ordinaire et sauf. Le* problème des places mortes *consiste en la détection de toutes les places mortes du réseau.*

La définition suivante présente la notion de transition morte au sein d'un réseau de Petri.

Définition 5.3 *Soit* (P, T, F, M_0) *un réseau de Petri. Une transition* t *est* morte ssi *il n'existe pas de marquage accessible où* t *est active, autrement dit* ssi $\neg\Re({}^\bullet t)$.

Nous introduisons le problème de la détermination des transitions mortes, comme pour celui des places mortes.

Définition 5.4 *Soit un réseau de réseau de Petri* (P, T, F, M_0) *ordinaire et sauf. Le* problème des transitions mortes *consiste en la détection de toutes ses transitions mortes.*

5.3 Travaux voisins

5.3.1 Définitions alternatives des places mortes

Une généralisation de la définition 5.1 est donnée dans [DE95, def. 4.16], où une place p est considérée « morte dans un marquage M » s'il n'existe pas de marquage accessible depuis M présentant un jeton dans p. Notre définition ne couvre que le cas du marquage initial M_0, et ne considère que les réseaux saufs. Cependant, puisque toute place p morte dans M_0 l'est également dans tout marquage accessible, notre définition est suffisamment générale pour répondre aux motivations de la section 5.1.

Nous éviterons d'employer le terme *place vivante*, qui est usuellement défini comme suit : une place p est vivante *ssi*, pour tout marquage M accessible (depuis le marquage initial M_0), il existe un marquage M' accessible depuis M et ayant un jeton dans p. Ainsi, toute

place vivante n'est pas morte, mais une place non-morte n'est pas nécessairement vivante.

5.3.2 Définitions alternatives des transitions mortes

Une généralisation de la définition 5.3 est donnée dans [DE95, def. 4.16], où une transition t est dite morte dans un marquage M si la transition t est inactive (c.-à-d. non-tirable) dans tout marquage accessible depuis M.

Dans [Mur89, sect. IV.C], il existe quatre niveaux de vivacité pour les transitions, allant de la plus forte, *L1-vivante*, à la plus faible, *L4-vivante*; autrement dit *L1-vivante* $\Rightarrow$ *L2-vivante* $\Rightarrow$ *L3-vivante* $\Rightarrow$ *L4-vivante*. De plus, [Mur89, sect. IV.C] abrège les termes *L1-vivante* et *L4-vivante* en respectivement *vivante* et *quasi-vivante*. Bien que les termes *L4-vivante* et *quasi-vivante* correspondent à la négation de *morte* et pourraient ainsi être employés, nous n'utilisons que le niveau de vivacité 4, ce qui fait que cette lourde surcharge du terme vivante, source possible de confusions, n'est ni nécessaire, ni souhaitable. En définitive, nous n'utiliserons pour les transitions que les termes « mortes » et « non-mortes », de manière analogue aux places, qui sont « mortes » ou « non-mortes ».

Il convient de remarquer que le problème de la définition 5.3 est différent du problème de l'*absence d'interblocage*, qui a été abondamment traité, contrairement au problème des transitions mortes. Formellement, un réseau possède une transition morte *ssi* $\exists t \in T \mid \forall M_0 \xrightarrow{*} M, {}^\bullet t \not\subseteq M$, tandis qu'un réseau présente un interblocage *ssi* $\exists M_0 \xrightarrow{*} M \mid \forall t \in T, {}^\bullet t \not\subseteq M$: il y a permutation des objets au sein des quantificateurs logiques « $\exists$ » et « $\forall$ ». En dépit de la « dualité » de leurs formulations, ces deux problèmes restent bien distincts : il existe des réseaux présentant un interblocage, mais sans transition morte, tout comme il existe des réseaux présentant une transition morte, mais sans interblocage.

Enfin, nous pouvons noter que le problème des transitions mortes généralise celui de la *quasi-vivacité*, qui est un prédicat booléen évalué à $\top$ *ssi* le vecteur des transitions mortes est le vecteur nul. Cette généralisation se justifie d'un point de vue pratique : déterminer qu'un réseau de grande taille n'est pas quasi-vivant est insuffisant si l'on ne peut savoir quel est l'ensemble exact de transitions mortes, par exemple, afin de les éliminer par la suite.

5.3.3 Lien avec la logique temporelle

Une voie possible pour résoudre ces deux problèmes est de les transcrire en logique temporelle (par exemple, en formules CTL ou LTL), puis de soumettre ces formules à un outil de vérification de modèles (ou en anglais, « model checker ») pour les réseaux de Petri.

Bien qu'attrayante, une telle approche présente néanmoins des inconvénients d'ordre pratique :

— Le nombre de formules en logique temporelle requises pour chaque problème est linéaire en fonction de la taille du réseau de Petri — respectivement $|P|$ et $|T|$. Ce nombre est en général trop grand pour être réalisé manuellement, il faudrait donc développer des outils *ad hoc* produisant ces formules (donnant en sortie, soit

un fichier unique de taille colossale, soit un très grand nombre de petits fichiers), appelant un outil de vérification de modèles sur ces formules, et finalement, collectant et agrégeant les résultats de ces appels.

— Appeler répétitivement un outil de vérification de modèles pour évaluer des centaines à plusieurs milliers de formules sur le même réseau de Petri est inefficace. À chaque appel, une formule est lue et analysée syntaxiquement, sa correction est ensuite vérifiée, puis elle est traduite d'une syntaxe concrète vers une représentation interne abstraite : la plupart de ces étapes sont redondantes, puisque toutes ces formules sont similaires et correctes par construction et sont évaluées sur le même réseau.

Pour ces raisons, il nous paraît plus opportun d'exprimer ces deux problèmes à un niveau supérieur, en augmentant les outils traitant les réseaux de Petri d'options dédiées (par exemple, « `-dead-places` » et « `-dead-transitions` »). La présence de ces options faciliterait non seulement la tâche des utilisateurs finaux, mais elle permettrait aux développeurs d'outils de concevoir et implanter des approches dédiées, plus efficaces.

Pour un outil reposant sur une logique temporelle, ces options rendraient possible une augmentation sensible des performances : (i) de toutes les logiques temporelles acceptées par l'outil, ces options pourraient sélectionner la plus appropriée pour produire les formules, éventuellement selon le contexte ; (ii) elles pourraient directement générer les formules dans leur représentation abstraite, ce qui éliminerait à la fois le coût de l'écriture de fichiers intermédiaires, puis de leur analyse syntaxique, tout en donnant plus de souplesse aux développeurs d'outils ; (iii) le coût (cumulatif) de la vérification syntaxique des formules pourrait être évité, puisqu'elles seraient correctes par construction ; (iv) le réseau de Petri ne serait lu et analysé qu'une unique fois ; (v) l'outil connaissant à l'avance le nombre et le patron des formules à évaluer sur le même réseau, il pourrait essayer d'appliquer des simplifications préliminaires (telles que les réductions structurelles) et des optimisations sophistiquées sur le modèle ou les formules (en exploitant, par exemple, la quasi-réflexivité de la relation de concurrence) ; (vi) l'outil pourrait plus aisément tirer parti de la *vérification globale de modèles* (c.-à-d., construire au préalable l'ensemble des marquages accessibles, puis utiliser celui-ci pour évaluer toutes les formules) en remplacement ou en complément d'une *vérification locale de modèles* (c.-à-d., pour chaque formule, effectuer une évaluation à la volée, n'explorant qu'un fragment pertinent de l'ensemble des marquages accessibles pour la formule en question), qui souvent est le choix « par défaut », voire « à défaut » ; (vii) de telles options donneraient un cadre propice à l'élaboration d'algorithmes non pas génériques, mais dédiés, plus efficaces sur ces problèmes particuliers.

Bien que les logiques temporelles sont une des voies possibles pour calculer les places mortes et les transitions mortes, elles ne sont pas l'unique solution. Nous proposons des approches alternatives, avec des algorithmes dédiés et implantés dans des outils logiciels.

5.4 Complexité des problèmes

Proposition 5.1 *Le problème des places mortes est un sous-problème de celui des transitions mortes.*

Preuve. La connaissance de l'ensemble des transitions mortes est suffisante pour calculer l'ensemble des places mortes. En effet, une place morte est une place qui n'a pas de jeton dans le marquage initial et qui ne reçoit jamais de jeton (autrement dit, une place qui, pour toute transition non-morte, n'est jamais une de ses places de sortie). La réciproque est fausse : une transition peut être morte alors que toutes ses places d'entrée et de sortie ne sont pas mortes. □

Proposition 5.2 *Le problème des places mortes et le problème des transitions mortes sont des sous-problèmes du* problème de couverture de marquage*, ce dernier consistant, dans un réseau, à décider si pour un marquage donné, il existe un marquage accessible l'incluant.*

Preuve. En utilisant le prédicat $\Re$ de la définition 2.21, ainsi que les définitions 5.1 et 5.3, l'ensemble des places mortes s'exprime $\{p \in P \mid \neg\Re(\{p\})\}$, tandis que celui des transitions mortes s'exprime $\{t \in T \mid \neg\Re({}^\bullet t)\}$. □

Proposition 5.3 *Pour les réseaux saufs, le problème des places mortes et le problème des transitions mortes sont PSPACE-complets.*

Preuve. Nous savons, d'après [CEP95, Th. 15], que le problème de couverture des marquages est PSPACE-complet pour les réseaux saufs. D'après la proposition 5.1, il suffit donc de démontrer que le problème des places mortes est PSPACE-complet pour montrer que le problème des transitions mortes est PSPACE-complet. Avec la proposition 5.2, nous savons déjà que le problème des places mortes est au plus PSPACE-complet. Il reste à démontrer que le problème des places mortes est au moins PSPACE-complet. Soit un réseau ordinaire et sauf N. Soit N' le réseau consistant en N auquel nous rajoutons une nouvelle place p et une nouvelle transition t tels que ${}^\bullet t = M$ et $t^\bullet = \{p\}$; puisque N est sauf, N' est également sauf ; enfin, décider dans N' si $\Re(\{p\})$ revient à décider dans N si $\Re(M)$. □

5.5 De la nécessité de solutions à valeurs ternaires

Les deux problèmes susmentionnés étant PSPACE-complets, il existera toujours un réseau de Petri suffisamment grand pour empêcher le calcul de leurs solutions exhaustives sur un ordinateur donné. Plutôt que de nous contenter d'une approche « tout ou rien », où un algorithme peut soit donner toutes les places mortes ou toutes les transitions mortes, soit être déclaré comme mis en échec, nous considérons une approche plus pragmatique : l'algorithme peut s'arrêter de lui-même, ou bien être interrompu après n'avoir calculé qu'une partie de la solution, en laissant éventuellement certains résultats inconnus.

Concrètement, cela signifie que dans une solution, chaque place ou transition peut avoir une valeur connue (à savoir « non-morte » $\bot$ ou « morte » $\top$), ou une valeur inconnue ($\perp\!\!\!\top$). Une

solution sera dite *partielle* si elle contient au moins une valeur inconnue, ou sinon *complète*. Un algorithme efficace doit être en mesure d'éliminer, au cours d'un laps de temps donné, le plus grand nombre de valeurs inconnues possible.

Définition 5.5 *Le* vecteur des places mortes *peut se représenter par un triplet* $(P_\bot, P_\top, P_\perp)$ *tel que :*

- $P_\bot \subseteq \{p \in P \mid \Re(\{p\})\}$ *est l'ensemble des places déterminées non-mortes ;*
- $P_\top \subseteq \{p \in P \mid \neg\Re(\{p\})\}$ *est l'ensemble des places déterminées mortes ;*
- $P_\perp \stackrel{\text{def}}{=} P \setminus (P_\bot \cup P_\top)$ *est le prédicat donnant les places dont on ne sait pas si elles sont mortes ou non.*

Notons que $P_\bot$, $P_\top$ *et* $P_\perp$ *sont deux à deux disjoints et que leur union est égale à* P. *Initialement, Le triplet est initialisé à* $(\varnothing, \varnothing, P)$, *c.-à-d. que la solution ne présente d'origine que des valeurs inconnues. Une fois la solution complètement déterminée, le triplet vaut* $(\{p \in P \mid \Re(\{p\})\}, \{p \in P \mid \neg\Re(\{p\})\}, \varnothing)$.

De manière analogue, le vecteur des transitions mortes sera représenté par le triplet défini comme suit :

Définition 5.6 *Le* vecteur des transitions mortes *peut se représenter par un triplet* $(T_\bot, T_\top, T_\perp)$ *tel que :*

- $T_\bot \subseteq \{t \in T \mid \Re({}^\bullet t)\}$ *est l'ensemble des transitions déterminées non-mortes ;*
- $T_\top \subseteq \{t \in T \mid \neg\Re({}^\bullet t)\}$ *est l'ensemble des transitions déterminées mortes ;*
- $T_\perp \stackrel{\text{def}}{=} T \setminus (T_\bot \cup T_\top)$ *est le prédicat donnant les transitions dont on ne sait pas si elles sont mortes ou non.*

Le triplet est initialisé à $(\varnothing, \varnothing, T)$, *c.-à-d. que la solution ne présente d'origine que des valeurs inconnues. Le triplet vaut* $(\{t \in T \mid \Re(\{t\})\}, \{t \in T \mid \neg\Re(\{t\})\}, \varnothing)$, *une fois la solution complète. Notons qu'à tout instant,* $T_\bot$, $T_\top$ *et* $T_\perp$ *sont deux à deux disjoints et que leur union est égale à* T.

Une solution partielle d'un vecteur des places mortes ou des transitions mortes peut être exploitée de manière conservatrice, en considérant les valeurs « inconnues » de la même manière que les valeurs « fausses », ce qui revient à dire que nous considérons qu'une place ou transition est morte uniquement si cela a été positivement prouvé.

5.6 Exploration des marquages accessibles

L'approche la plus aisée, du moins conceptuellement, pour calculer les places mortes et les transitions mortes d'un réseau de Petri sauf, consiste à déterminer les places ayant reçu un jeton et les transitions devenues actives lors d'une exploration de l'ensemble de marquages accessibles du réseau. Il s'agit d'une approche de vérification *globale* de modèles (voir la sous-section 5.3.3).

Dans cet algorithme, les marquages accessibles sont représentés sous la forme de diagrammes de décision binaires : une telle exploration, dite *symbolique*, obtient en pratique de meilleurs résultats qu'une exploration reposant sur une représentation explicite des marquages, qu'il s'agisse en terme de temps ou de mémoire.

De plus, cette étape tire parti des informations concernant l'arbre des unités et de la répartition des places dans les unités, afin d'élaborer un codage optimisé des marquages, respectant la propriété unit-sauf (voir la section 3.4). Ceci permet d'accélérer l'exploration.

Pour des réseaux trop grands, qui excèdent les capacités de l'exploration symbolique, il est possible de spécifier une borne supérieure dans la profondeur de l'exploration, ou bien une limite de temps à ne pas dépasser.

Si au terme de cette exploration, l'espace d'états a pu être parcouru exhaustivement, nous pouvons en déduire, par simple négation, les ensembles de places et de transitions mortes. À l'opposé, si l'espace d'états est trop vaste pour être parcouru exhaustivement, les vecteurs ne contiendront que des valeurs $\bot$ ou $\Bumpeq$, mais jamais de valeurs $\top$.

Enfin, il est possible d'obtenir des vecteurs complets, sans valeurs inconnues, tout en évitant de parcourir exhaustivement l'ensemble de l'espace d'états. Ceci est possible via l'utilisation de *raccourcis* algorithmiques, en interrompant l'exploration dès que toutes les informations requises ont été déterminées (ce qui reprend l'idée de vérification *à la volée*) :

— Le premier raccourci s'applique aussi bien au calcul du vecteur des places mortes que celui des transitions mortes : lorsque le vecteur des transitions mortes ne contient aucune de valeur inconnue, l'exploration peut être interrompue. Un exemple de réseaux bénéficiant très largement de ce raccourci sont ceux sans transition morte.

— Le second raccourci s'applique seulement pour la détermination du vecteur des places mortes : lorsque le vecteur des places mortes ne contient plus aucune valeur inconnue, l'exploration peut être interrompue. Un exemple de réseaux bénéficiant très largement de ce raccourci sont ceux contenant de nombreuses transitions redondantes.

L'algorithme de cette section peut se formaliser ainsi.

```
1  P⊥ := M0

3  DELAI_EXPIRÉ := ⊥        — booléen positionné à ⊤ en cas de dépassement de délai
4  BORNE := +∞                                  — borne sur la profondeur d'itération
5  E := ∅
6  E' := {M0}

8  repeat
9        for t ∈ T⊤ ∪ T⊥ loop
10             E'' := marquages_où_la_transition_est_active (E', t) \ E
11             if E'' ≠ ∅ then
12                   T⊥ := T⊥ ∪ {t}
13                   P⊥ := P⊥ ∪ t•
```

```
14                    if T_⊥ = ∅ then
15                        return      — raccourci pour les places et transitions mortes
16                    else if P_⊥ = ∅ then
17                        return      — raccourci pour les places mortes (uniquement)
18                    else
19                        E' := E' ∪ tirer_la_transition (E'', t)
20                    end if
21                end if
22            end loop
23            E' := E' \ E
24            E := E ∪ E'
25            BORNE := BORNE − 1                                  — avec +∞ − 1 = +∞
26    until E' = ∅ or BORNE = 0 end repeat

28    if E' = ∅ and ¬DELAI_EXPIRÉ then                            — l'exploration est complète
29            T_⊤ := T_⊤ ∪ T_⊥
30            P_⊤ := P_⊤ ∪ P_⊥
31    end if
```

Algorithme A_1 – Exploration des marquages accessibles pour les places et transitions mortes

5.7 Règles structurelles

L'exploration des marquages accessibles est une approche par force brute, qui peut échouer sur des réseaux de grande taille, aboutissant à des vecteurs incomplets. Nous examinons, de cette section jusqu'à la section 5.9, des algorithmes de moindre complexité, prenant en entrée un vecteur possédant des valeurs inconnues et produisant un nouveau vecteur préservant les valeurs déjà connues en entrée, tout en ayant moins de valeurs inconnues.

Dans cette section, nous présentons un ensemble de quatre règles « structurelles » pouvant déterminer que certaines places ou transitions sont mortes ou non.

Proposition 5.4 *Toute place appartenant au marquage initial M_0 est non-morte.*

Proposition 5.5 *Toute transition n'ayant pas de place d'entrée est non-morte.*

Les deux règles suivantes tirent parti des propriétés sauf (nous faisons l'hypothèse que le réseau de Petri est sauf) et unit-sauf (une importante proportion de nos modèles sont unit-saufs par construction, voir la section 5.12) pour détecter certaines classes de transitions mortes.

Proposition 5.6 *Dans un réseau sauf, toute transition dont les places d'entrée sont un strict sous-ensemble des places de sortie est morte.*

Preuve. S'il est possible de tirer une telle transition t une fois, il est possible de la tirer un nombre infini de fois en accumulant un nombre infini de jetons dans les places de $t^\bullet \setminus {}^\bullet t$. □

Notons que, si une transition n'a pas de place d'entrée, mais au moins une place de sortie, le réseau n'est pas sauf, puisque nous pouvons toujours franchir cette transition et donc accumuler des jetons dans ses places de sortie.

Proposition 5.7 (d'après [Gar19, prop. 8]) *Si le réseau est unit-sauf, toute transition ayant au moins deux places d'entrée (respectivement deux places de sortie) localisées dans deux unités NUPN non-disjointes est morte.*

Nous pouvons remarquer que les règles des propositions 5.4 et 5.5 apportent des $\bot$, tandis que les règles des propositions 5.6 et 5.7 apportent des $\top$.

Avec les notations des chapitres 2 à 3, les règles structurelles des propositions 5.4 à 5.7 forment l'algorithme suivant :

1 $P_\bot \coloneqq P_\bot \cup M_0$ — *d'après la prop. 5.4*
2 $T_\bot \coloneqq T_\bot \cup \{t \in T \mid {}^\bullet t = t^\bullet = \varnothing\}$ — *d'après la prop. 5.5*

4 — *la suite de cet algorithme suppose que le réseau est sauf*
5 $T_\top \coloneqq T_\top \cup \{t \in T \mid ({}^\bullet t \subseteq t^\bullet) \wedge ({}^\bullet t \neq t^\bullet)\}$ — *d'après la prop. 5.6*

7 — *la suite de cet algorithme suppose que le réseau est unit-sauf*
8 $T_\top \coloneqq T_\top \cup$ — *d'après la prop. 5.7*
9 $\quad \{t \in T \mid \exists (p_1, p_2) \in ({}^\bullet t \times {}^\bullet t) \cup (t^\bullet \times t^\bullet), \neg\, \mathsf{disjoint}\,(\mathsf{unité}\,(p_1), \mathsf{unité}\,(p_2))\}$

Algorithme A_2 – Règles structurelles

Ces règles ne dépendent pas des valeurs initiales des vecteurs de places mortes $(P_\bot, P_\top, P_\top)$ et des transitions mortes $(T_\bot, T_\top, T_\top)$.

5.8 Autres règles structurelles

Nous présentons un algorithme consistant en l'application jusqu'à saturation de quatre nouvelles règles élémentaires, permettant de propager, dans certains cas, le fait qu'une place (respectivement une transition) soit morte ou non vers ses transitions (respectivement ses places) adjacentes.

Proposition 5.8 *Si une transition t est morte, toute place p telle que ${}^\bullet t = \{p\}$ l'est aussi.*

Proposition 5.9 (d'après [DE95, prop. 4.17(3)]) *Si une place p est morte, toute transition de ${}^\bullet p \cup p^\bullet$ l'est aussi.*

À noter que la réciproque de la proposition 5.9 est fausse : une transition peut être morte tout en n'étant connectée qu'à des places non-mortes.

Proposition 5.10 (contraposée de la proposition 5.8) *Si une place p est non-morte, toute transition t telle que ${}^\bullet t = \{p\}$ l'est aussi.*

Proposition 5.11 (contraposée de la proposition 5.9) *Si une transition t est non-morte, toute place de ${}^\bullet t \cup t^\bullet$ l'est aussi.*

L'algorithme A_3 ci-dessous applique itérativement ces quatre règles jusqu'à saturation. La boucle des lignes 1 à 15 applique les propositions 5.8 à 5.9, lesquelles apportent des $\top$: elle calcule ainsi une sous-approximation des vecteurs des places mortes et des transitions mortes. La boucle des lignes 17 à 31 applique, quant à elle, les propositions 5.10 à 5.11, qui apportent au contraire des $\bot$: de ce fait, elle représente une sur-approximation pour ces deux vecteurs.

1 $T' \coloneqq \varnothing$; $P' \coloneqq \varnothing$
2 **repeat**
3 **for** $t \in T_\top \setminus T'$ **loop**
4 $P_\top \coloneqq P_\top \cup \{p \in P \mid {}^\bullet t = \{p\}\}$ — *d'après la prop. 5.8*
5 **end loop**
6 $T' \coloneqq T_\top$
7 **assert** $(T' = T_\top) \wedge (P' \subseteq P_\top)$

9 **for** $p \in P_\top \setminus P'$ **loop**
10 $T_\top \coloneqq T_\top \cup {}^\bullet p \cup p^\bullet$ — *d'après la prop. 5.9*
11 **end loop**
12 $P' \coloneqq P_\top$
13 **assert** $(T' \subseteq T_\top) \wedge (P' = P_\top)$
14 **until** $T' = T_\top$ **end repeat**
15 **assert** $(T' = T_\top) \wedge (P' = P_\top)$

17 $P' \coloneqq \varnothing$; $T' \coloneqq \varnothing$
18 **repeat**
19 **for** $p \in P_\bot \setminus P'$ **loop**
20 $T_\bot \coloneqq T_\bot \cup \{t \in T \mid {}^\bullet t = \{p\}\}$ — *d'après la prop. 5.10*
21 **end loop**
22 $P' \coloneqq P_\bot$
23 **assert** $(P' = P_\bot) \wedge (T' \subseteq T_\bot)$

25 **for** $t \in T_\bot \setminus T'$ **loop**
26 $P_\bot \coloneqq P_\bot \cup {}^\bullet t \cup t^\bullet$ — *d'après la prop. 5.11*
27 **end loop**
28 $T' \coloneqq T_\bot$
29 **assert** $(P' \subseteq P_\bot) \wedge (T' = T_\bot)$
30 **until** $P' = P_\bot$ **end repeat**
31 **assert** $(P' = P_\bot) \wedge (T' = T_\bot)$

Algorithme A_3 – Autres règles structurelles

Proposition 5.12 *L'algorithme A_3 se termine.*

Preuve. L'ensemble $P' \cup T'$ est strictement croissant pour l'inclusion entre deux itérations de la première boucle (des lignes 1 à 15). Or, à la ligne 14, nous avons : $(P' \cup T') \subseteq (P \cup T)$.

Puisque les ensembles de places et de transitions sont finis, la première boucle se termine après un nombre fini d'itérations. Ce même raisonnement s'applique pour la seconde boucle (des lignes 17 à 31), qui se termine après un nombre fini d'itérations. □

Lors de la terminaison de l'algorithme A_3, les propositions 5.8 à 5.11 ont été appliquées à saturation sur toutes les places de $P_\bot \cup P_\top$ et toutes les transitions de $T_\bot \cup T_\top$.

Il est à noter que l'algorithme A_3 ne peut déterminer aucune valeur si $P_\mathbb{I} = P$ et $T_\mathbb{I} = T$. Plus les vecteurs des places et des transitions mortes contiennent de valeurs connues (suite par exemple, à l'application des algorithmes des sections 5.6 et 5.7), plus A_3 est susceptible, par propagation, de déterminer davantage de nouvelles valeurs.

5.9 Sur-approximation linéaire

Définition 5.7 *Soient deux marquages M_1 et M_2, ainsi qu'une transition t. Nous notons $M_1 \overset{t}{\Longrightarrow} M_2$* ssi *$t$ est active dans M_1 (autrement dit, ${}^\bullet t \subseteq M_1$) et $M_2 = M_1 \cup t^\bullet$.*

Cette relation se distingue de la relation de tir usuelle $M_1 \xrightarrow{t} M_2$ (voir la définition 2.19), dans la mesure où cette dernière utilise $(M_1 \setminus {}^\bullet t)$ à la place de M_1. Ainsi, lorsqu'une transition t est franchie en employant la définition 5.7, les jetons des places d'entrée sont conservés, tandis que chaque place de sortie gagne un jeton. Autrement dit, si nous supposons que chaque place peut contenir soit aucun jeton (lorsqu'elle n'est pas marquée), soit une infinité de jetons (lorsqu'elle est marquée), $\overset{t}{\Longrightarrow}$ se comporte comme $\xrightarrow{t}$.

Proposition 5.13 *Si un marquage M est accessible depuis le marquage initial M_0, c.-à-d., $M_0 \xrightarrow{*} M$, il existe un marquage M' tel que $M_0 \overset{*}{\Longrightarrow} M'$ où $M \subseteq M'$.*

Preuve. Par induction sur les séquences de tir $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots M_{n-1} \xrightarrow{t_n} M$. Base (chemin de taille 0) : nous avons $M_0 \overset{*}{\Longrightarrow} M_0$. Induction : nous supposons que pour tout chemin de longueur d'au plus n, la proposition 5.13 est vérifiée. Soit une séquence de franchissements $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots M_{n-1} \xrightarrow{t_n} M_n$. Ce chemin étant de longueur n, il existe un marquage M'_n tel que $M_0 \overset{*}{\Longrightarrow} M'_n$ et $M_n \subseteq M'_n$. Montrons que pour tout chemin de longueur $n+1$, la proposition 5.13 est vraie. Soit une transition t_{n+1} franchissable depuis M_n. Nous avons : $M_n \xrightarrow{t_{n+1}} M_{n+1}$, avec $M_{n+1} = (M_n \setminus {}^\bullet t_{n+1}) \cup t_{n+1}{}^\bullet$. La transition t_{n+1} est franchissable depuis M'_n au sens de $\overset{t}{\Longrightarrow}$, car elle est franchissable depuis M_n au sens de $\xrightarrow{t}$: $({}^\bullet t_{n+1} \subseteq M_n) \wedge (M_n \subseteq M'_n) \Rightarrow ({}^\bullet t_{n+1} \subseteq M'_n)$. Nous pouvons ainsi écrire $M'_n \overset{t_{n+1}}{\Longrightarrow} M'_{n+1}$, avec $M'_{n+1} = M'_n \cup t_{n+1}{}^\bullet$. Nous obtenons alors $M_{n+1} = ((M_n \setminus {}^\bullet t_{n+1}) \cup t_{n+1}{}^\bullet) \subseteq (M_n \cup t_{n+1}{}^\bullet) \subseteq (M'_n \cup t_{n+1}{}^\bullet) = M'_{n+1}$ et ainsi, l'hypothèse de récurrence à l'ordre $n+1$ est validée. □

L'algorithme A_4 ci-après repose sur la contraposée de la proposition 5.13. Il effectue une exploration des marquages accessibles depuis M_0 en utilisant $\overset{*}{\Longrightarrow}$ à la place de $\xrightarrow{*}$. Lors de cette exploration, aucune place ne perd ses jetons une fois marquée. Ce qui permet de représenter simplement son espace d'états par les ensembles P' et P'', qui représentent respectivement l'ensemble des places visitées et l'ensemble des marquages accessibles et qui

sont liés par l'inéquation suivante : $(M_0 \cup P') \subseteq P'' \subseteq P$.

Nous accélérons le tir des transitions en attachant à chaque transition t (n'ayant pas été détecté morte) un compteur $t \mapsto c\,[t]$ contenant le nombre de places d'entrée de t n'ayant pas encore été marquées (selon $\overset{t}{\Longrightarrow}$). Ainsi t devient active (pour $\overset{t}{\Longrightarrow}$) lorsque $c\,[t]$ atteint zéro.

```
1  P' := ∅
2  P'' := P⊥ ∪ M0
3  for t ∈ T \ T⊤ loop
4        c[t] := |•t|
5  end loop

7  while P' ≠ P'' loop
8        assert (P' ⊂ P'') ∧ (P'' ∩ P⊤ = ∅) ∧ (∀t ∈ T \ T⊤) (c[t] = |•t \ P'|)
9        let p = oneof (P'' \ P')
10       P' := P' ∪ {p}

12       for t ∈ p• \ T⊤ loop
13             c[t] := c[t] − 1
14             if c[t] = 0 then
15                   P'' := P'' ∪ t•
16             end if
17       end loop
18 end loop

20 assert (P' = P'') ∧ (P' ∩ P⊤ = ∅) ∧ (∀t ∈ T \ T⊤) (c[t] = |•t \ P'|)
21 P⊤ := P⊤ ∪ (P \ P')
22 T⊤ := T⊤ ∪ {t ∈ T \ T⊤ | c[t] > 0}
```

Algorithme A_4 – Sur-approximation linéaire

Proposition 5.14 *L'algorithme A_4 se termine.*

Preuve. Le nombre de places est fini, donc P' et P'' sont finis. Puisque, $P' \subseteq P''$, à chaque itération, P' est strictement croissant pour l'inclusion. Conséquemment, la boucle « **while** » se termine au bout d'un nombre fini d'itérations. □

Lorsque algorithme A_4 se termine, $P' = P''$, ce qui signifie que l'ensemble de l'espace d'états a été visité. Après terminaison de l'exploration, toute place n'ayant pas été marquée est morte (elle est donc ajoutée à $P_\top$) et toute transition n'ayant jamais été franchie est morte (elle est donc ajoutée à $T_\top$). La réciproque est cependant fausse : à moins que les transitions ont toutes au plus une place d'entrée, cet algorithme réalisant une sur-approximation du nombre de jetons et de l'ensemble des transitions actives, il peut omettre certaines places ou transitions mortes, d'où le fait que $P_\bot$ et $T_\bot$ restent inchangés.

À la ligne 21, le nouvel ensemble de places mortes ($P \setminus P'$) contient l'ensemble initial des places mortes $P_\top$.

5.10 Ordre des algorithmes

Les algorithmes présentés au sein des sections 5.6, 5.7, 5.8 et 5.9 sont respectivement nommés A_1, A_2, A_3 et A_4. Il est aisé de constater qu'après avoir exécuté chacun de ces algorithmes, l'appliquer à nouveau immédiatement n'apportera aucun bénéfice concernant le nombre de valeurs inconnues. Mais cela ne veut pas dire qu'exécuter deux fois ces mêmes algorithmes serait toujours infructueux dans le cas où l'on intercalerait, entre les deux appels, un autre algorithme retirant d'autres valeurs inconnues.

Cela conduit à cette nouvelle interrogation : dans quel ordre, et combien de fois, devrions-nous appliquer ces quatre algorithmes ? Une analyse des apports et coûts potentiels de ces algorithmes suggère que $(A_2; A_3; A_4; A_1; A_3)$ serait l'ordre d'appel présentant les meilleurs résultats. Il tient notamment compte du fait que : (i) A_3 et A_4 bénéficient des informations pré-calculées par A_2, l'algorithme le moins coûteux ; (ii) A_1, l'algorithme le plus coûteux, peut bénéficier des informations précalculées par $(A_2; A_3; A_4)$, étant donné que A_1 peut éviter d'essayer de franchir des transitions qui sont déjà prouvées mortes et que ces résultats déjà connus accroissent l'efficacité des raccourcis définis à la section 5.6 ; (iii) enfin, l'algorithme A_3 est le seul pouvant, éventuellement, apporter de nouvelles valeurs suite à une première exécution de la séquence $(A_2; A_3; A_4; A_1)$.

Finalement, entre deux algorithmes, nous devons vérifier que le nombre de valeurs inconnues soit toujours strictement supérieur à zéro. Si ce n'est plus le cas, nous pouvons nous arrêter, la solution étant déjà complète.

5.11 Implantation logicielle et formats de fichiers

Nous avons implantés nos algorithmes dans l'outil CÆSAR.BDD, présenté à la section 3.7.

Cet outil accepte des modèles en entrée au format NUPN, qui peuvent être produit à partir de modèles au format normalisé PNML, à l'aide d'une traduction automatisée (voir la section 3.6).

5.11.1 Calcul des places et transitions mortes

Les options permettant le calcul des places mortes et des transitions mortes, respectivement « `-dead-places` » et « `-dead-transitions` », produisent, comme expliqué précédemment dans la section 5.5, des vecteurs dont chaque élément peut avoir une valeur parmi $\{\bot, \top, \mathbb{I}\}$.

Ceci associé au fait que dans le format NUPN, les ensembles de places et les ensembles de transitions sont des intervalles, les vecteurs des définitions 5.5 et 5.6 se concrétisent alors sous la forme de fichiers textes (voir [Gar20, sect. 8 et annexe A]), constitués d'une unique

ligne contenant des caractères « `0` » pour $\bot$, « `.` » pour $\amalg$ et « `1` » pour $\top$, représentant la valeur affectée à chacune des places ou des transitions du réseau, triées par ordre croissant.

Ces vecteurs pouvant être de grande taille pour certains réseaux, une compression par plages est utilisée : plutôt que d'écrire $n > 3$ fois consécutivement un caractère, mettons « `0` », le vecteur contiendra « `0(n)` » pour ces n places ou n transitions. Cette compression permet une réduction totale de la taille des vecteurs d'un facteur de 9,91, pour la collection de la section 3.8.

5.11.2 Vérification de fiches pour le Model Checking Contest

Lors du déroulement des épreuves du MCC (voir la section 3.8), une vingtaine de propriétés structurelles et comportementales des réseaux de Petri sont mises à la disposition des outils participants, dont la présence d'au moins une place morte et la présence d'au moins une transition morte. Ces propriétés booléennes sont enregistrées dans les fiches [1] associées aux modèles et sont annotées de textes explicatifs, permettant par exemple de donner des noms de places ou de transitions mortes.

Ces fiches sont préalablement complétées et fournies par leurs auteurs lors de la soumission de chaque modèle. Il est donc important de s'assurer de la correction systématique de ces fiches. Cette étape permet également aux organisateurs de la compétition de s'assurer que les modèles soumis n'ont pas un nombre exagéré de places mortes ou de transitions mortes, ce qui conduirait à en surestimer leur difficulté.

À ces fins, l'outil CÆSAR.BDD est couramment utilisé par l'équipe d'organisation de la compétition, à l'aide d'une option dédiée « `-mcc` », calculant toutes ces propriétés de manière automatisée.

5.12 Résultats expérimentaux

Nous avons appliqué ces algorithmes implantés dans CÆSAR.BDD pour calculer les vecteurs des places mortes et des transitions mortes sur le jeu de tests présenté à la section 3.8. Les expérimentations de ce chapitre proviennent de [BG21a] et ont été réalisées en janvier 2021. La collection contenait alors 13 116 modèles. Les 3 084 modèles ajoutés depuis ne changent guère ses caractéristiques : les chiffres de [BG21a, Table 1] sont très proches de ceux des tables 3.3 et 3.4.

Nos expérimentations ont été réalisées (séquentiellement) sur des ordinateurs individuels équipés d'un processeur Intel Xeon E5-1650 v4 (3,6 GHz), de 128 Go de RAM et exécutant OpenIndiana, la version à code source ouvert du système d'exploitation Solaris 11. Ces expérimentations sont paramétrées par un temps t (en secondes), représentant le délai (maximal) d'exécution accordé à l'algorithme A_1 pour l'exploration symbolique des marquages accessibles, (ou, du moins, du plus grand sous-ensemble des marquages accessibles possible).

1. Voir `mcc.lip6.fr/verdict-properties.php` et `mcc.lip6.fr/models.php`.

Dans CÆSAR.BDD, cette exploration est réalisée, en interne, par la bibliothèque CUDD pour les diagrammes de décision binaires. Si t est nul, seul le marquage initial est exploré et aucune transition n'est tirée par A_1.

Nos expérimentations révèlent qu'au moins 16,2 % (respectivement 15,9 %) des modèles contiennent des places (respectivement transitions) mortes, et qu'au moins 20,4 % (respectivement 37.7 %) des places (respectivement transitions) mortes sont globalement présentes parmi les modèles. De tels ratios élevés confirment l'importance pratique de la détection et de l'élimination de places et de transitions mortes pour réduire la complexité des réseaux de Petri.

valeur de t	0	5	10	15	30	45
% vecteurs complets	44.6	93.0	93.6	93.8	94.4	94.6
% valeurs inconnues	48.9	33.5	32.0	31.3	28.9	28.3
% remplissage vecteurs	69.3	97.0	97.3	97.5	97.7	97.9

valeur de t	45	60	120	180	240	300
% vecteurs complets	94.6	95.1	95.3	95.4	95.5	95.6
% valeurs inconnues	28.3	27.9	27.1	26.5	25.9	25.8
% remplissage vecteurs	97.9	97.9	98.1	98.1	98.2	98.2

TABLE 5.1 – Résultats expérimentaux pour les places mortes

valeur de t	0	5	10	15	30	45
% vecteurs complets	29.3	92.3	92.9	93.2	93.7	94.0
% valeurs inconnues	68.7	65.0	63.5	62.0	61.0	59.3
% remplissage vecteurs	50.9	95.8	96.2	96.4	96.7	96.8

valeur de t	45	60	120	180	240	300
% vecteurs complets	94.0	94.1	94.4	94.7	94.9	95.0
% valeurs inconnues	59.3	57.8	54.6	45.2	39.9	29.8
% remplissage vecteurs	96.8	96.9	97.1	97.2	97.3	97.3

TABLE 5.2 – Résultats expérimentaux pour les transitions mortes

La table 5.1 (respectivement la table 5.2) apporte, pour diverses valeurs de t, trois métriques relatives au calcul des places (respectivement transitions) mortes. La première métrique, *% vecteurs complets*, donne le pourcentage de modèles dont le vecteur peut être complètement déterminé sous t secondes. La seconde métrique, *% valeurs inconnues*, s'obtient en calculant la somme de tous les vecteurs, puis en prenant le pourcentage de valeurs inconnues dans le vecteur résultant. La troisième et dernière métrique, *% remplissage vecteurs*, s'obtient en calculant le pourcentage de valeurs connues pour chaque modèle, puis en prenant la moyenne de ces pourcentages.

La première métrique montre pour $t = 0$, que les algorithmes A_2 et A_3 suffisent, à eux seuls, pour complètement déterminer 44,6 % (respectivement 29.3 %) vecteurs des places (respectivement transitions) mortes des modèles ; mais, dès lors que l'algorithme A_1 est mis à contribution (lorsque $t > 0$), la proportion de modèles complètement résolus s'améliore drastiquement, passant à 93,0 % (respectivement 92,3 %) de la collection. Partant de là, l'incrémentation de la valeur de t permet une augmentation de cette proportion, de manière certes plus modeste. Appliquer de nouveau l'algorithme A_3 après A_1 permet d'augmenter la proportion de modèles complètement résolus de 0,1 (respectivement 0,1) points. La première métrique est représentée sur la figure 5.1.

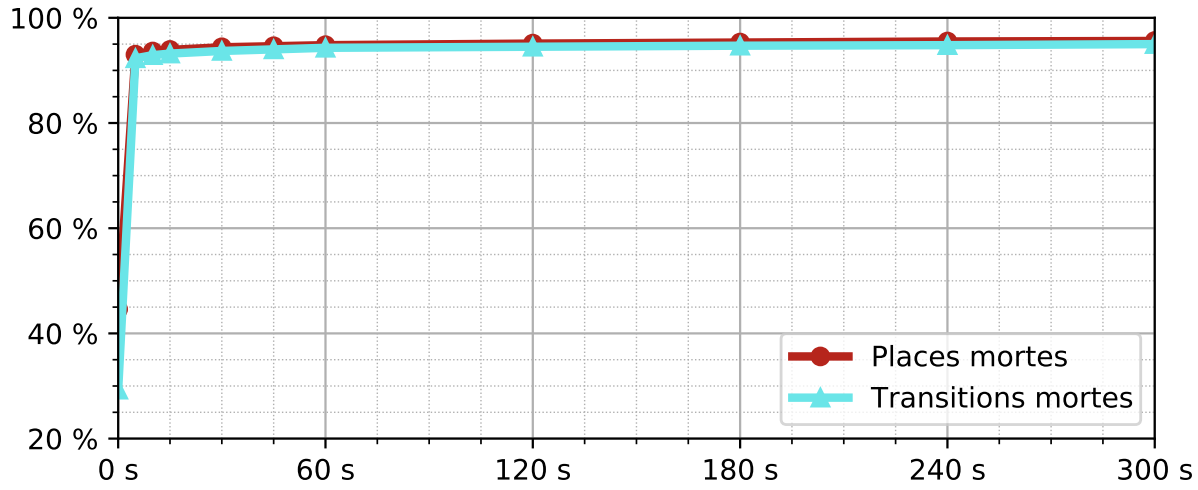

FIGURE 5.1 – Pourcentage de vecteurs complets en fonction de t

La seconde métrique, qui concerne les valeurs inconnues, donne des résultats assez similaires, bien que l'influence d'A_1 ne soit pas aussi importante que pour la première métrique. De plus, le pourcentage des valeurs inconnues ne converge pas aussi rapidement vers zéro, en raison d'un petit nombre de modèles de grande taille, incomplètement résolus pour des valeurs de t élevées, avec des milliers de valeurs inconnues. La figure 5.2 représente cette seconde métrique.

La troisième métrique corrobore la première, mais présente des pourcentages de réussite plus élevés, du fait que pour chaque modèle, la proportion de valeurs connues dans son vecteur de solution est comptabilisée, plutôt qu'une valeur binaire (soit le vecteur est entièrement complet, soit il ne l'est pas). Ceci est montré par la figure 5.3.

Des mesures supplémentaires pour les places (respectivement transitions) mortes indiquent que :

— les raccourcis de l'algorithme A_1 sont efficaces, puisqu'ils sont déclenchés pour plus de 82,6 % (respectivement 79,6 %) des modèles ;

— 94,8 % (respectivement 94,0 %) des modèles sont totalement résolus en moins d'une

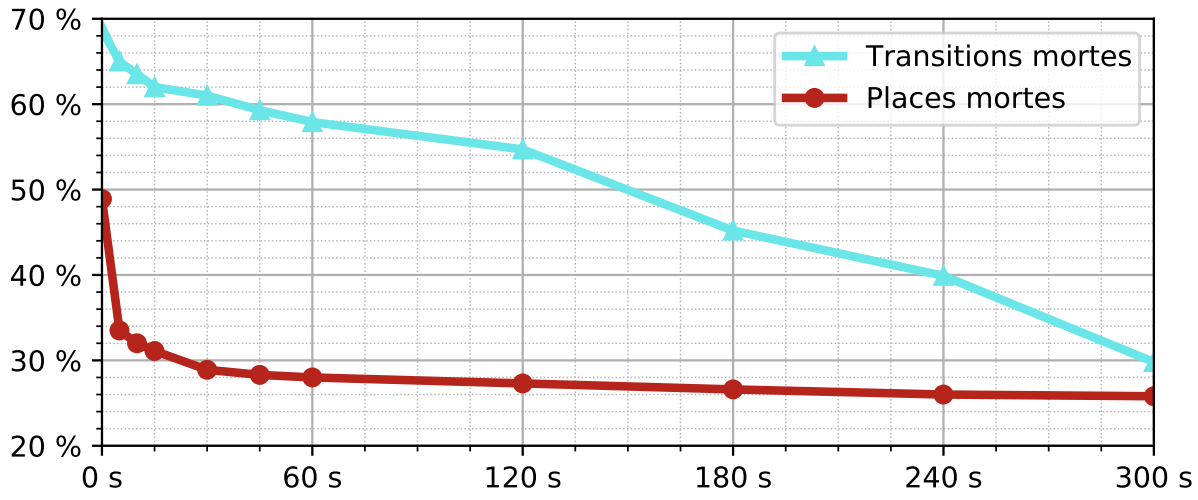

FIGURE 5.2 – Taux global de valeurs inconnues dans les vecteurs en fonction de t

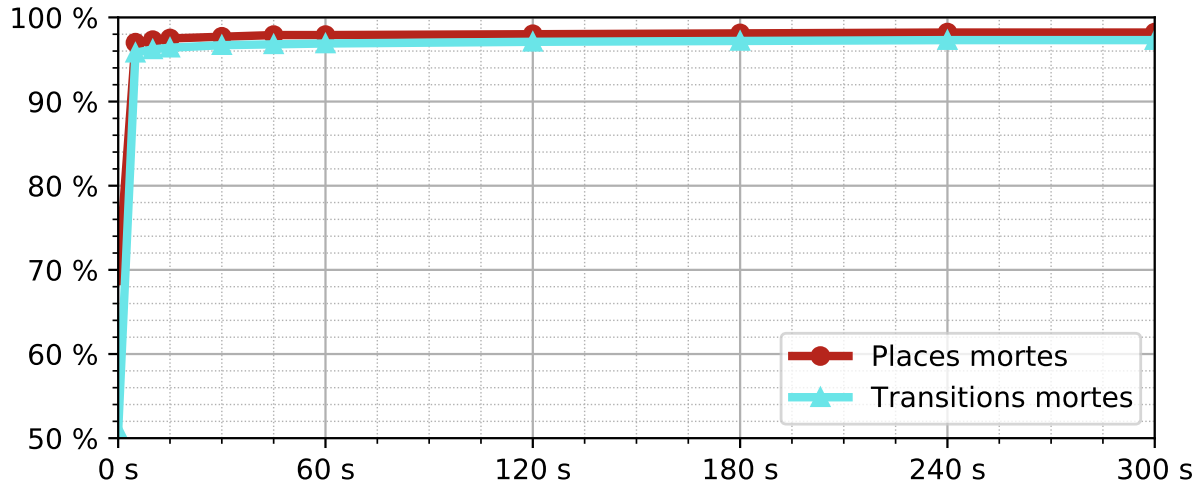

FIGURE 5.3 – Taux de remplissage moyen des vecteurs en fonction de t

seconde, les autres étant (incomplètement) traités en moins de $1{,}56 \times t$ (respectivement $1{,}48 \times t$) secondes ;

— avec $t = 60$ (respectivement $t = 180$), tous les réseaux ayant moins de 74 places, 92 transitions et 366 arcs peuvent être complètement traités.

5.13 Comparaison avec ITS-TOOLS

Dans cette section, nous souhaitons comparer nos algorithmes avec une autre approche, reposant sur l'évaluation de formules en logique temporelle (évoquée à la sous-section 5.3.3).

Pour cela, nous avons opté pour l'outil ITS-Tools [TM15] (développé par Yann Thierry-Mieg, au LIP6, Paris, France), un vérificateur de modèles pour divers formalismes, dont les réseaux de Petri. Il permet l'évaluation de diverses propriétés de sûreté et de formules en logique temporelle. Cet outil associe une exploration symbolique de l'espace d'états, reposant sur des *diagrammes de décision hiérarchiques* [TMPHK09], avec une large palette d'algorithmes reposant sur des formules logiques du premier ordre, des contraintes en algèbre linéaire ou sur de la marche aléatoire. Il possède de bonnes performances : par exemple, dans la catégorie « accessibilité » du MCC, ITS-Tools a reçu les médailles d'or en 2020 et 2021 et la médaille d'argent en 2022.

En 2021, Yann Thierry-Mieg a incorporé à ITS-Tools la possibilité de résoudre les problèmes des places mortes ou des transitions mortes par la génération de formules en logique temporelle. Ces formules, bien que générées par ITS-Tools, ne lui sont pas spécifiques : tout outil en mesure de concourir dans la catégorie « accessibilité » du MCC est en théorie capable de les traiter. La génération proprement dite d'une formule est réalisée en invoquant la commande « `its-tools` » avec l'un des drapeaux « `-gen-dead-place` » ou « `-gen-dead-transition` ». Une fois la formule évaluée, un script annexe [2] écrit en langage Perl permet, suivant le type de formule générée, de reconstituer le vecteur des places mortes ou du vecteur des transitions mortes selon le format décrit dans la section 5.11 [3].

En décembre 2022, nous avons effectué une comparaison des résultats d'ITS-Tools avec ceux de CÆSAR.BDD, en utilisant leurs dernières versions disponibles [4]. Aucun délai imparti pour l'exploration des marquages accessibles n'a été spécifié dans CÆSAR.BDD ou ITS-Tools. Ceci facilite l'interprétation des résultats : soit un vecteur est produit (il ne contient aucune valeur inconnue), soit aucun résultat n'est produit (la solution ne possède que des valeurs inconnues).

Les réseaux utilisés pour notre comparaison sont les 702 instances ordinaires et sauves des modèles de l'édition 2022 du MCC (voir la section 3.8), qui ont été obtenus comme suit. L'édition 2022 du MCC [5] fournit un total de 1 387 réseaux de Petri non-colorés. De ces 1 387 réseaux, nous avons gardé ceux qui étaient ordinaires et dont le marquage initial était sauf ; ceci a été fait en écartant tous les fichiers NUPN contenant une directive « `!multiple_arcs` » ou « `!multiple_initial_tokens` » (voir la section 3.6). Il en a résulté 777 réseaux, parmi lesquels nous avons sélectionné ceux qui étaient unit-saufs, soit parce qu'ils comportaient une directive « `!unit_safe` » (voir la section 3.6), soit en invoquant CÆSAR.BDD ou ConcNUPN (voir la section 6.14) pour vérifier la propriété en question. Nous avons ainsi obtenu 705 réseaux, dont nous en avons exclu trois qui étaient potentiellement isomorphes, en raison de signatures identiques à trois autres réseaux, selon l'approche de la section 4.5. Par la suite, l'approche de la section 4.6 a révélé qu'il n'y avait qu'une seule paire de réseaux réellement isomorphes, l'approximation reposant sur les signatures nous a fait perdre seulement deux réseaux sur 705.

2. github.com/yanntm/pnmcc-tests/blob/master/analysis/logs2matrix.pl.
3. Sans tenir compte de la phase de compression, qui a été omise, car optionnelle.
4. github.com/yanntm/ITS-Tools-MCC/tree/f8b3935215bc4726a71db61f5aef9eb3e2ea28e2.
5. https://mcc.lip6.fr/2022/models.php

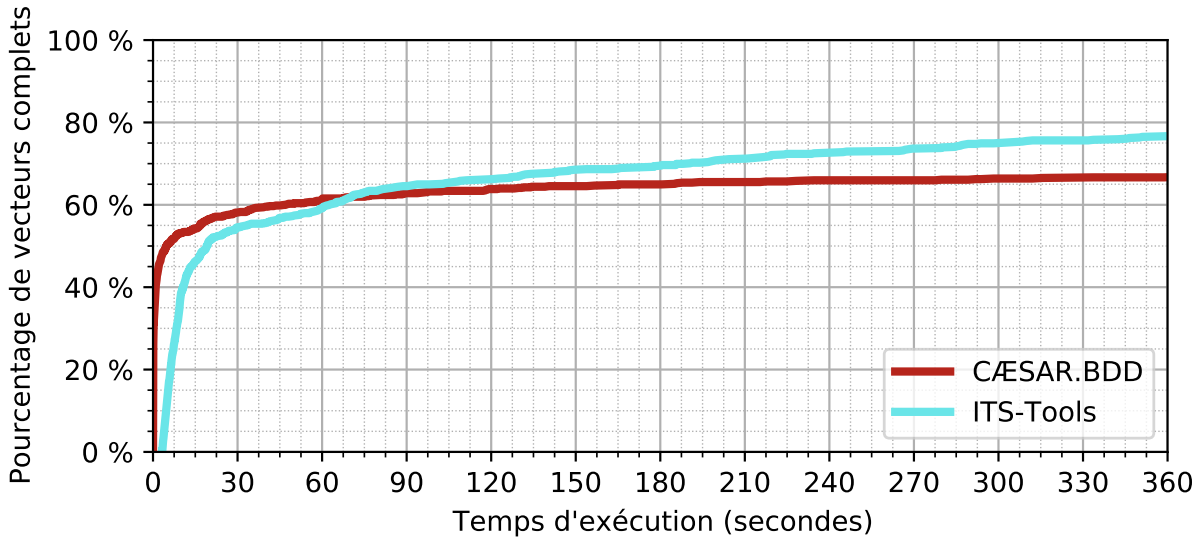

FIGURE 5.4 – Comparaison entre CÆSAR.BDD et ITS-TOOLS pour les places mortes

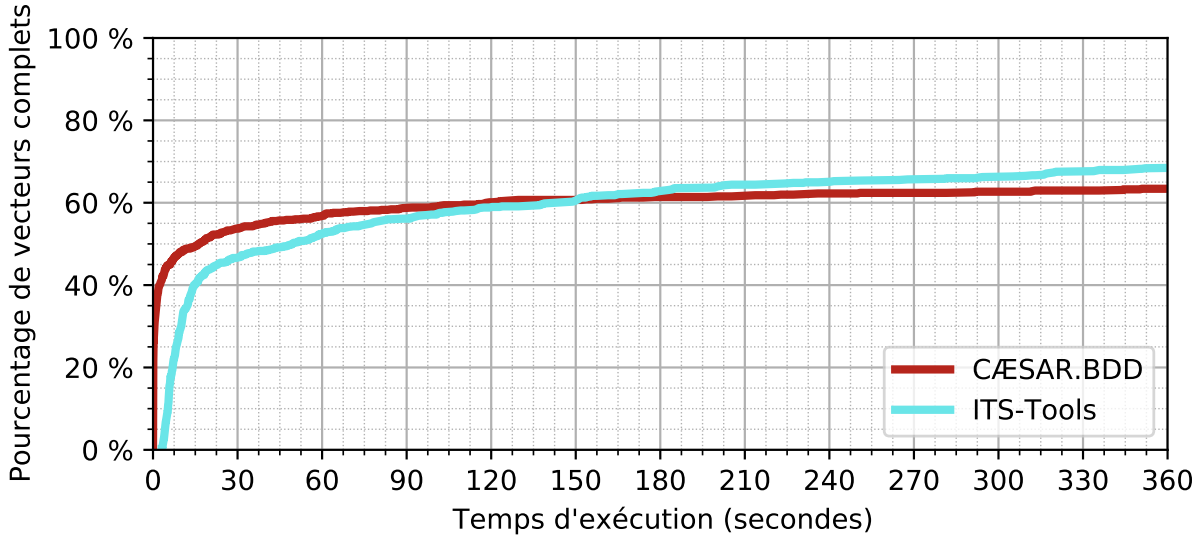

FIGURE 5.5 – Comparaison entre CÆSAR.BDD et ITS-TOOLS pour les transitions mortes

Les figures 5.4 et 5.5 représentent les proportions de vecteurs complets en fonction du temps d'exécution alloué par réseau. Nous constatons que, pour résoudre ces deux problèmes, les algorithmes dédiés de CÆSAR.BDD sont plus efficaces que les algorithmes génériques d'ITS-TOOLS : par exemple, pour les places mortes (respectivement transitions mortes), CÆSAR.BDD ne demande que 37 secondes (respectivement 23 secondes) pour obtenir le même nombre de vecteurs complets qu'ITS-TOOLS en une minute. Lorsque l'on laisse

davantage de temps aux outils (environ 70 secondes pour les places mortes et 150 secondes pour les transitions mortes), la situation s'inverse et ITS-Tools devient alors plus performant que CÆSAR.BDD : par exemple, en six minutes, ITS-Tools détermine 7,9 % (respectivement 15,0 %) de vecteurs de places mortes (respectivement de transitions mortes) supplémentaires par rapport à CÆSAR.BDD. La procédure d'exploration des marquages accessible d'ITS-Tools (reposant sur la bibliothèque libDDD [TMPHK09]) semble un peu plus efficace que celle CÆSAR.BDD (reposant sur la bibliothèque CUDD [Som01]).

5.14 Validation des résultats

À la section 5.13, nous nous sommes assurés que les résultats apportés par CÆSAR.BDD et ITS-Tools étaient identiques lorsque ces deux outils réussissaient tous les deux pour un modèle donné.

Afin de valider plus finement nos résultats, nous avons développé une collection de scripts Awk et shell, permettant de comparer les valeurs de deux vecteurs de places ou de transitions mortes. Ces scripts permettent de détecter les incohérences, autrement dit lorsqu'un vecteur évalue une place ou une transition comme morte, alors que l'autre évalue cette même place ou transition comme non-morte. Ces scripts permettent également de déterminer : (i) que les deux vecteurs sont égaux ; (ii) que l'un des deux vecteurs est meilleur que l'autre (l'ensemble des places ou transitions indéterminées d'un vecteur est strictement inclus dans l'ensemble des places ou transitions indéterminées dans l'autre vecteur) ; ou (iii) que deux vecteurs possèdent chacun des valeurs « exclusives » (les deux vecteurs sont distincts et aucun des deux n'est strictement meilleur que l'autre).

Pour cette validation, nous avons utilisé l'outil ConcNUPN (qui utilise d'autres algorithmes, du chapitre 6). Pour calculer les places mortes et les transitions mortes, ConcNUPN possède deux options « `-dead-places` » et « `-dead-transitions` », suivant les formats d'entrée et de sortie de CÆSAR.BDD.

Des vérifications croisées de CÆSAR.BDD, de ConcNUPN et de leurs versions successives ont été réalisées, en comparant deux à deux (à l'aide des scripts évoqués ci-dessus) les vecteurs de places mortes et de transitions mortes produits par ces outils, pour chaque modèle du jeu de tests de la section 3.8, qui contenait à l'époque 13 116 réseaux.

Afin de découvrir d'éventuels bogues sporadiques, des exécutions additionnelles de CÆSAR.BDD ont été réalisées en court-circuitant certains algorithmes, ou en limitant le temps ou la profondeur maximale de l'exploration des marquages accessibles.

Nous conservons, pour chaque modèle, une série de vecteurs des places mortes et de vecteurs des transitions mortes pré-calculés, dont quatre de référence (les meilleurs que CÆSAR.BDD ou ConcNUPN aient pu déterminer, pour chacun des deux problèmes).

Nous avons eu recours à des machines locales, ainsi qu'à la grille de calculs Grid'5000, afin de lancer des tests à la fois massifs, systématiques et automatisés, représentant plusieurs années de temps processeur, ceci afin de calculer des millions de vecteurs et de conserver

précieusement le contexte dans lequel ils ont été générées (à savoir les binaires utilisés, le temps maximal alloué et le nombre maximal d'itérations autorisées pour l'exploration des marquages accessibles, la borne globale sur le temps d'exécution autorisé, l'environnement logiciel utilisé, la configuration matérielle utilisée, et ainsi de suite). Pour donner plus d'assurance aux résultats, deux systèmes d'exploitations ont été utilisés, pour compiler CÆSAR.BDD et lancer les outils : Debian Linux (versions 8 à 11) et OpenIndiana (versions 2018.10 à 2021.10). À la suite de ces vérifications croisées, aucune erreur n'a été repérée dans les versions publiées de CÆSAR.BDD.

Par ailleurs, comme indiqué à la sous-section 5.11.2, lors de la soumission de modèles au MCC, les résultats délivrés par CÆSAR.BDD sont scrupuleusement comparés au contenu des fiches de ces modèles. Ces fiches contiennent (entre autres) des informations concernant les places mortes et les transitions mortes. Les résultats de CÆSAR.BDD ont aussi été comparés aux consensus obtenus pour la propriété « QuasiLiveness » par les outils participants au MCC. Aucune erreur n'a été repérée dans CÆSAR.BDD.

5.15 Bilan et perspectives

Ce chapitre a proposé une solution pragmatique pour deux problèmes anciens et utiles, ayant des formulations simples, tout en étant complexes à résoudre (PSPACE-complets) : les places mortes et les transitions mortes dans les réseaux de Petri ordinaires et saufs.

Diverses approches complémentaires ont été proposées, visant à détecter dans des temps raisonnables un maximum de places mortes et de transitions mortes.

Il sera toujours possible d'améliorer la détection des places mortes et des transitions mortes, afin de traiter des modèles de plus grande taille, en ayant recours à des stratégies toujours plus créatives et complexes. Le perfectionnement des algorithmes peut se faire, soit en calculant des solutions comportant moins de valeurs inconnues, soit en fournissant des résultats équivalents, mais plus rapidement. En ce sens, voici quelques pistes.

— L'exploration symbolique des marquages accessibles de la section 5.6 pourrait être améliorée de diverses manières. Un codage plus compact des marquages pourrait être envisagé pour CÆSAR.BDD, exploitant mieux l'imbrication hiérarchique des unités des NUPN. Notons qu'actuellement, les diagrammes de décision binaires représentent des marquages accessibles : une représentation des marquages couverts pourrait s'avérer plus compacte pour certains réseaux.

— D'après [Val89, corollaire 1.30], une exploration des marquages accessibles recourant à des ordres partiels et adressant le « problème de l'ignorance » tire toutes les transitions non-mortes. Ainsi, le problème des transitions mortes, et conséquemment, le problème des places mortes peuvent être traités avec des diagrammes de décision binaire (voir notamment [ABH+01] qui emploie un parcours en largeur) en réutilisant les méthodes classiques pour les ordres partiels.

— Nous pourrions également détecter les transitions tirables en série, utiliser des

techniques de *lignes de balayage* [MW04] [Sch06] [JKM12], réordonner l'ordre de franchissement des transitions en fonction de leurs places d'entrée et de sortie, et ainsi de suite.

— Une approche alternative consisterait en la traduction des problèmes des places mortes et des transitions mortes pour un réseau de Petri donné en formules logiques. Nous pourrions également recourir à des outils de vérification dédiés aux réseaux de Petri, pour cibler spécifiquement les valeurs inconnues demeurant dans un vecteur de solution calculé par d'autres algorithmes.

Deuxièmement, le problème des places mortes pourrait être généralisé. Il peut être étendu au cas des réseaux non-saufs, où il s'agirait de déterminer combien de jetons $k_p \in \mathbb{N} \cup \{+\infty\}$ chaque place p pourrait, au plus, contenir. Ceci généraliserait également le problème consistant à déterminer si un réseau est *k-sauf*, avec $k = \mathsf{max}\,(\{k_p \mid p \in P\})$, autrement dit, une valeur représentant une borne supérieure du nombre maximal de jetons que toute place du réseau pourrait contenir. Pour les réseaux non-saufs, mais bornés (autrement dit, *k-sauf*, avec $1 < k < +\infty$), utiliser les bornes k_p plutôt que la seule borne k permettrait d'améliorer l'efficacité des procédures d'exploration des marquages accessibles, en réduisant la taille de codage des marquages, tout en permettant de réduire la taille des réseaux produits par les procédures de réductions de tels réseaux en réseaux saufs. Concrètement, une solution pratique aurait, pour toute place, une borne inférieure $P_{Min}\,(p) \in \mathbb{N} \cup \{+\infty\}$ et une borne supérieure $P_{Max}\,(p) \in \mathbb{N} \cup \{+\infty\}$, avec $P_{Min}\,(p) \leq k_p \leq P_{Max}\,(p)$, en remplacement des valeurs ternaires $\bot$, $\top$ et $\perp\!\!\!\top$.

Troisièmement, le problème des transitions mortes pourrait être généralisé. Nous pourrions, dans les vecteurs des transitions mortes, distinguer les valeurs correspondant à L1-vivante, L2-vivante, L3-vivante et L4-vivante (voir la section 5.2), mais une telle généralisation ne semble pas avoir d'intérêt pratique évident : ces définitions alternatives font apparaître de manière sous-jacente des notions correspondant à des chemins d'exécution. De ce fait, les formules en logique temporelle sembleraient cette fois-ci être mieux adaptées pour résoudre les problèmes concrets.

Enfin, les problèmes des places mortes et des transitions mortes pourraient devenir partie intégrante du Model Checking Contest, comme le propose [Gar20], éventuellement après avoir été généralisés pour les réseaux non-saufs, ou les réseaux colorés.

Chapitre 6

Détermination des places concurrentes

Ce chapitre aborde le problème des places concurrentes, qui consiste en la recherche de toutes les paires de places pouvant recevoir chacune un jeton dans au moins un marquage accessible. Elles caractérisent les parties d'un réseau pouvant être simultanément actives et jouent, à ce titre, un rôle crucial dans la partie IV pour la structuration automatique de réseaux Petri ordinaires et saufs. Le présent chapitre présente divers algorithmes destinés au calcul des places concurrentes.

6.1 Motivations

Les places concurrentes sont utiles en pratique pour (au moins) quatre raisons.

Premièrement, elles servent à décomposer un réseau de Petri en NUPN (voir les chapitres 7 et 8). En effet, dans un NUPN unit-sauf, toute place d'une unité u est non-concurrente avec toute place d'une unité imbriquée dans u (voir la définition 3.7). De ce fait, la connaissance des places concurrentes est une condition à la fois nécessaire et suffisante pour déterminer l'intégralité des structures NUPN $(U, u_0, \sqsubseteq, \mathsf{unité})$ permettant de produire, à partir d'un réseau de Petri (P, T, F, M_0) ordinaire et sauf donné, un NUPN unit-sauf.

Deuxièmement, les places concurrentes aident à évaluer certaines propriétés de sûreté, qui peuvent se traduire par la non-accessibilité de marquages « indésirables ». Par exemple, elles permettent de définir des conditions suffisantes pour prouver qu'un réseau de Petri donné soit (non-) sauf ou qu'un NUPN donné soit (non-) unit-sauf : cette application sera abordée de manière plus élaborée à la section 6.14.

Troisièmement, les places concurrentes permettent des analyses de flux de données sur les réseaux de Petri. Par exemple, la boîte à outils CADP traduit les spécifications LOTOS en réseaux de Petri interprétés, avec des unités imbriquées représentant les processus du code LOTOS source (voir les sections 1.4 à 1.7). Afin de réduire le nombre de marquages

accessibles des réseaux produits, sur certaines transitions sont ajoutées des opérations de remise à zéro de variables dont les valeurs ne sont plus réutilisées par la suite. Mais ces remises à zéro peuvent introduire des conflits de type « lecture/écriture » pour certaines variables héritées d'un processus père et partagées par plusieurs processus fils. Ce problème est résolu par [GS06, sect. 5] au moyen d'une déduplication des variables en conflit, dont leur détection repose sur la notion d'*unités concurrentes* (voir la définition 6.3). Le problème des places concurrentes généralise le problème des unités concurrentes.

Quatrièmement, le problème des places concurrentes généralise le problème des places mortes (une place est concurrente à elle-même *ssi* elle est non-morte) : les algorithmes de ce chapitre (plus puissants, mais plus coûteux) peuvent être employés pour améliorer les résultats chapitre 5, lorsque des valeurs inconnues subsistent.

La suite de ce chapitre s'articule ainsi. La section 6.2 définit formellement le problème des places concurrentes. La section 6.3 dresse un inventaire de l'état de l'art. La section 6.4 analyse la complexité théorique de ce problème. La section 6.5 explique pourquoi il est nécessaire d'avoir des solutions dites « incomplètes », car pouvant présenter des valeurs inconnues. Les sections 6.6 à 6.10 décrivent ensuite des algorithmes dédiés, qui sont implantés dans des outils logiciels spécialisés, présentés à la section 6.11. La section 6.12 fait le point sur les performances de la solution proposée, une fois appliquée à un ensemble complet de modèles. La section 6.13 analyse les apports des réductions structurelles à la solution proposée, sur un (autre) ensemble complet de modèles. La section 6.14 propose une application des places concurrentes : la détermination de la propriété unit-sauf pour les NUPN (ou, de la propriété sauf pour les réseaux de Petri, en tant que cas particulier des NUPN). La section 6.15 aborde les démarches entreprises pour la validation des résultats. Finalement, la section 6.16 formule des observations conclusives et évoque les perspectives ouvertes.

6.2 Énoncé du problème

Dans la suite de ce chapitre, nous considérons, afin de simplifier les explications, que deux places forment toujours une paire, même lorsqu'il s'agit de places non-distinctes. Ceci rejoint la convention $\{p, p\} = \{p\}$ adoptée à la définition 2.3.

La définition suivante présente la notion de paires de places concurrentes au sein d'un réseau de Petri, sous la forme d'une relation binaire $\|$, en utilisant le prédicat $\Re$ de couverture de marquages de la définition 2.21.

Définition 6.1 *Soit* (P, T, F, M_0) *un réseau de Petri. Deux places* p_1, p_2 *sont* concurrentes, *ce que l'on note* $p_1 \parallel p_2$, ssi *il existe un marquage accessible* M *ayant à la fois un jeton dans* p_1 *et un jeton dans* p_2, *autrement dit,* ssi $\Re(\{p_1, p_2\})$.

La proposition suivante présente des propriétés utiles de cette relation pour ce chapitre.

Proposition 6.1 *La relation* $\parallel$ *est symétrique et quasi-réflexive*[1]*. Elle est réflexive* ssi *le réseau ne possède pas de place morte. Elle est antisymétrique* ssi *le réseau est une machine d'états.*

Preuve. La symétrie et la quasi-réflexivité découlent directement de la définition 6.1. Concernant la réflexivité, une place est concurrente avec elle-même *ssi* elle n'est pas morte (voir la définition 5.1). Une relation est à la fois antisymétrique et symétrique *ssi* son graphe est inclus dans sa diagonale. À noter que, dans le cas général, la relation $\parallel$ est ni transitive, ni intransitive. Ceci vaut également pour $\nparallel$, sa négation. Ceci implique que ni $\parallel$, ni $\nparallel$, ne sont des relations d'égalité ou d'inégalité. □

De cette relation $\parallel$, le problème de la détermination des paires de places concurrentes d'un réseau de Petri est introduit comme suit.

Définition 6.2 *Soit un réseau de réseau de Petri* (P, T, F, M_0) *ordinaire et sauf. Le* problème des places concurrentes *consiste en la détection de toutes les paires de places concurrentes.*

La définition suivante transpose la notion de places concurrentes aux unités d'un NUPN.

Définition 6.3 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN. Deux unités* u_1, u_2 *sont* concurrentes*, ce que l'on note* $u_1 \parallel u_2$*, si (i)* $\exists p_1, p_2 \in \mathsf{places}\,(u_1) \times \mathsf{places}\,(u_2) \mid (p_1 \parallel p_2)$ *; et (ii)* $\mathsf{disjoint}\,(u_1, u_2)$*.*

6.3 Travaux voisins

6.3.1 Définitions alternatives des places concurrentes

Dans certaines publications, cette relation $\parallel$ est mentionnée sous différents noms, tels que : *coexistence définie par marquages* [Jan84, sect. 9], *relation de concurrence* [Kov92, KE96, Kov00] [SY95] [GS06], *graphe de concurrence* [WKA+18], *graphe séquentiel* [SY95] [WWJ19], etc. Ces diverses définitions diffèrent par des détails, tels que les classes de réseaux de Petri considérés ou encore comment est définie la diagonale de la relation, c.-à-d. sous quelles conditions une place est-elle concurrente avec elle-même. Par exemple, [Jan84] définit $\parallel$ comme une relation irréflexive, alors que pour [Kov92], $p \parallel p$ est vraie *ssi* il existe un marquage accessible ayant deux jetons dans la place p (ceci n'est possible que pour des réseaux non-saufs). Un argument en faveur de notre définition 6.1 est que le problème des places concurrentes généralise le problème des places mortes. Des références aux notions de graphe de concurrence et de graphe séquentiel sont ultérieurement réalisées, (voir, respectivement, les définitions 7.1 et 7.4), ces représentations étant bien adaptées aux besoins du chapitre suivant.

La relation de concurrence $\parallel$ est définie par certains auteurs non pas sur P^2, mais sur $(P \cup T)^2$ [Bes79] [Pet79] [KE96] ou sur T^2 [PCR01].

1. Autrement dit, $\forall p_1, p_2 \in P, (p_1 \parallel p_2) \Rightarrow (p_1 \parallel p_1)$

Si nous distinguons les définitions « structurelles », reposant sur une lecture (plus ou moins directe) du quadruplet (P, T, F, M_0), des définitions « comportementales », reposant sur les marquages accessibles, nous pouvons qualifier notre définition de comportementale, contrairement aux définitions des relations *co* de [Bes79] et [Pet79] et aux définitions des matrices *place-conflit* [JM85, def. 2.2], *place-confluence* [JM85, def. 2.4] et *place-précédence* [JM85, def. 2.6], qui sont « structurelles ».

L'adoption d'une définition « comportementale » de la relation n'exclut en rien l'utilisation de règles ou d'algorithmes dits « structurels » pour obtenir une solution approchée du problème. Par exemple, les travaux de [Kov92, KE96, Kov00] reposent sur une abstraction de la notion de marquages accessibles et du tir de transitions, permettant de calculer un sur-ensemble de leur relation en un temps polynomial. Ces trois références seront abordées de manière plus détaillée dans la section 6.9, qui s'inspire de ces travaux.

Bien que la relation de concurrence puisse aisément se définir sur des réseaux de Petri ordinaires, mais non-saufs (voir, par exemple, [Kov92]), ceci a un intérêt limité. Considérons par exemple un réseau de Petri de la classe des machines à états, ne présentant aucune place morte, et dont les jetons du marquage initial sont tous localisés dans la même place. Dans ce réseau, si le marquage initial ne contient qu'un seul jeton, la relation $\|$ est simplement l'égalité entre deux places de P (deux places distinctes sont non-concurrentes), mais si le marquage initial contient au moins deux jetons, la relation $\|$ est le produit cartésien P^2 (toutes les places sont deux à deux concurrentes). Il serait possible de généraliser la relation $\|$, en calculant, pour chaque paire de places, le nombre maximal de jetons qu'elle serait susceptible de contenir, dans tout marquage accessible. Mais il s'agit d'un autre problème, qui généraliserait également celui de la détermination, pour un réseau donné, du plus petit k tel que le réseau soit k-sauf.

Enfin, une approche permettant d'accélérer le calcul des places concurrentes d'un réseau de Petri est introduite par [AC22] ; la section 6.13 abordera de manière plus détaillée ce travail.

6.3.2 Lien avec la logique temporelle

Les réflexions concernant l'utilisation de formules en logique temporelle pour calculer les places et les transitions mortes de la sous-section 5.3.3 s'appliquent également au calcul des places concurrentes. L'argumentation s'en trouve même renforcée ici, puisque le nombre de formules n'est plus linéaire en fonction de la taille du réseau en entrée, mais quadratique : plus précisément, le nombre de formules à traiter, qui était de $|P|$ pour les places mortes et de $|T|$ pour les transitions mortes, est de $|P|(|P|+1)/2$ pour les places concurrentes, ce qui fait que des réseaux comportant seulement une centaine de places peuvent en pratique devenir insolubles avec une telle approche.

À titre d'exemple, la collection de la section 3.8 comprend un réseau ayant 8,61 millions de paires de places.

Ainsi, la solution la plus opportune nous paraît être ici encore (et dans une plus grande

mesure) de traiter le problème des places concurrentes à un niveau supérieur, à l'aide d'une option dédiée « `-concurrent-places` », implantée dans les outils dédiés.

6.4 Complexité du problème

Cette section montre que la complexité théorique du problème des places concurrentes est identique à celle des problèmes des places mortes et des transitions mortes, autrement dit PSPACE-complet.

Proposition 6.2 *Le problème des places concurrentes est un sous-problème du* problème de couverture de marquages.

Preuve. En utilisant le prédicat $\Re$ de la définition 2.21 et la définition 6.1, l'ensemble des places concurrentes s'exprime ainsi $\{\{p_1, p_2\} \in P^2 \mid \Re(\{p_1, p_2\})\}$. □

Proposition 6.3 *Le problème des places mortes est un sous-problème du problème des places concurrentes.*

Preuve. La négation de la diagonale de la relation $\|$ est l'ensemble des places mortes (voir la définition 5.2). □

Proposition 6.4 *Le problème des places concurrentes est PSPACE-complet.*

Preuve. D'après la proposition 5.3, nous savons que le problème des places mortes est de même complexité que le problème de couverture de marquages, c.-à-d. PSPACE-complet. D'après les propositions 6.2 et 6.3, le problème des places concurrentes a une complexité encadrée par les problèmes des places mortes et de couverture de marquages. □

6.5 De la nécessité de solutions à valeurs ternaires

Nous réutilisons l'approche élaborée à la section 5.5, qui permet une réponse pragmatique à la complexité du problème des places concurrentes (voir la section 6.4). Lorsqu'il est impossible d'obtenir une solution *complète* (où les valeurs de toutes les paires de places sont connues), il est en effet préférable d'avoir une solution *partielle* (où l'on ne sait pas si certaines paires de places sont « non-concurrentes » ou « concurrentes ») que de ne pas avoir de solution du tout. Un algorithme efficace doit être en mesure d'éliminer, au cours d'un laps de temps donné, le plus grand nombre de valeurs inconnues possible.

Définition 6.4 *Les places concurrentes peuvent être représentées par un triplet* $(R_\perp, R_\top, R_\intercal)$*, tel que :*

- $R_\perp \subseteq \{\{p_1, p_2\} \in P \otimes P \mid p_1 \nparallel p_2\}$ *sont les paires de places déterminées non-concurrentes ;*
- $R_\top \subseteq \{\{p_1, p_2\} \in P \otimes P \mid p_1 \parallel p_2\}$ *sont les paires de places déterminées concurrentes ;*
- $R_\intercal \stackrel{\text{def}}{=} (P \otimes P) \setminus (R_\perp \cup R_\top)$ *est le prédicat donnant les paires de places dont on ne*

sait pas si elles sont concurrentes ou non.

Les éléments de ce triplet sont deux à deux disjoints et leur union est égale à $P \otimes P$*. Le triplet complètement indéterminé vaut* $(\varnothing, \varnothing, P \otimes P)$ *et le triplet correspondant à une solution complète vaut* $(\{\{p_1, p_2\} \in P \otimes P \mid p_1 \nparallel p_2\}, \{\{p_1, p_2\} \in P \otimes P \mid p_1 \parallel p_2\}, \varnothing)$.

L'interprétation des valeurs inconnues dans une solution partielle des places concurrentes dépend du contexte :

— Par exemple, les décompositions présentées dans la partie IV de ce document peuvent fonctionner avec des solutions incomplètes, en considérant les valeurs $\perp\!\!\!\perp$ comme des $\top$, ce qui revient à dire que deux places sont supposées concurrentes à moins que le contraire n'ait été prouvé. En raison de ces hypothèses conservatrices, les algorithmes produisant des valeurs $\perp$ sont clairement plus utiles que les algorithmes produisant des valeurs $\top$.

— Mais les algorithmes produisant des valeurs $\top$ pourraient néanmoins s'avérer propices à d'autres applications (notamment celles évoquées à la section 6.16), pouvant cette fois-ci aussi reposer sur des hypothèses optimistes.

6.6 Exploration des marquages accessibles

Les places concurrentes peuvent être déterminées par une exploration des marquages accessibles, semblable à celle présentée dans la section 5.6, sachant que les places de chaque marquage accessible sont deux à deux concurrentes. Si l'espace d'états peut être exhaustivement parcouru, nous obtenons une solution complète ; sinon, lorsque l'espace d'états est trop vaste pour être complètement exploré, la solution obtenue ne contient que des paires de places concurrentes ou inconnues, mais aucune paire de places non-concurrentes.

A priori, le seul raccourci algorithmique envisageable consisterait à arrêter l'exploration lorsque la solution calculée ne contient plus aucune valeur inconnue. Puisque tant que l'exploration est incomplète, aucune paire de places non-concurrentes n'est découverte, le raccourci ne serait effectif que lorsque toutes les paires de places non-concurrentes auraient au préalable été détectées par un autre algorithme, lancé en amont.

Or, en pratique, cette situation n'est susceptible d'arriver que dans des cas extrêmement favorables : (i) le problème des places non-concurrentes étant la négation du problème des places concurrentes, ils partagent la même complexité théorique et expérimentale ; (ii) les solutions calculées à la section 6.12 contiennent largement plus de paires de places non-concurrentes que de paires de places concurrentes. Il est donc illusoire d'espérer que ce raccourci puisse avoir un intérêt pratique, car cela nécessiterait d'avoir un autre algorithme, détectant l'intégralité des places non-concurrentes d'un nombre suffisamment significatif de modèles.

L'algorithme de cette section peut se formaliser ainsi.

```
1  R_⊤ ≔ M_0 ⊗ M_0

3  DELAI_EXPIRÉ ≔ ⊥          — booléen positionné à ⊤ en cas de dépassement de délai
4  BORNE ≔ +∞                — borne sur la profondeur d'itération
5  E ≔ ∅
6  E' ≔ {M_0}

8  repeat
9      for t ∈ T \ T_⊤ loop   — les transitions de T_⊤ sont mortes (voir la def. 5.6)
10         E'' ≔ marquages_où_la_transition_est_active (E', t) \ E
11         if E'' ≠ ∅ then
12             E' ≔ E' ∪ tirer_la_transition (E'', t)
13         end if
14     end loop
15     E' ≔ E' \ E
16     E ≔ E ∪ E'
17     BORNE ≔ BORNE − 1      — avec +∞ − 1 = +∞
18 until E' = ∅ or BORNE = 0 end repeat

20 R_⊤ ≔ R_⊤ ∪ {(M ⊗ M) | M ∈ E}

22 if E' = ∅ and ¬DELAI_EXPIRÉ then   — l'exploration est complète
23     R_⊥ ≔ R_⊥ ∪ R_⊤
24 end if
```

Algorithme C_1 – Exploration des marquages accessibles pour les places concurrentes

6.7 Règles structurelles

Nous étudions maintenant, de cette section à la section 6.9, le cas où l'exploration de l'espace d'états (voir la section 6.6) ne peut être exhaustivement réalisée, et proposons des algorithmes complémentaires, d'une plus faible complexité algorithmique, qui aident à réduire le nombre de paires inconnues dans la solution calculée.

Les deux règles suivantes exploitent les informations obtenues à propos des places et transitions non-mortes obtenues lors de la construction de l'ensemble des marquages accessibles.

Proposition 6.5 *Les places du marquage initial M_0 sont deux à deux concurrentes.*

Proposition 6.6 *Si une transition n'est pas morte, (i) ses places d'entrée sont deux à deux concurrentes et (ii) ses places de sortie sont deux à deux concurrentes.*

Nous appliquons maintenant l'algorithme A_2 (les règles structurelles) de la section 5.7, afin d'identifier (un sous-ensemble de) places et de transitions mortes. Connaissant cette

information, davantage de valeurs inconnues pourront être éliminées de la solution calculée.

Proposition 6.7 *(i) Une place non-morte est concurrente avec elle-même et (ii) une place morte n'est concurrente avec aucune place, y compris elle-même.*

Proposition 6.8 *Si une transition morte a deux places d'entrée distinctes, alors ces places ne sont pas concurrentes (y compris lorsqu'elles sont non-mortes).*

La règle suivante tire parti du fait que les réseaux de Petri sont supposés saufs.

Proposition 6.9 *Si une transition t (morte ou non) a une unique place d'entrée p, cette place est non-concurrente avec toute place de sortie de t différente de p.*

Preuve. Raisonnons par l'absurde : s'il existait un marquage accessible M contenant p et une quelconque place de sortie de t, ce réseau serait non-sauf, puisque t serait active dans M. À noter que si t est morte, alors p est également morte, le résultat suit alors directement de la proposition 6.7 (ii). □

La proposition suivante généralise la règle précédente, par un algorithme de fermeture transitive, à toute séquence de transitions ayant chacune une unique place d'entrée.

Proposition 6.10 *Pour tout chemin $p_1, t_1, p_2, t_2, ..., p_n, t_n, p_{n+1}$ tel que chaque transition t_i possède une unique place d'entrée p_i et au moins une place de sortie p_{i+1}, si les places p_1 et p_{n+1} sont distinctes, elles sont non-concurrentes.*

La dernière règle exploite la propriété unit-sauf, pour les réseaux qui sont unit-saufs par construction (voir la directive « `!unit_safe` », à la section 3.6).

Proposition 6.11 (d'après [Gar19, prop. 6]) *Si le réseau est un NUPN unit-sauf, toute paire de places distinctes appartenant à des unités non-disjointes ne sont pas concurrentes, ce qui s'écrit formellement* $(\forall p_1 \in P)\ (\forall p_2 \in P)\ (p_1 \neq p_2) \wedge \neg \mathsf{disjoint}\,(\mathsf{unité}\,(p_1), \mathsf{unité}\,(p_2)) \Rightarrow p_1 \nparallel p_2$. *En particulier, toute paire de places distinctes appartenant à la même unité est non-concurrente.*

Les règles structurelles des propositions 6.5 à 6.11 peuvent être appliquées par l'algorithme suivant :

1 $R_\top \coloneqq R_\top \cup (M_0 \otimes M_0)$ — *d'après la prop. 6.5*
2 $R_\top \coloneqq R_\top \cup \bigcup_{t \in T_\bot} ((\bullet t \otimes \bullet t) \cup (t^\bullet \otimes t^\bullet))$ — *d'après la prop. 6.6*
3 $R_\top \coloneqq R_\top \cup P_\bot$; $R_\bot \coloneqq R_\bot \cup (P_\top \otimes P)$ — *d'après la prop. 6.7*
4 $R_\bot \coloneqq R_\bot \cup \{\bullet t \mid (t \in T_\top) \wedge (|\bullet t| = 2)\}$ — *d'après la prop. 6.8*
5 $E \coloneqq \{\{p_1, p_2\} \in (\bullet t \otimes (t^\bullet \setminus \bullet t)) \mid (t \in T) \wedge (\bullet t = \{p_1\})\}$
7 — *la suite de cet algorithme suppose que le réseau est sauf*
8 $R_\bot \coloneqq R_\bot \cup E$ — *d'après la prop. 6.9*
9 **repeat**
10 $E' \coloneqq E$; $E \coloneqq E \cup \{\{p_1, p_3\} \mid (p_1 \neq p_3) \wedge (\exists p_2 \mid \{\{p_1, p_2\}, \{p_2, p_3\}\} \subseteq E)\}$
11 **until** $E' = E$ **loop**
12 $R_\bot \coloneqq R_\bot \cup E$ — *d'après la prop. 6.10*

```
14                                  — la suite de cet algorithme suppose que le réseau est unit-sauf
15  R_⊥ := R_⊥ ∪                                                    — d'après la prop. 6.11
16        {{p_1, p_2} ∈ (P ⊗ P)) | (p_1 ≠ p_2) ∧ (¬ disjoint (unité (p_1), unité (p_2)))}
```

Algorithme C_2 – Règles structurelles

6.8 Sous-approximation quadratique

Dans cette section, nous proposons un algorithme permettant de détecter davantage de paires de places concurrentes. Plus précisément, cet algorithme étend l'ensemble $R_\top$, calculé lors des phases précédentes, examinant toutes les transitions ayant une ou deux places d'entrée. Pour cela, il combine les propositions 5.10, 6.6 et 6.8 avec le résultat suivant :

Proposition 6.12 *Si deux places p_1 et p_2 sont distinctes et concurrentes, alors p_2 est également concurrente avec chaque place de sortie de toute transition t telle que ${}^\bullet t = \{p_1\}$.*

L'algorithme C_3 ci-dessous sauvegarde les paires de places ayant été prouvées concurrentes dans l'ensemble R'.

L'approximation effectuée par cet algorithme est dite *quadratique*, car tout marquage visité M est rendu abstrait et représenté par son ensemble de paires concurrentes $M \otimes M$, contrairement à l'algorithme A_4 de la section 5.9, qui effectue une approximation dite *linéaire*, car ne sauvegardant que les ensembles de places apparaissant dans au moins un marquage visité.

Cet algorithme effectue une sous-approximation, car il peut omettre d'explorer certaines paires de places concurrentes, qui sont pourtant accessibles.

```
1   R' := ∅
2   while R' ≠ R_⊤ loop
3         assert R' ⊂ R_⊤
4         let {p_1, p_2} = oneof (R_⊤ \ R')                  — éventuellement avec p_1 = p_2
5         R' := R' ∪ {{p_1, p_2}}

7         for t ∈ T \ T_⊤ | •t = {p_1, p_2} loop
8               assert (1 ≤ |•t| ≤ 2) ∧ (t ∉ T_⊤)   — d'après les prop 5.10 et 6.8 (contr.)
9               R_⊤ := R_⊤ ∪ (t• ⊗ (t• \ •t))                — d'après la prop. 6.6 (ii)
10        end loop

12        for t ∈ T \ T_⊤ | (•t = {p_1}) xor (•t = {p_2}) loop
13              assert (|•t| = 1) ∧ (p_1 ≠ p_2) ∧ (t ∉ T_⊤)        — d'après la prop. 5.10
14              R_⊤ := R_⊤ ∪ (({p_1, p_2} \ •t) ⊗ (t• \ •t))      — d'après la prop. 6.12
15        end loop
16  end loop
```

Algorithme C_3 – Sous-approximation quadratique

Proposition 6.13 *L'algorithme C_3 se termine.*

Preuve. Le nombre de paires de places est fini, donc $R_\top$ et R' sont finis. Puisque, $R' \subseteq R_\top$, à chaque itération, R' est strictement croissant pour l'inclusion. Conséquemment, la boucle « **while** » se termine au bout d'un nombre fini d'itérations. □

Proposition 6.14 *Cet algorithme est correct, c.-à-d. que toutes les paires de places ajoutées dans $R_\top$ sont concurrentes.*

Preuve. Au sein de la boucle « **while** », $R_\top$ n'est modifié qu'aux lignes 9 et 14. Puisque à la ligne 9 : $({}^\bullet t = \{p_1, p_2\}) \wedge (\exists M_1 \subseteq P \mid (M_0 \xrightarrow{*} M_1) \wedge ({}^\bullet t \subseteq M_1))$, alors : $\exists M_2 \subseteq P \mid (M_1 \xrightarrow{t} M_2) \wedge (t^\bullet \subseteq M_2)$. Puisque à la ligne 14 : $({}^\bullet t = \{p_1\}) \wedge (p_1 \neq p_2) \wedge (\exists M_1 \subseteq P \mid (M_0 \xrightarrow{*} M_1) \wedge ({}^\bullet t \cup \{p_2\} \subseteq M_1))$, alors : $\exists M_2 \subseteq P \mid (M_1 \xrightarrow{t} M_2) \wedge (t^\bullet \cup \{p_2\} \subseteq M_2)$. Conclusion : dans les deux cas, $\forall \{p_1, p_2\} \in R_\top, p_1 \parallel p_2$. □

L'algorithme C_3 généralise la découverte de places non-mortes, à titre de cas particulier du problème des places concurrentes (voir les lignes 17 à 31 de l'algorithme A_3, à la section 5.8).

Nous aurions pu appliquer la proposition 6.12 seule, au moyen d'une fermeture transitive [2], par exemple avec l'algorithme de Tarjan, mais cela aurait conduit à des résultats moins précis. En effet, la proposition 6.12 ne tient compte que des transitions n'ayant qu'une seule place d'entrée. L'algorithme C_3 donne, de son côté, (au moins) toutes les paires de places concurrentes des réseaux ayant au plus deux jetons dans tout marquage accessible à partir duquel une transition avec au moins deux places d'entrée est tirable.

6.9 Sur-approximation quadratique

Notre quatrième et dernier algorithme puise une partie de son inspiration dans les travaux de Kovalyov et d'Esparza, qui ont proposé divers algorithmes [Kov92, KE96, Kov00] ayant une complexité polynomiale. Ces différents algorithmes calculent tous un plus petit point fixe à partir de trois règles dérivées du jeu de jetons des réseaux de Petri. Ils permettent de produire une sur-approximation (autrement dit, un sur-ensemble) de la relation des places concurrentes, de laquelle nous pouvons extraire un sous-ensemble de $R_\perp$.

La sur-approximation de [Kov92, KE96, Kov00], notée $\|^A$, est définie au moyen des trois règles suivantes :

(i) $\forall p_1, p_2 \in P, (\{\!|\, p_1, p_2 \,|\!\} \leq M_0) \Rightarrow (p_1 \parallel^A p_2)$;

(ii) $\forall t \in T, \forall p_1, p_2 \in t^\bullet, (p_1 \neq p_2) \Rightarrow (p_1 \parallel^A p_2)$;

(iii) $\forall p \in P, \forall t \in T, (({p\} \otimes {}^\bullet t) \subseteq \|^A) \Rightarrow (({p\} \otimes t^\bullet) \subseteq \|^A)$.

L'abstraction $\|^A$ est exacte pour les réseaux *réguliers* [Kov00], cette classe comprend notamment les réseaux de Petri à choix libres étendus et vivants.

Pour rappel, la relation des places concurrentes de Kovalyov et Esparza emploie une

2. Cette fermeture transitive n'a aucun lien avec celle décrite par la proposition 6.10.

définition alternative des valeurs de la diagonale de $\|$ (voir la section 6.3) ; elle est irréflexive *ssi* le réseau de Petri est sauf. Cette particularité est reprise et généralisée à la section 6.14, dans la proposition 6.21, pour décider la propriété sauf.

Ces algorithmes calculent tous la même approximation $\|^A$, mais fonctionnent différemment : [Kov92] tire répétitivement toutes les transitions jusqu'à saturation de la relation calculée, avec un coût en $O(n^5)$, tandis que [KE96] et [Kov00] visitent les paires de places individuellement, avec un coût en $O(n^4)$.

Notre algorithme C_4, décrit ci-dessous, étend ces travaux dans quatre directions :

(i) Il ne suppose pas que le réseau ne possède aucune transition morte [3] : au lieu de toutes les « tirer » dès les départ, il les « tire » à la volée, seulement lorsque certaines conditions sur les paires de places d'entrée deviennent valides, à l'instar de l'algorithme pour les places et les transitions mortes de la section 5.9 ; il réutilise en outre $T_\top$, l'ensemble des transitions mortes déjà connues (obtenues par un autre algorithme, lancé au préalable).

(ii) Il ne suppose pas que le réseau ne présente aucune place morte [3] : nous enregistrons cette information dans la diagonale de la relation $\|$ (voir par exemple la proposition 6.7).

(iii) Afin d'obtenir des résultats plus précis (et plus rapidement), notre algorithme réutilise les ensembles de paires $R_\bot$ et $R_\top$ précalculés par d'autres algorithmes, alors que les algorithmes de Kovalyov et Esparza démarrent sans la moindre connaissance préalable de paires (non-) concurrentes.

(iv) Contrairement aux algorithmes de Kovalyov et Esparza, qui notent dans la diagonale de la relation $\|$ si une place serait susceptible d'avoir plusieurs jetons, C_4 nécessite que le réseau d'entrée soit sauf, mais utilise cette hypothèse pour produire des résultats plus précis, en écartant les marquages à partir desquels le tir d'une transition mènerait vers un marquage non-sauf : autrement dit, les marquages de $\{(\bullet t \cup \{p\} \cup M) \subseteq P \mid (t \in T) \wedge (p \in (t^\bullet \setminus \bullet t)) \wedge (M \subseteq P)\}$ sont écartés.

Nous formalisons maintenant la sur-approximation (que nous appelons *quadratique* en raison de son coût mémoire) sous-jacente.

Définition 6.5 *Soient deux ensembles R et R' contenant des paires de places, une transition t et un marquage M. Nous notons $R \overset{t}{\Longrightarrow} R'$* ssi *nous avons $\bullet t \otimes \bullet t \subseteq R$ et $R' = R \cup (t^\bullet \otimes t^\bullet) \cup \{\{p\} \otimes t^\bullet \mid (p \in P \setminus \bullet t) \wedge (\{p\} \otimes \bullet t \subseteq R)\}$. Nous notons $R \overset{*m}{\Longrightarrow} M$* ssi *il existe un R' tel que $R \overset{*}{\Longrightarrow} R'$ et $M \otimes M \subseteq R'$.*

Contrairement à la relation $\overset{t}{\longrightarrow}$ définie entre deux marquages (voir la définition 2.19), cette relation $\overset{t}{\Longrightarrow}$ est définie entre deux ensembles de paires de places. Avec $\overset{t}{\longrightarrow}$, l'espace d'états est l'ensemble des marquages accessibles M, tandis qu'avec $\overset{t}{\Longrightarrow}$, l'espace d'états (une fois rendu abstrait) est l'union de tous les ensembles de paires $M \otimes M$, pour tout marquage « accessible » M. D'un point de vue plus algorithmique, le calcul de la relation $\overset{t}{\Longrightarrow}$ est

3. D'après la section 5.12, au moins 15,9 % (respectivement 16,2 %) des modèles ont des transitions (respectivement places) mortes : ces deux premières améliorations sont ainsi cruciales.

identique à celui de la relation $\xrightarrow{t}$, à ceci près que la structure de données représentant l'espace d'états devient un filtre de Bloom.

Proposition 6.15 *Dans un réseau de Petri sauf, si un marquage M est accessible depuis le marquage initial M_0 (autrement dit, $M_0 \xrightarrow{*} M$), alors $M_0 \otimes M_0 \overset{*m}{\Longrightarrow} M$.*

Preuve. Ceci se montre aisément par induction sur les séquences de tir de transitions depuis $M_0 \otimes M_0$. □

Pour les réseaux saufs, l'algorithme C_4 donne des résultats plus précis qu'une application des algorithmes de Kovalyov et Esparza et de l'algorithme pour les places et transitions mortes de la section 5.9 réunis, y compris en les appliquant de manière itérée jusqu'à saturation.

Notre algorithme démarre des paires de places contenues dans l'union de l'abstraction du marquage initial $M_0 \otimes M_0$ et de $R_\top$ (contenant les paires concurrentes déjà connues). Il explore, en utilisant deux variables R' et R'', l'espace d'états de toutes les paires pouvant être accédées via le tir de toutes les transitions (non-mortes) par la relation $\overset{*}{\Longrightarrow}$. À la terminaison, toutes les paires n'ayant pas été explorées sont non-concurrentes et peuvent donc être ajoutées à $R_\bot$.

Afin d'améliorer la qualité de la sur-approximation, les transitions de $T_\top$, que l'on sait déjà mortes, ne sont jamais tirées et les paires de places de $R_\bot$, sont systématiquement omises de l'espace d'états, car non-concurrentes.

Pour accélérer l'algorithme, nous réutilisons le compteur $t \mapsto c[t]$ de la section 5.9, qui comptabilise désormais combien de paires de $({}^\bullet t \times {}^\bullet t)$ ne sont toujours pas déclarées concurrentes ; une transition est considérée active lorsque son compteur devient nul (le compteur des lignes 19 et 27 pourrait être remplacé par un simple test $({}^\bullet t \times {}^\bullet t) \subseteq R'$).

```
1  function tirable (M, t, R) is
2          return (M ⊗ •t ⊆ R) ∧ ((M ⊗ •t) ∩ R⊥ = ∅)
3  end function

5  R' ≔ ∅ ; R'' ≔ (M0 ⊗ M0) ∪ R⊤
6  T⊤ ≔ T⊤ ∪ {t ∈ T | ((•t ⊗ •t) ∩ R⊥ ≠ ∅) ∨ ((t• ⊗ t•) ∩ R⊥ ≠ ∅)}
7  for t ∈ T \ T⊤ loop
8          c[t] ≔ |•t| × (|•t| + 1)/2
9  end loop

11 while R' ≠ R'' loop
12         assert (R' ⊂ R'') ∧ (R'' ∩ R⊥ = ∅)
13         assert (∀t ∈ T \ T⊤) c[t] = |(•t ⊗ •t) \ R'|
14         let {p1, p2} = oneof (R'' \ R')          — éventuellement avec p1 = p2
15         R' ≔ R' ∪ {{p1, p2}}

17         for t ∈ T \ T⊤ | {p1, p2} ⊆ •t loop
18                 c[t] ≔ c[t] − 1
```

19 **if** $c[t] = 0$ **then**
20 **for** $p \in (P \setminus {}^\bullet t) \cup t^\bullet \mid \mathsf{tirable}\,(\{p\}, t, R'')$ **loop**
21 **assert** $(p \notin {}^\bullet t \setminus t^\bullet) \wedge (M_0 \otimes M_0 \overset{*m}{\Longrightarrow} \{p\} \cup {}^\bullet t)$
22 $R'' \coloneqq R'' \cup (\{p\} \otimes (t^\bullet \setminus {}^\bullet t))$
23 **end loop**
24 **end if**
25 **end loop**

27 **for** $t \in T \setminus T_\top \mid (c[t] = 0) \wedge ((p_1 \in {}^\bullet t)$ **xor** $(p_2 \in {}^\bullet t)) \wedge$
28 $\mathsf{tirable}\,((\{p_1, p_2\} \setminus {}^\bullet t), t, R'')$ **loop**
29 **assert** $(|\{p_1, p_2\} \setminus {}^\bullet t| = 1) \wedge (M_0 \otimes M_0 \overset{*m}{\Longrightarrow} \{p_1, p_2\} \cup {}^\bullet t)$
30 $R'' \coloneqq R'' \cup ((\{p_1, p_2\} \setminus {}^\bullet t) \otimes (t^\bullet \setminus {}^\bullet t))$
31 **end loop**
32 **assert** $(\forall t \in T \setminus T_\top)\ (\forall p \in (P \setminus {}^\bullet t) \cup t^\bullet)\ (c[t] = 0) \wedge \mathsf{franchir}\,(\{p\}, t, R')$
33 $\Rightarrow (\{p\} \otimes t^\bullet \subseteq R'')$
34 **end loop**

36 **assert** $(R' = R'') \wedge (R' \cap R_\perp = \varnothing) \wedge (R_\perp \subseteq (P \otimes P) \setminus R')$
37 $R_\perp \coloneqq (P \otimes P) \setminus R'$

Algorithme C_4 – Sur-approximation quadratique

Proposition 6.16 *L'algorithme C_4 se termine.*

Preuve. Le nombre de paires de places est fini, donc R' et R'' sont finis. Puisque, $R' \subseteq R''$, à chaque itération, R' est strictement croissant pour l'inclusion. Conséquemment, la boucle « **while** » se termine au bout d'un nombre fini d'itérations. □

Proposition 6.17 *Cet algorithme est correct, autrement dit, toutes les paires de places ajoutées dans $R_\perp$ sont non-concurrentes.*

Preuve. Nous allons démontrer, par induction, que toutes les paires de places concurrentes sont dans R' à la ligne 37. Base : après la ligne 5, $(M_0 \otimes M_0) \subseteq R'$.

Induction : soit $t \in T$, $M_1 \subseteq P$ et $M_2 \subseteq P$ tels que $(M_0 \xrightarrow{*} M_1 \xrightarrow{t} M_2) \wedge ((M_1 \otimes M_1) \subseteq R')$.

La transition t n'est pas morte, $({}^\bullet t \otimes {}^\bullet t) \subseteq R'$ et lorsque la boucle « **while** » se termine, la ligne 18 a été exécutée $|{}^\bullet t|(|{}^\bullet t| + 1)/2$ fois, ainsi, on a $c\,[t] = 0$; de ce fait, la branche « **then** » de la ligne 19 a été empruntée (une seule fois) pour t : la boucle des lignes 20 à 23 nous donne $(t^\bullet \otimes t^\bullet) \subseteq R'$.

Soit $p \in M_1 \setminus {}^\bullet t$. Puisque $p \in M_2$ et M_2 est accessible, $(\{p\} \otimes t^\bullet) \not\subseteq R_\perp$. Par conséquent, lorsque la boucle des lignes 20 à 23 se termine pour t, si $(\{p\} \otimes t^\bullet) \not\subseteq R''$, cela signifie que juste avant d'emprunter la branche « **then** » pour t, nous avions $(\{p\} \otimes {}^\bullet t) \not\subseteq R''$. Puisqu'à la ligne 36, nous avons $(\{p\} \otimes {}^\bullet t) \subseteq R'$, il existe nécessairement une itération de la boucle « **while** » telle qu'à la ligne 15, l'inclusion mentionnée ci-dessus soit vraie et que $\{p_1, p_2\} \in (\{p\} \otimes M_1)$. À cette itération, lorsque la boucle des lignes 27 à 31 se termine,

nous avons $(\{p\} \otimes t^\bullet) \subseteq R''$. □

Proposition 6.18 *Cet algorithme préserve toutes les paires préexistantes dans* $R_\perp$.

Preuve. Nous devons prouver que toutes les paires préexistantes de $R_\perp$ sont préservées, autrement dit, à la ligne 36, $R_\perp \cap R'' = \varnothing$. À la ligne 5, cela est vrai. Ensuite, des paires sont ajoutées à R'' aux lignes 22 et 30. À la ligne 22, si $p_3 \in t^\bullet$, on sait déjà que $(t^\bullet \otimes t^\bullet) \cap R_\perp = \varnothing$, sinon d'après la ligne 5, nous aurions $t \in T_\top$, et par conséquent la boucle des lignes 20 à 23 ne serait jamais exécutée pour t. Dans les autres cas, les conditions de garde des lignes 22 et 30 empêchent clairement l'ajout à R'' de toute paire déjà dans $R_\perp$. □

Proposition 6.19 *Cet algorithme est idempotent, autrement dit, l'appliquer plusieurs fois de suite ne change pas les résultats produits.*

Preuve. Par induction sur les séquences de tir de transitions depuis R''. □

Remarquons que C_4 itère sur toutes les paires de places contenues au préalable dans $(M_0 \otimes M_0) \cup R_\top$, qui peut être grand lorsque $R_\top$ ne contient au préalable que peu de paires de places. Pour éviter d'itérer sur toutes les paires déjà dans $R_\top$, une implantation efficace peut essayer de « tirer », selon $\xrightarrow{*m}$, toutes les transitions (une seule fois), puis de continuer la visite des paires de places nouvellement découvertes, comme le fait actuellement C_4.

6.10 Ordre des algorithmes

Les algorithmes présentés au sein des sections 6.6, 6.7, 6.8, et 6.9 sont respectivement nommés C_1, C_2, C_3 et C_4. Après avoir exécuté un de ces algorithmes, l'appliquer à nouveau immédiatement ne décroîtra jamais le nombre de valeurs inconnues (voir la proposition 6.19 pour C_4, le constat est immédiat pour les autres algorithmes).

Une analyse des apports et coûts potentiels de ces algorithmes met en évidence les points suivants : (i) C_1 ne dispose pas de raccourci algorithmique (voir la section 6.6) ; (ii) C_2 peut bénéficier des informations apportées par C_1 pour obtenir de meilleurs résultats ; (iii) C_3 et C_4 bénéficient tous les deux des informations apportées par C_1 et C_2 pour obtenir de meilleurs résultats.

Ceci, ainsi que nos expérimentations, suggère que $(C_1; C_2; C_3; C_4)$ serait le meilleur ordre d'application pour ces algorithmes. Enfin, cette séquence d'exécution s'arrête dès lors que le nombre de valeurs inconnues devient nul.

6.11 Implantations logicielles et formats de fichiers

Nous avons implanté nos algorithmes dans deux outils distincts : CÆSAR.BDD et ConcNUPN (voir la section 3.7). L'option permettant le calcul des places concurrentes est, pour ces deux outils, « `-concurrent-places` ». CÆSAR.BDD implante tous les algorithmes évoqués à la section 6.10 et les exécute selon l'ordre établi dans cette section.

ConcNUPN dispose d'une implantation des algorithmes C_3 et C_4.

Les outils CÆSAR.BDD et ConcNUPN acceptent des modèles en entrée au format NUPN, qui peuvent être produits à partir de modèles au format normalisé PNML, à l'aide d'une traduction automatisée (voir la section 3.6).

Les places concurrentes peuvent aussi se représenter sous la forme d'une *matrice des places concurrentes*, qui est symétrique (voir la proposition 6.1) donnant, pour chaque paire de places, sa valeur dans $\{\bot, \top, \mathbb{I}\}$ (voir la section 6.5).

Puisque, dans le format de fichier « .nupn », les places forment un ensemble ordonné, cette matrice se concrétise sous la forme d'un fichier texte (voir [Gar20, sect. 8 et annexe A]), constitué d'autant de lignes qu'il y a de places dans le réseau. Plus précisément, le caractère associé à la paire de places p_1 et p_2 telle que $p_1 \leq p_2$ est localisé au sein de la $\#p_1^{\text{ème}}$ colonne, de la $\#p_2^{\text{ème}}$ ligne. Lorsque les places p_1 et p_2 sont identiques (cas de la diagonale) ou que les unités les contenant (directement) sont disjointes, la cellule correspondante dans la matrice peut posséder l'une des valeurs suivantes : (i) « `0` » pour $\bot$; (ii) « `.` » pour $\mathbb{I}$; ou (iii) « `1` » pour $\top$. Sinon, les places p_1 et p_2 sont distinctes et appartiennent à des unités non-disjointes. Cette information est alors retranscrite dans la matrice par l'introduction de nouveaux caractères.

- Lorsque $\mathsf{unité}(p_1) \sqsubset \mathsf{unité}(p_2)$ (respectivement $\mathsf{unité}(p_1) \sqsupset \mathsf{unité}(p_2)$ et $\mathsf{unité}(p_1) = \mathsf{unité}(p_2)$), la valeur $\bot$ est retranscrite dans la matrice par « `<` » (respectivement « `>` » et « `=` »).
- La valeur $\mathbb{I}$ est quant à elle retranscrite par « `[` » (respectivement « `]` » et « `~` »).
- Enfin, la valeur $\top$ reste retranscrite par « `1` », car elle correspond au cas où le NUPN enfreindrait la propriété unit-sauf : la structure NUPN n'aurait alors plus aucun intérêt.

Cette matrice pouvant être de grande taille pour certains réseaux, l'algorithme de compression par plages de la section 5.11 est réutilisé tel quel : puisque les lignes de la matrice sont séparées par des retours chariots, la compression par plages de la matrice complète équivaut à une compression individuelle, de chaque ligne, par plages. Cette compression permet en pratique une réduction totale de la taille des matrices symétriques d'un facteur de 45,95, pour la collection de la section 3.8.

6.12 Résultats expérimentaux

Les expérimentations de ce chapitre proviennent de [BG21a], où CÆSAR.BDD a été utilisé, en janvier 2021, pour calculer les matrices des places concurrentes de la collection de la section 3.8, qui contenait alors 13 116 modèles. Les 3 084 modèles ajoutés depuis ne changent guère ses caractéristiques : les chiffres de [BG21a, Table 1] sont très proches de ceux des tables 3.3 et 3.4.

Nos expérimentations ont été réalisées sur des serveurs de la grille de calcul Grid'5000,

équipés de deux processeurs Intel Xeon E5-2660 v2 (2,2 GHz), de 128 Go de RAM et fonctionnant avec la distribution Linux Debian 10. Sur chacun de ces ordinateurs nous avons décidé de n'exécuter que deux expériences en parallèle, afin de bénéficier de la fréquence d'horloge la plus élevée du processeur et d'éviter de diviser la mémoire vive entre un trop grand nombre de tâches.

Le temps alloué à l'exploration symbolique des marquages accessibles de l'algorithme C_1 est paramétré par un délai (maximal) d'exécution t. Le calcul des places concurrentes pour les modèles de grande taille pouvant se révéler bien plus long que le calcul des places mortes ou des transitions mortes, chaque exécution de CÆSAR.BDD est bornée par un délai (maximal) d'exécution de 4 000 secondes qui, pour les diverses valeurs de t, a été atteint pour au plus 0,82 % de nos modèles.

valeur de t	0	5	10	15	30	45	60
% matrices complètes	51.0	91.6	92.2	92.5	93.0	93.6	94.0
% valeurs inconnues	45.0	44.7	44.7	44.4	44.4	43.7	43.7
% remplissage matrices	81.6	96.3	96.6	96.8	97.0	97.1	97.2

valeur de t	60	120	180	240	300	360	420
% matrices complètes	94.0	94.2	94.4	94.5	94.6	94.7	94.7
% valeurs inconnues	43.7	43.7	43.6	43.6	43.6	43.6	43.6
% remplissage matrices	97.2	97.3	97.4	97.4	97.4	97.5	97.5

TABLE 6.1 – Résultats expérimentaux pour les places concurrentes

Les trois métriques des tables 5.1 et 5.2, « % matrices complètes », « % valeurs inconnues » et « % remplissage matrices », sont reprises pour la table 6.1, en les adaptant pour passer des vecteurs vers les matrices symétriques (les vecteurs ont $|P|$ ou $|T|$ éléments, alors que les matrices en ont $(|P|+1)|P|/2$).

La première métrique, représentée par la figure 6.1, montre que 94,0 % des modèles peuvent être intégralement résolus avec $t = 60$, ce qui est très légèrement inférieur aux pourcentages obtenus dans les tables 5.1 et 5.2. Mais cette différence s'explique du fait que le problème des places concurrentes requiert en général davantage de temps processeur que les problèmes des places et des transitions mortes.

La seconde métrique, représentée par la figure 6.2, décroît plus lentement que dans les tables 5.1 et 5.2, et paraît se stabiliser à un pourcentage bien plus élevé de valeurs inconnues, ce qui peut s'expliquer par la taille quadratique de quelques matrices de très grande taille et incomplètes.

Cependant, pour la majeure partie des modèles, la troisième métrique, représentée par la figure 6.3, atteint un haut taux moyen de valeurs connues, similaire à ceux des tables 5.1 et 5.2.

Les résultats pour $t = 0$ sont meilleurs dans le cas des places concurrentes que dans le cas des places mortes, ceci pouvant s'expliquer par la présence bénéfique de la structure NUPN,

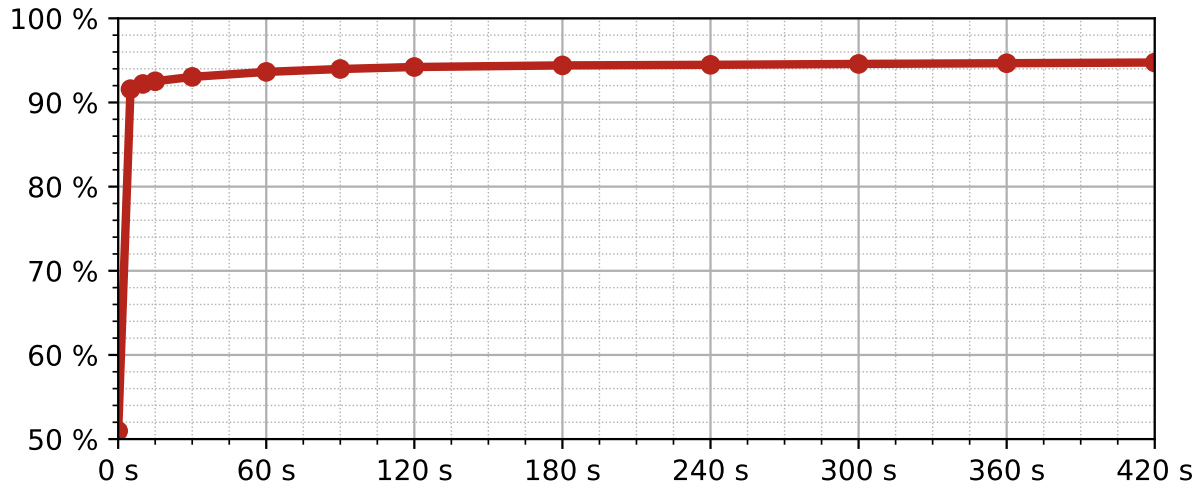

FIGURE 6.1 – Pourcentage de matrices complètes en fonction de t

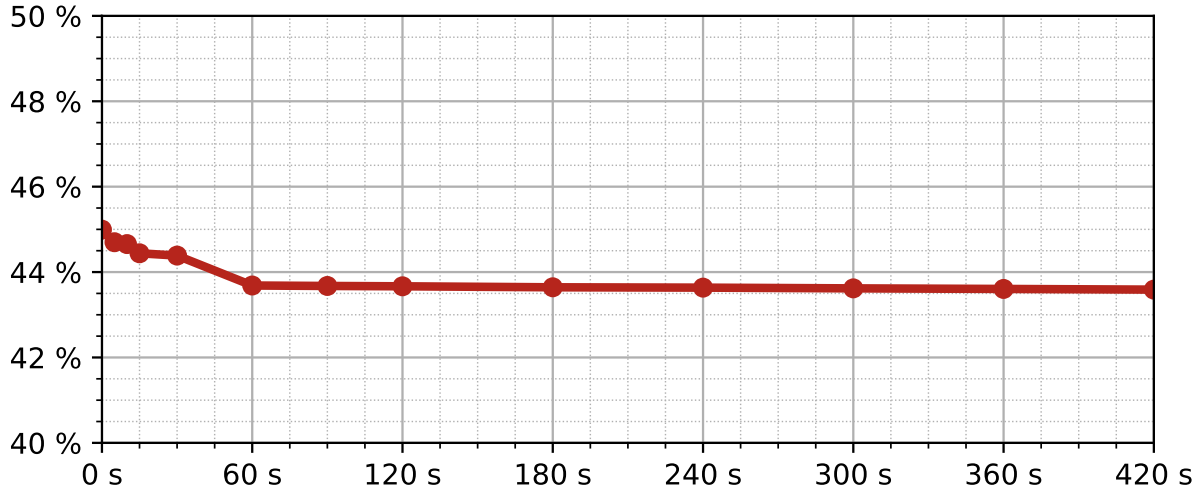

FIGURE 6.2 – Taux global de valeurs inconnues dans les matrices en fonction de t

mais aussi par la complexité supérieure des algorithmes structurels employés.

Des mesures complémentaires indiquent que :

— Sur l'ensemble des 2.66×10^{10} paires de places provenant des modèles où (au moins) une matrice a été obtenue : 4,0 % sont concurrentes, 67,0 % sont non-concurrentes, les autres demeurant inconnues.

— Les algorithmes C_2, C_3 et C_4 peuvent à eux seuls (autrement dit, pour $t = 0$) compléter intégralement 51,0 % des modèles, mais dès que l'algorithme C_1 est mis à contribution, ses résultats sont tels que, sur les modèles pouvant être totalement explorés avec les diagrammes de décision binaires, les algorithmes C_2 à C_4 ne

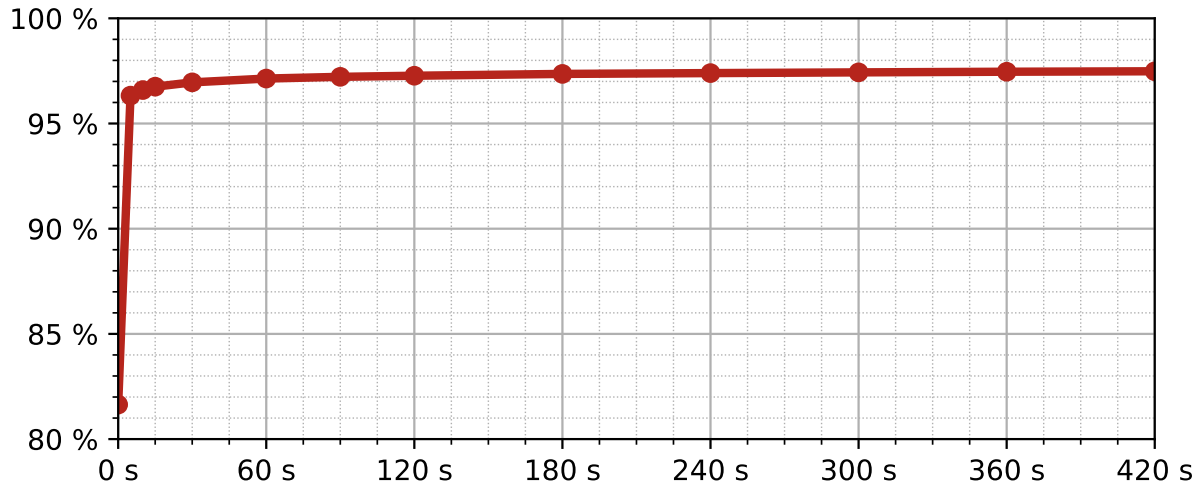

FIGURE 6.3 – Taux de remplissage moyen des matrices en fonction de t

contribuent qu'à hauteur de 1 % au nombre de matrices complètes.

— En revanche, sur les modèles de grande taille ne pouvant être entièrement explorés à l'aide de diagrammes de décision binaires, les algorithmes C_2, C_3 et C_4 jouent un rôle nettement accru : pour $t = 420$, l'algorithme C_1 enlève 33,8 % des valeurs inconnues, les algorithmes suivants enlevant 22,5 % d'inconnues (21,7 % par C_2, 0,4 % par C_3 et enfin 0,4 % par C_4). Ces 0,4 % ont plus d'importance que celle perçue de prime abord : ce sont les valeurs les plus difficiles à déterminer, les pourcentages ne montrant que les apports différentiels des algorithmes. Par exemple, la section 6.14 montre (indirectement) que C_4 peut à lui seul déterminer la quasi-totalité des valeurs apportées par la proposition 6.11 de C_2, dans la collection de la section 3.8.

— Avec $t = 60$, tous les réseaux ayant moins de 66 places, 64 transitions et 256 arcs peuvent être complètement traités.

— Avec $t = 60$, il suffit, en moyenne, de 2,6 secondes pour traiter les modèles ayant une matrice complète, et 301 secondes suffisent pour traiter les modèles ayant une matrice partielle.

6.13 Incorporation des réductions polyédriques

Une approche complémentaire, visant à faciliter la détermination de la relation des places concurrentes pour des réseaux de Petri de grande taille est introduite par [AC22]. Elle repose sur la simplification d'un NUPN en entrée à l'aide de réductions structurelles (suppression de places redondantes et fusion de places similaires), mettant en œuvre une *abstraction polyédrique* [ABD21, ABD22], appliquée en vue de produire un NUPN dit *réduit*, qui est associé à un ensemble d'équations linéaires liant les places de ce réseau réduit avec les

places du réseau originel. Ces équations permettent de reconstruire les marquages accessibles du réseau d'entrée depuis les marquages accessibles du réseau réduit.

Cette approche est implantée dans un outil, intitulé Kong [4] (voir la section 3.7). Pour calculer les réductions de réseaux, cet outil invoque en interne l'outil Reduce de la boîte à outils Tina [5]. Kong calcule ensuite les places concurrentes sur le réseau réduit, en invoquant l'outil CÆSAR.BDD, qui implante les algorithmes de ce chapitre. Kong peut aussi calculer l'ensemble des places mortes et déterminer si un marquage donné est accessible.

Les expérimentations de cette section proviennent de [ABG23] et ont été réalisées à partir d'une collection de 850 réseaux extraits de l'édition 2022 du MCC et exempte de doublons isomorphes. Notons que 81,41 % de ces réseaux ont une structure NUPN non-triviale.

À partir de cette collection de 850 NUPN, nous avons évalué les performances individuelles de CÆSAR.BDD et de Kong, en réalisant douze séries d'expérimentations (totalisant 18 504 calculs de matrices) :

- en exploitant la structure NUPN (résultats suffixés par « NUPN »), ou en la retirant (résultats suffixés par « PT ») ;
- en positionnant un temps imparti de dix minutes (correspondant à la patience d'un être humain attendant le résultat calculé par un outil) ou d'une heure (correspondant à la durée accordée par le MCC à chaque examen) ;
- en allouant à l'algorithme C_1 de CÆSAR.BDD (autrement dit, à la phase d'exploration du graphe des marquages accessibles) 25 %, 50 % ou 75 % de la valeur correspondant au délai d'attente.

résultats observés	outil	temps imparti	
		10 min	1 heure
matrices complètes	Kong (NUPN)	64,5 %	67,6 %
	CÆSAR.BDD (NUPN)	62,7 %	66,6 %
	Kong (PT)	59,6 %	63,4 %
	CÆSAR.BDD (PT)	52,5 %	59,9 %
remplissage matrices	Kong (NUPN)	74,4 %	77,4 %
	CÆSAR.BDD (NUPN)	72,1 %	76,2 %
	Kong (PT)	69,8 %	73,7 %
	CÆSAR.BDD (PT)	62,1 %	68,1 %
temps de calcul moyen	Kong (NUPN)	3 min 14 s	16 min 40 s
	CÆSAR.BDD (NUPN)	3 min 36 s	17 min 44 s
	Kong (PT)	3 min 39 s	19 min 16 s
	CÆSAR.BDD (PT)	4 min 30 s	23 min 28 s

Table 6.2 – Bilan des performances de CÆSAR.BDD et de Kong

4. github.com/nicolasAmat/Kong
5. projects.laas.fr/tina/manuals/reduce.html

La table 6.2 synthétise les résultats de ces expériences, en mettant l'accent sur trois types de résultats : (i) le pourcentage de matrices complètes générées par un outil en respectant un délai donné (correspond à « % matrices complètes » à la section 6.12) ; (ii) le pourcentage moyen de valeurs connues ($\bot$ ou $\top$) dans les 850 matrices générées (correspond à « % remplissage matrices » à la section 6.12) et (iii) le temps moyen nécessaire à l'outil pour générer ces matrices.

Dans ce tableau, le temps de calcul alloué à l'algorithme C_1 est fixé à 50 % du délai d'attente (soit cinq et trente minutes) ; nos expériences montrent que lorsque le temps de calcul alloué à C_1 augmente, le nombre de matrices complètes croît légèrement (de 3,8 % au plus), mais que le taux de remplissage des matrices diminue (de -6,3 % au plus), dans la mesure où l'algorithme C_1 consomme la majeure partie du temps, ce qui entrave l'application des approches ultérieures (algorithmes C_2, C_3 et C_4).

Ces chiffres mettent en évidence l'importance pratique de la structure NUPN, dans la mesure où l'exploration symbolique des marquages accessibles devient plus exigeante une fois que les informations relatives aux unités ont été retirées (fait évoqué dans la section 3.4).

Ces chiffres confirment également l'apport des réductions structurelles, y compris lorsqu'une structure NUPN est déjà présente. Ceci est d'ailleurs une amélioration de [ABG23] par rapport à [AC22] : la structure NUPN est préservée lors de l'application des réductions structurelles, ce qui donne des résultats impossible à obtenir lorsque ces deux techniques sont appliquées isolement.

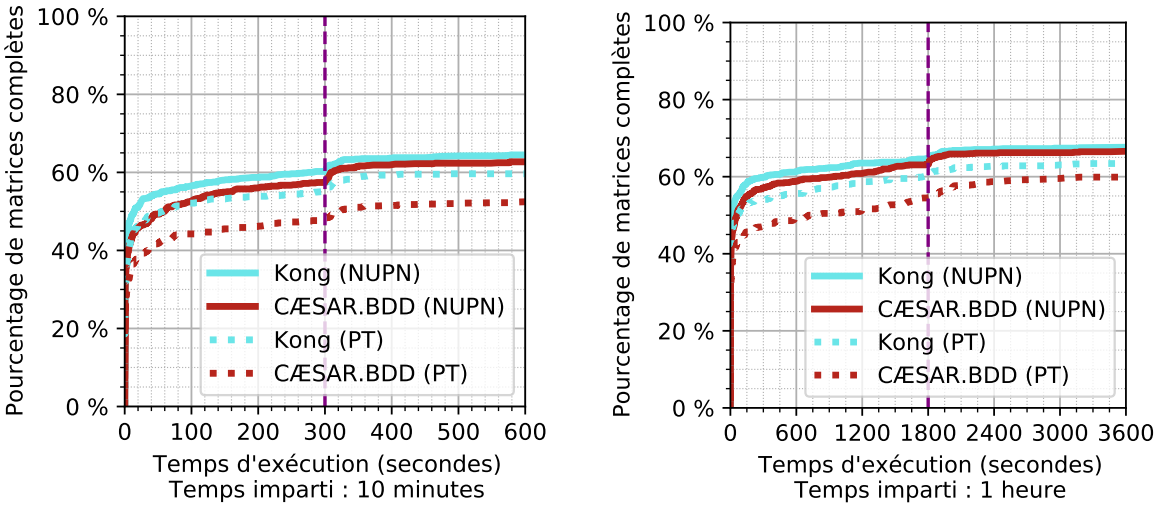

FIGURE 6.4 – Pourcentages de matrices complètes

Les figures 6.4 et 6.5 apportent des informations sur la vitesse d'exécution des outils, en montrant la croissance, en fonction du temps écoulé, du nombre de solutions complètes générées et du taux de remplissage des matrices. La ligne verticale en pointillés violets correspond à la valeur de 50 % du délai d'attente attribuée à l'algorithme C_1 (cinq et trente

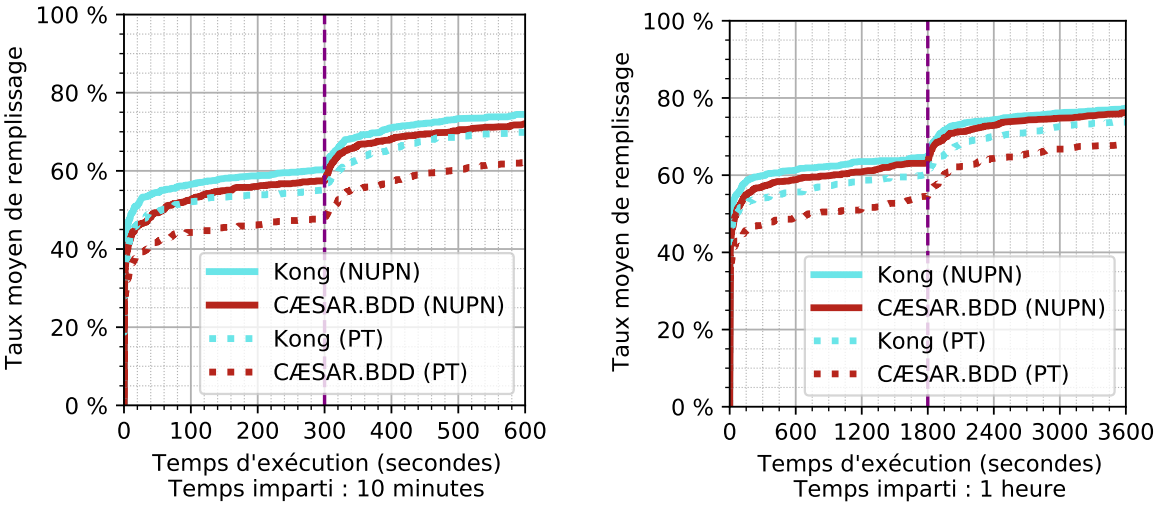

FIGURE 6.5 – Taux moyens de remplissage des matrices

minutes, respectivement).

Ces figures montrent de fortes progressions pendant les premières minutes, qui s'amenuisent ensuite progressivement, suivies d'un rebond à l'expiration du délai imparti pour l'exploration des marquages accessibles, à mesure que les autres approches (algorithmes C_2 à C_4) sont exécutées et montrent leur efficacité. Malgré l'impression donnée par ces chiffres, il n'y a pas d'asymptote horizontale, car un temps processeur infini devrait permettre aux résultats d'atteindre 100 %.

Ces figures montrent que plus l'exploration de l'espace d'états prend un temps considérable, plus la structuration NUPN est importante et plus les réductions structurelles de KONG sont efficaces.

L'association de CÆSAR.BDD et KONG a permis de déterminer des solutions complètes pour environ 55 % des réseaux en moins d'une minute et 65 % en moins de dix minutes.

6.14 Décision des propriétés sauf et unit-sauf

À partir d'une solution pour les places concurrentes, il est possible, comme indiqué à la section 6.1, de définir des conditions suffisantes pour déterminer si un NUPN (respectivement un réseau de Petri) donné est unit-sauf, ou non (respectivement sauf, ou non-sauf).

Le problème de détermination de la propriété sauf est PSPACE-complet [JLL77, corollaire 3.4], tout comme le problème de couverture de marquages pour les réseaux saufs [CEP95, Th. 15] et conséquemment, la détermination de la propriété unit-sauf.

L'approche de cette section n'a pas recours à un parcours des marquages accessibles, que ce soit pour calculer la matrice des places concurrentes, ou pour établir les conditions

suffisantes sur cette matrice. Elle emploie des algorithmes de complexité polynomiale (en temps et en mémoire), mais qui, en contrepartie, peuvent échouer et donner des réponses indéterminées pour certains modèles.

Cette section repose sur l'outil prototype ConcNUPN (voir les sections 3.7 et 6.11) pour l'implantation et les expérimentations de cette approche.

La proposition suivante est une condition suffisante pour montrer qu'un réseau de Petri soit non-sauf.

Proposition 6.20 *Soit (P, T, F, M_0) un réseau de Petri ordinaire. S'il existe une transition t non-morte ayant une unique place d'entrée, concurrente avec l'une des places de $p \in t^\bullet \setminus {}^\bullet t$ (une place p gagnant un jeton suite au franchissement de t), alors le réseau est non-sauf.*

Preuve. On a : $\exists p \in t^\bullet \setminus {}^\bullet t \mid \Re(\{p\} \cup {}^\bullet t)$, d'où le résultat. □

La proposition suivante est une condition suffisante pour montrer qu'un réseau de Petri soit sauf. Elle est inspirée du traitement de la diagonale de la sur-approximation $\|^A$ de Kovalyov (voir la section 6.9).

Proposition 6.21 *Soit (P, T, F, M_0) un réseau de Petri ordinaire. Si, pour toute transition $t \in T$ non-morte et toute place $p \in t^\bullet \setminus {}^\bullet t$ (autrement dit, toute place p gagnant un jeton suite au franchissement de t), s'il existe une place d'entrée de t non-concurrente avec p, alors, le réseau de Petri est sauf.*

Preuve. Par contraposée : si le réseau de Petri n'est pas sauf, il existe alors un marquage accessible M_1 sauf, une transition t active dans M_1 et un marquage accessible M_2 non-sauf tels que $M_0 \xrightarrow{*} M_1 \xrightarrow{t} M_2$. Soit une place p ayant plusieurs jetons dans M_2. Le réseau étant ordinaire, p a un (unique) jeton dans M_1, est une place de sortie de t, mais pas d'entrée de t. Ainsi, $\exists p \in t^\bullet \setminus {}^\bullet t \mid \Re(\{p\} \cup {}^\bullet t)$; d'où le résultat. □

L'algorithme ci-dessous, intitulé S_1, détecte des réseaux non-saufs en appliquant la contraposée de la proposition 5.6 et la proposition 6.20 ; S_1 détecte les réseaux saufs en appliquant la proposition 6.21 ; dans les cas restants, S_1 donne une réponse indéterminée.

```
1  function sauf() is
2        T⊥ ≔ T⊥ ∪ {t ∈ T | (|•t| ≤ 2) ∧ ((•t ⊗ •t) ≤ R⊤)}
3        for t ∈ T⊥ | (•t ⊂ t•) ∨ ((|•t| = 1) ∧ (((t• \ •t) ⊗ •t) ∩ R⊤ ≠ ∅)) loop
4              assert ℜ(•t) ∧ ((|•t| = 1) ⇒ (∃M ∈ ((t• \ •t) ⊗ •t), ℜ(M)))
5              return ⊥                  — d'après la prop. 5.6 (contr.) et la prop. 6.20
6        end loop

8        for t ∈ T⊥ ∪ T⊤ loop
9              for p ∈ t• \ •t | ({p} ⊗ •t) ⊈ R⊥ loop
10                   return ⊤            — l'algorithme ne peut prouver que ¬ℜ({p} ∪ •t)
11             end loop
12       end loop
```

```
14      assert T_⊤ ⊆ {t ∈ T | ¬ℜ(•t)}        — T_⊤ n'a pas de transition non-morte
15      assert R_⊥ ⊆ ⋕                        — R_⊥ n'a pas de places concurrentes
16      return ⊤                              — d'après la prop. 6.21
17  end function
```

Algorithme S_1 – Algorithme de ConcNUPN pour la propriété « sauf »

L'algorithme ci-après, dénommé S_2, permet d'évaluer la propriété « sauf ⇒ unit-sauf » par une lecture (directe) de la matrice des places concurrentes ; si elle est incomplète, la propriété peut rester indéterminée. Lorsque S_1 et S_2 sont tous deux évaluées à $\top$, le NUPN est prouvé unit-sauf.

```
1  function unit_sauf_si_sauf () is
2      if ∃{p1, p2} ∈ R_⊤ | (p1 ≠ p2) ∧ (¬ disjoint (unité (p1), unité (p2))) then
3          return ⊥                                  — d'après la def. 3.7
4      else if ∃{p1, p2} ∈ R_⌶ | (p1 ≠ p2) ∧ (¬ disjoint (unité (p1), unité (p2))) then
5          return ⌶   — valeur inconnue entre deux places d'unités non-disjointes
6      else
7          return ⊤                                  — d'après la def. 3.7
8      end if
9  end function
```

Algorithme S_2 – Algorithme de ConcNUPN pour la propriété « sauf ⇒ unit-sauf »

L'outil ConcNUPN calcule une matrice des places concurrentes avec les algorithmes C_3 et C_4 des sections 6.8 et 6.9, puis exécute les algorithmes S_1 et S_2 de cette section. Pour calculer la matrice des places concurrentes, d'autres algorithmes que C_3 et C_4 pourraient être utilisés, mais, comme eux, ils ne doivent pas recourir à l'hypothèse que le réseau en entrée soit sauf ou unit-sauf.

Sur les 16 200 NUPN de notre jeu de tests (de la section 3.8), les algorithmes S_1 et S_2 de ConcNUPN prouvent que 99,95 % d'entre eux sont saufs et que 99,86 % sont unit-saufs :

— Les huit modèles pour lesquels ConcNUPN donne un résultat indéterminé pour la propriété sauf peuvent être entièrement explorés par CÆSAR.BDD : le plus grand possède 540 places, 16 780 transitions, 137 780 arcs, ainsi que $4{,}07.10^6$ marquages accessibles. Puisque ces huit modèles sont saufs et triviaux, ils sont unit-saufs (d'après la proposition 3.3).

— Les neuf autres modèles où ConcNUPN ne peut déterminer la propriété unit-sauf, sont néanmoins prouvés saufs par ConcNUPN. Sur ces neufs modèles, huit proviennent de décompositions de réseaux en automates communicants [BGdL20], alors que le neuvième dérive d'une spécification écrite en LNT : de ce fait, ils ont tous une directive « `!unit_safe` » (voir la section 3.6), garantissant automatiquement la propriété unit-sauf. En outre, ces neufs modèles peuvent être entièrement explorés par CÆSAR.BDD, qui confirme qu'ils sont bien unit-saufs : le plus grand possède

480 places, 1 280 transitions, 8 988 arcs, 38 unités, ainsi que $1{,}37.10^{12}$ marquages accessibles.

L'édition 2022 du MCC comporte 777 réseaux non-colorés, ordinaires et ayant un marquage initial sauf. Sur ces 777 réseaux, 705 sont saufs et 72 ne le sont pas. Le prototype ConcNUPN a été en mesure de détecter 704 réseaux saufs et 4 réseaux non-saufs, ce qui représente des taux de succès respectifs de 99,86 % et 5,56 %. Parmi les 705 réseaux saufs, 695 ne sont pas triviaux (ils ont une décomposition NUPN) : ConcNUPN détermine que 688 d'entre-eux sont unit-saufs, ce qui représente un taux de succès de 98,99 % (tous les modèles non-triviaux de la compétition sont unit-saufs).

6.15 Validation des résultats

L'approche et les outils de validation croisée des vecteurs des places mortes et des transitions mortes décrite à la section 5.14 a été réutilisée : nous avons comparé et vérifié les matrices des places concurrentes produites par CÆSAR.BDD et ConcNUPN et de leurs versions successives, sur les réseaux de la section 3.8.

Ces contrôles mutuels ont dévoilé des erreurs dans CÆSAR.BDD, qui ont été corrigées ; depuis avril 2020, plus aucun nouveau bogue n'a été détecté dans CÆSAR.BDD. Des erreurs ont également été détectées dans des algorithmes pour des prototypes de ConcNUPN, mais ils concernaient des algorithmes expérimentaux qui n'ont, à ce jour, jamais été présentés ou utilisés dans des publications.

Ensuite, les matrices produites par Kong ont été comparées aux matrices produites par CÆSAR.BDD et ConcNUPN, toujours sur la collection de la section 3.8 : des problèmes ont été repérés et corrigés sur des versions de développement de Kong ; depuis décembre 2021, plus aucun nouveau bogue n'a été détecté dans Kong. Aucune erreur n'a été détectée dans CÆSAR.BDD et ConcNUPN.

Enfin, les résultats de ConcNUPN pour la décision des propriétés sauf et unit-sauf (voir la section 6.14) ont été comparés avec ceux de CÆSAR.BDD, sur les réseaux du MCC : leurs réponses concordent.

6.16 Bilan et perspectives

Ce chapitre a abordé l'étude du problème des places concurrentes qui, bien qu'utile en pratique, est mal couvert par les outils « classiques » de réseaux de Petri. Nous n'avons pas d'algorithme unifié permettant de résoudre ce problème, mais nous avons proposé une combinaison d'algorithmes (exploration statique ou dynamique de l'espace d'états, solution exacte ou approchée, coût polynomial ou exponentiel), implantés dans deux outils logiciels (écrits en langages C et Python). Notre approche est statistiquement performante sur un très grand nombre de modèles mais, ce problème étant PSPACE-complet, il subsistera toujours des modèles trop grands pour être analysés dans des délais raisonnables.

Les places concurrentes sont un sujet d'étude vaste. Plusieurs voies sont envisageables pour étendre les travaux de ce chapitre et leur portée.

Ce problème des places concurrentes possède d'autres applications potentielles, par exemple, il pourrait devenir partie intégrante du Model Checking Contest, comme le propose [Gar20], éventuellement après avoir été généralisé pour les réseaux non-saufs, ou les réseaux colorés.

Les matrices de places concurrentes pourraient également être employées pour évaluer certaines formules en logique temporelle : ceci pourrait permettre non seulement d'améliorer les performances des outils de vérification de modèles, mais également de proposer en amont des formules plus complexes à résoudre dans les compétitions d'outils (notamment pour celles du Model Checking Contest), en filtrant celles déductibles de la matrice.

Il serait toujours possible d'améliorer le calcul des places concurrentes, en perfectionnant les algorithmes afin de leur permettre de traiter des modèles encore plus grands, soit en calculant des solutions avec moins de valeurs inconnues, soit en fournissant des résultats équivalents, en moins de temps.

— Des formules en logique temporelle ou de validité booléenne pourraient être définies afin de cibler les paires de places restant inconnues lorsqu'il en reste, en faible nombre.

— Les algorithmes C_3 et C_4, des sections 6.8 et 6.9 réalisent leurs sous- et sur-approximations respectives au moyen d'une abstraction dite quadratique car chaque marquage M est enregistré sous la forme de paires de places $M \otimes M$. Ces algorithmes pourraient être généralisés en dimension $n \geq 3$, en mémorisant non pas des paires de places, mais les éléments de $\{M' \subseteq M \mid |M'| \leq n\}$, autrement dit, les sous-ensembles de M d'une taille au plus n. Par exemple, pour $n = 3$, l'abstraction devient *cubique*, le marquage M devenant $M \otimes M \otimes M$. Augmenter les dimensions de ces deux algorithmes leur permettraient de réduire davantage le nombre de paires inconnues dans la matrice. En outre, pour tout réseau N, il existe un certain rang $n \leq |P|$ tel que l'algorithme C_3 explore tous les marquages de $\{M \subseteq P \mid \Re(M)\}$, et ainsi découvre toutes les paires de places concurrentes, et soit donc exact.

  Le même raisonnement s'applique avec la généralisation de C_4, qui découvre toutes les paires de places non-concurrentes à partir d'un certain rang $n' \leq |P|$.

  De ce fait, à partir d'un certain rang $n'' \leq \min(n, n')$, la combinaison de ces algorithmes généralisés permettrait de déterminer une solution complète.

— Il existerait une alternative pour renforcer les résultats de C_4. Cette alternative a un coût mémoire de $O(|P|^3)$. L'espace d'états de cette généralisation alternative associerait à chaque paire de places, en remplacement des valeurs de $\{\bot, \top, \text{⊥⊤}\}$, le cardinal du plus grand marquage potentiellement accessible (autrement dit, accessible selon l'abstraction) contenant simultanément ces deux places.

  Dans la mesure où la structure en unités fournit également des contraintes sur la taille des marquages accessibles (dans un NUPN unit-sauf, un tel marquage ne peut, par exemple, comporter plus de places que le nombre d'unités feuilles), l'arbre des

unités pourrait, en complément, être utilisé pour raffiner encore davantage cette nouvelle généralisation.

— Les *invariants de places* (aussi appelés *semi-flots*) [MS81] [Kor92] [PCP99] [Sch03] [CMPW09] pourraient réduire le nombre de valeurs inconnues dans les matrices des places concurrentes. Cependant, une telle approche reste limitée : il existe des réseaux fortement connexes, vivants, présentant des paires de places non-concurrentes, mais pas d'invariant, même en étendant leur définition afin de leur autoriser de perdre leur jeton.

Voici, à titre d'exemple, un réseau vérifiant toutes les conditions susmentionnées : $P = \{p_1, \ldots, p_4\}$; $T = \{t_1, \ldots, t_4\}$; $M_0 = \{p_1, p_2\}$; $({}^\bullet t_1, t_1{}^\bullet) = (\{p_1, p_2\}, \{p_3\})$; $({}^\bullet t_2, t_2{}^\bullet) = (\{p_1, p_2\}, \{p_4\})$; $({}^\bullet t_3, t_3{}^\bullet) = (\{p_3\}, \{p_1, p_2\})$ et $({}^\bullet t_4, t_4{}^\bullet) = (\{p_4\}, \{p_1, p_2\})$.

— Nous analysons des modèles NUPN intrinsèquement parallèles, sur des machines ayant des nombres croissants de cœurs, avec des outils purement séquentiels. En particulier, l'ordre des algorithmes élaboré dans la section 6.10 pourrait être amélioré en ordonnançant en parallèle l'exécution des algorithmes, ce qui pourrait accélérer CÆSAR.BDD dans certaines situations. Les algorithmes C_2 à C_4 bénéficient certes des résultats de l'algorithme C_1 pour produire de meilleurs résultats, mais il s'agit de dépendances artificielles. Il est toujours possible de les appliquer en parallèle de C_1, puis une seconde fois après terminaison de C_1, si la matrice est restée incomplète. En outre, ces trois algorithmes pourraient être améliorés de sorte à ne pas recommencer ex nihilo les calculs lorsqu'ils sont appliqués plusieurs fois, mais uniquement sur les parties du réseau où de nouvelles paires pourraient être découvertes, permettant ainsi des applications successives sans surcoût notable.

Enfin, au delà de l'ordonnancement, les algorithmes de ce chapitre peuvent tous être implantés en parallèle, y compris C_1.

Quatrième partie

Structuration

Chapitre 7

Décomposition de réseaux de Petri en automates communicants

Le présent chapitre traite du *problème de la décomposition* automatique de tout réseau de Petri ordinaire et sauf en un *réseau d'automates communicants*, c'est-à-dire en un ensemble de composants séquentiels (machines à états finis) s'exécutant de manière asynchrone, pouvant se synchroniser entre eux et possédant le même comportement global que celui du réseau de Petri originel.

7.1 Motivations

Sur le plan théorique, les réseaux de Petri sont certes expressifs, mais ils sont faiblement structurés : la décomposition est un moyen de les (re-) structurer automatiquement, les rendant ainsi plus modulaires et espérons-le, plus faciles à comprendre et à raisonner.

Sur le plan pratique, les réseaux d'automates communicants contiennent des informations structurelles que les algorithmes de vérification formelle peuvent exploiter pour accroître leur efficacité, en utilisant, par exemple :

— Des codages logarithmiques des marquages accessibles (chaque automate contenant au plus un jeton) permettant une réduction potentiellement exponentielle du nombre de nœuds dans la représentation mémoire des marquages accessibles des diagrammes de décision binaires (voir la section 3.4).

— Une détection plus fine et facilitée des transitions indépendantes pour les méthodes de réduction de l'espace d'états reposant sur des ordres partiels.

— Des stratégies de type « diviser pour régner » pour la vérification compositionnelle (en évaluant, par exemple, des formules en logique temporelle sur chacun des automates, puis en reconstituant le résultat final sur le réseau entier), etc. En outre, une fois le réseau de Petri décomposé, il est possible de vérifier sur ce dernier de multiples propriétés, ce qui peut conduire à des gains de ressources conséquents. Ces gains

en terme de ressources peuvent aussi être utilisés pour traiter des modèles de plus grande taille.

La suite de ce chapitre est organisée comme suit. La section 7.2 énonce le problème de décomposition et discute de la forme des réseaux acceptés en entrée et produits en sortie. La section 7.3 situe ce chapitre parmi les travaux existants, et relie ce problème de décomposition au problème de coloriage de graphes. La section 7.4 expose le nombre de solutions au problème, expose un critère pour une solution « souhaitable », puis fournit des moyens pour la rechercher efficacement. Les sections 7.5 à 7.8 proposent quatre familles d'approches de décomposition, reposant sur des outils de graphes ou des formules logiques. La section 7.9 présente la chaîne d'outils implantée dans ce chapitre. La section 7.10 discute des résultats expérimentaux sur une collection complète de modèles. La section 7.11 aborde les démarches entreprises pour la validation des résultats. La section 7.12 compare notre solution de décomposition avec un outil existant, Hippo. La section 7.12 mentionne la contribution de notre chaîne d'outils pour structurer le jeu de tests de la compétition MCC (voir la section 3.8). La section 7.14 aborde la suite VLSAT, constituée de formules logiques produites par nos approches de décompositions, qui sont utilisées dans trois compétitions internationales. Finalement, la section 7.15 formule des observations conclusives et évoque les perspectives ouvertes.

7.2 Énoncé du problème

Dans ce chapitre, nous cherchons à apporter la modularité des réseaux d'automates communicants (voir la section 1.4) aux réseaux de Petri (voir la section 1.5).

Nous considérons le problème de décomposition comme une opération dans le domaine des réseaux de Petri, acceptant en entrée un réseau et produisant un réseau décomposé en sortie. Cette décomposition doit avoir un réseau de Petri sous-jacent isomorphe (voir la définition 4.3) au réseau d'entrée, autrement dit, préserver toutes les places, transitions, arcs et le marquage initial du réseau d'entrée. Elle doit ainsi permettre d'ajouter une structure en automates communicants au réseau d'entrée, mais en préservant toutes les propriétés structurelles et comportementales du réseau existant.

Nous reformulons le problème de décomposition en termes de NUPN. Puisque un automate est par essence un composant séquentiel et que la notion de hiérarchie est absente des réseaux d'automates communicants, le problème consiste à structurer automatiquement tout réseau de Petri ordinaire et sauf donné en un NUPN plat (voir la définition 3.6) et unit-sauf (voir la définition 3.7).

La figure 7.1 illustre à sa gauche un réseau de Petri ordinaire et sauf, en son centre un NUPN trivial équivalent (voir la section 3.2), et sa à droite, une décomposition possible en NUPN plat et unit-sauf.

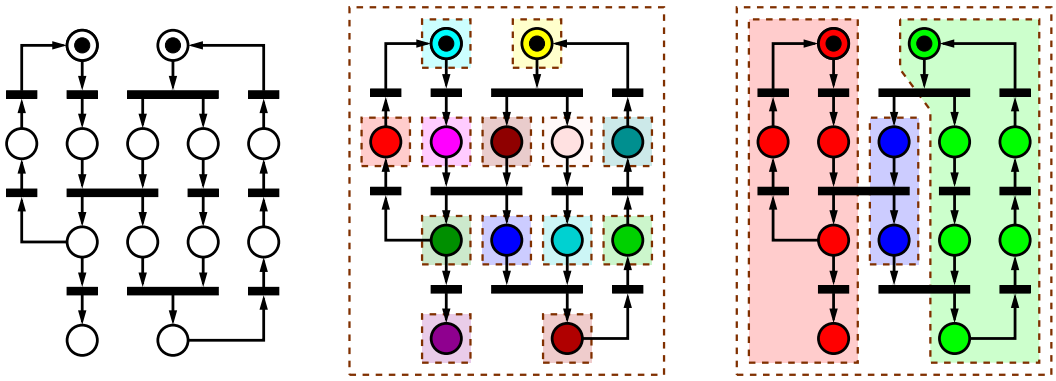

FIGURE 7.1 – Exemple de réseau de Petri (à gauche), de son NUPN trivial équivalent (au milieu) et une de ses décompositions possibles (à droite)

7.2.1 Graphe de concurrence

La suite de cette section introduit la notion de *graphe de concurrence*, afin de relier ce problème de décomposition au problème de coloriage de graphes. Il s'agit d'un graphe paramétré selon les places du réseau en entrée, qui sont ses sommets et selon $(R_\bot, R_\top, R_\amalg)$, la solution des places concurrentes calculée au chapitre précédent (voir la définition 6.4).

Définition 7.1 *Soit un réseau de Petri* (P, T, F, M_0) *et son triplet* $(R_\bot, R_\top, R_\amalg)$ *des places concurrentes. Le* graphe de concurrence *est le graphe* $(P, (R_\amalg \cup R_\top) \setminus \{\{p, p\} \mid p \in P\})$*, autrement dit, le graphe non-orienté, sans boucle, ayant pour sommets les places du réseau et dont les arêtes représentent les paires de places concurrentes ou inconnues de la matrice.*

Ce graphe de concurrence réalise (entre autres) une abstraction des paires de places à $\amalg$, qui sont traitées comme des paires de places concurrentes : autrement dit, deux places sont considérées concurrentes, à moins que le contraire ne soit prouvé.

Une approche alternative consisterait à générer une décomposition en supposant que ces valeurs inconnues soient des paires de places non-concurrentes, puis de vérifier que la décomposition soit bien unit-sauf. Si tel est le cas, la décomposition est acceptée, sinon, un contre-exemple permet de compléter $R_\top$, puis de produire une nouvelle décomposition, et ainsi de suite. Cette stratégie, de type « tentatives et échecs » présuppose néanmoins que nous puissions décider en temps raisonnable si un réseau est unit-sauf, alors qu'il s'agit d'un problème PSPACE-complet (voir la section 3.4).

7.2.2 Lien avec le coloriage de graphes

La proposition suivante montre qu'un coloriage du graphe de concurrence (voir la définition 2.11) est une solution au problème de décomposition de son réseau de Petri d'origine. Cette proposition montre également que la décomposition obtenue possède autant d'unités feuilles que le graphe coloré comporte de couleurs. Elle montre enfin que la décomposition obtenue est unit-sauve.

Proposition 7.1 *Soit N un réseau de Petri, (P, E) son graphe de concurrence associé et $\Pi_P : P \to \mathbb{N}$ un coloriage. L'ensemble $C = \{\{p \in P \mid \Pi_P(p) = k\} \mid k \in \{\Pi_P(p) \mid p \in P\}\}$ forme une partition des places P en autant de parties qu'il y a de couleurs, où chacune des parties est un ensemble de places deux à deux non-concurrentes.*

Preuve. Soit p_1, p_2 deux places distinctes. Nous avons $(\exists c \in C \mid \{p_1, p_2\} \in c) \Leftrightarrow (\Pi_P(p_1) = \Pi_P(p_2)) \Leftrightarrow (\{p_1, p_2\} \notin E) \Leftrightarrow (\{p_1, p_2\} \in R_\perp)$. D'où le résultat. □

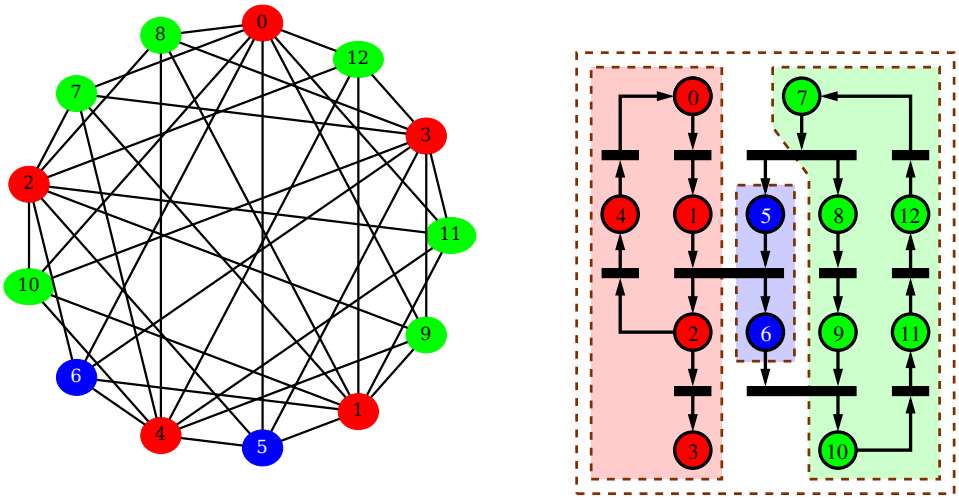

FIGURE 7.2 – Exemple de graphe de concurrence coloré (à gauche) et de sa décomposition associée (à droite)

Dans la figure 7.2, le coloriage du graphe de concurrence (à gauche) correspond à la décomposition $\{\{0 : 4\}, \{5 : 6\}, \{7 : 12\}\}$, qui est représentée par le NUPN (à droite).

La proposition suivante permet de montrer que, le problème de décomposition de ce réseau en un ensemble minimal d'automates communicants, en connaissant au préalable le graphe de concurrence d'un réseau de Petri, possède la même complexité théorique que le problème de coloriage de graphe, soit NP-complet.

Proposition 7.2 *Soit un graphe (V, E). Il est possible de résoudre le problème du coloriage de ce graphe en le réduisant vers un problème de décomposition d'un réseau de Petri en un réseau d'automates communicants, tel que $P \stackrel{\text{def}}{=} V$ et $\| \stackrel{\text{def}}{=} E$.*

7.3 Travaux voisins

La décomposition d'un réseau de Petri en processus séquentiels remonte au moins au début des années soixante-dix [Hac72] et a donné lieu à un corpus important de travaux scientifiques. Sur le plan théorique, nous pouvons mentionner :

— la décomposition de réseaux élémentaires en un ensemble de composants séquentiels communicants et concurrents [RE96, sect. 4.3–4.4] ;

— la décomposition de réseaux à choix libres bornés et vivants en *S-composants* ou *T-composants* [DE95, chap. 5] ;

— la distribution d'un réseau de Petri vers des localisations géographiques [vGGS08, vGGSU12, vGGSU13] [BD11] — ce dernier recourant à la théorie des régions [BD96] ;

— la synthèse de réseaux de Petri par l'élaboration de règles élémentaires de compositions et des taxonomies des classes de réseaux créés selon les règles employées [Esp90] [BdC92] ;

— la synthèse de réseaux de Petri depuis des expressions régulières [LRR03] et une catégorisation de la classe ainsi générée.

Sur le plan algorithmique, nous pouvons mentionner les approches :

— calculant une couverture d'un réseau de Petri par des machines à états fortement connexes [Hac72] ;

— reposant sur la structure d'un réseau de Petri pour le distribuer vers des localisations géographiques [Zai04] ;

— reposant sur la structure d'un automate pour le décomposer en un réseau d'automates (équivalent à des appels de fonctions) [DN89] [MO98, MO00, MO04] ;

— reposant sur la structure d'un réseau de Petri pour le décomposer hiérarchiquement en diagramme d'états de la norme UML [Esh09] ;

— reposant sur les invariants ou les semi-flux, calculés depuis les arcs du réseau de Petri [Sil85, sect. 7.4.3] [Kor92] [PCP99] [Sch03] [BCH10, BCH12] ou bien depuis ses marquages accessibles [Dyl13] ;

— reposant sur une analyse d'accessibilité des marquages d'un réseau de Petri, puis de l'expression du problème de décomposition de ce réseau sous la forme d'un problème de coloriage de graphe [WKAK14, WKA+18] ;

— reposant sur une analyse d'accessibilité des marquages d'un réseau de Petri, puis de l'expression du problème de décomposition de ce réseau sous la forme d'un problème de détermination des transversaux d'un hypergraphe [AW06, Wis11, WWA12, Wiś12, SAW13, TA13, GWW+14, WKA+18, WWJ19] (d'après ces publications, la représentation des hypergraphes construits serait plus compacte que celle des graphes) ;

— reposant sur une analyse d'accessibilité des états d'un automate, afin de le décomposer hiérarchiquement, suivant l'irréversibilité de ses actions [ENN06].

Concernant les implantations logicielles, nous pouvons mentionner l'outil Diane [MOW09], qui à ce jour, ne semble plus accessible, ainsi que l'outil Hippo, développé au sein de

l'université de Zielona Góra (Pologne) et disponible en ligne via un portail Web dédié [1]. À la section 7.12, nous comparons les approches de décomposition proposées dans ce chapitre avec celles fournies par Hippo.

La diversité des travaux existants concernant le problème de décomposition montre qu'il existe bien des façons d'aborder ce problème. La suite de cette section donne davantage de références concernant ces travaux liés, en discutant de la forme des réseaux acceptés en entrée et en énonçant précisément les contraintes auxquelles un formalisme de sortie approprié devrait se conformer.

7.3.1 Formalisme d'entrée

À la différence de, notamment, [PCP99], nous ne considérons pas les réseaux bornés qui ne sont pas saufs, pour les raisons évoquées dans les sections 7.1 et 7.2. Si nécessaire, un réseau borné peut être traité par notre approche, après avoir été converti en un réseau sauf, en dupliquant (tripliquant, et ainsi de suite) certaines de ses places.

Contrairement à d'autres approches, nous n'imposons pas de restriction supplémentaire sur les réseaux de Petri acceptés en entrée (à savoir ordinaires et saufs). Nous n'exigeons pas qu'ils soient :

- *à choix libres* et *bien formés* (soit vivants et saufs) [Hac72] ;
- *couvrables par des machines à états* [Sta90, def. 16.2.2 (180)] ;
- *propres* [Jan84] ;
- *décomposables en machines à états* [2] [Hac72, chap. 5] ;
- *allouables en machines à états* [2] [Hac74] ;
- *purs* (soit exempts de transition t telle que ${}^\bullet t \cap t^\bullet \neq \varnothing$) [WKA+18, Wiś18] ;
- *connexes* ou *fortement connexes* ;
- sans place morte et sans transition morte (voir le chapitre 5) [Jan84].

7.3.2 Formalisme de sortie

Chaque automate produit doit être séquentiel, contrairement à [Zai04], [MOW09] et [BD11] [ABC+19, sect 2.2, « modularité de Louvain »], où des places concurrentes peuvent être affectées au même endroit ou au même composant.

Les réseaux d'automates doivent être plats, contrairement à [KA06], [Esh09] et [MCvdA13], où les réseaux de Petri sont traduits en modèles hiérarchiques dans lesquels les processus peuvent avoir des sous-processus imbriqués. Le chapitre 8 suivant aborde de telles décompositions.

1. `hippo.iee.uz.zgora.pl`.
2. Voir [BdC92, sect. 7.1] pour une discussion relative à cette notion.

Les transitions de chaque automate doivent se comporter comme les transitions des réseaux de Petri, ce qui signifie que la synchronisation entre plusieurs automates doit être réalisée en respectant les règles de tir des transitions des réseaux de Petri (voir la définition 2.19) et non recourir à des mécanismes alternatifs, tels que les états de synchronisation dans les *automates synchronisés* [Pet90], le passage asynchrone de messages dans les *automates réactifs* [BCD02] ou encore les tampons « Premier Entré, Premier Sorti » dans les *automates communicants* [GKM07]. Il existe divers modèles répondant à de telles exigences, notamment les *réseaux d'automates* définis dans [Pau16] ou étudiés dans [BMS17].

Les ensembles d'états locaux de tous les automates d'un réseau doivent être deux à deux disjoints. Cette interdiction des états partagés apporte un fondement solide de la concurrence (les automates concurrents sont susceptibles d'être exécutés sur différentes unités de calcul et, de ce fait, ne devraient pas avoir d'états partagés) et est mathématiquement plus simple (les états locaux de tous les automates constituent une partition de l'espace d'états). Au contraire, les réseaux *fonctionnels* [Zai04] et *ouverts* [MOW09], reposent sur des places partagées pour les communications d'entrée/sortie entre plusieurs automates.

Toutes les propriétés structurelles et comportementales du réseau de Petri doivent être préservées par la décomposition ; ceci se traduit par le fait que le réseau de Petri sous-jacent de la décomposition produite doit être isomorphe (voir la définition 4.3) au réseau d'entrée. En particulier, nous préservons toutes les places et transitions du réseau en entrée (y compris les mortes), contrairement à [Sil85, sect. 7.4.3], [CSV86] et [Kor92], qui peuvent recourir à la duplication de places ou de transitions, ou à [KW15, WKA+18], qui produit d'abord une décomposition avec des états partagés, mais s'en délivre ensuite en introduisant des états auxiliaires « non-opérationnels ».

Un formalisme de sortie approprié doit être suffisamment général pour traiter, sans la moindre restriction injustifiée, tous les réseaux de Petri d'entrée (sous réserve qu'ils soient ordinaires et saufs). Par exemple, parmi les huit modèles susmentionnés de [BdC92, sect. 6–7], trois modèles (les réseaux *décomposables en machines à états*, *décomposables en machines allouables* et *à choix libres stricts*) exigent que chaque automate soit fortement connexe ; une telle restriction entrave évidemment la décomposition, car un réseau très simple composé de deux places et d'une transition entre ces places ne peut être exprimé à l'aide d'un seul automate fortement connexe.

Une autre restriction fréquente, mais discutable, est l'exigence selon laquelle les automates doivent être *conservatifs*, autrement dit, les automates comportent toujours un jeton dans n'importe quel état global. Dans le cadre des réseaux de Petri, cela veut dire que le marquage initial possède exactement un jeton par automate, et que toute transition non-morte comporte le même nombre de places d'entrée et de sortie. Les huit modèles de [BdC92, sect. 6–7] comportent cette restriction, qui, selon nous, est injustifiée : un réseau très simple comportant une seule place et une transition sortant de cette place ne peut être décomposé en un ensemble d'automates conservatifs. En pratique, la table 3.3 indique que seulement 19,2 % des modèles de la section 3.8 vérifient une telle condition. Il faut donc considérer un modèle plus flexible, dans lequel les automates peuvent être

démarrés et arrêtés dynamiquement, ce qui signifie que chaque automate ne comporte pas impérativement de jeton dans l'état initial, et qu'il est susceptible de perdre son jeton au fil de son exécution (par exemple, l'unité bleue de la décomposition à droite de la figure 7.1, qui peut gagner, puis perdre son jeton).

7.4 Solutions au problème de la décomposition

Cette section étudie l'existence et l'unicité des solutions de la décomposition d'un réseau de Petri en un NUPN plat et unit-sauf. Elle étudie des critères de qualité pour une solution souhaitable et expose un procédé pour la rechercher efficacement.

7.4.1 Existence d'une solution

Notre problème de décomposition, tel qu'énoncé ci-dessus, consiste à trouver un ensemble adéquat d'unités (à savoir, une partition de l'ensemble des places) pour convertir un réseau de Petri ordinaire et sauf en un NUPN « correspondant » plat et unit-sauf.

Contrairement à d'autres approches de décomposition (telles que [Hac72] [CSV86] [Kor92] [Esh09] [vGGSU13]) qui peuvent n'avoir aucune solution pour certaines classes de réseaux d'entrée, notre problème a systématiquement au moins une solution : le NUPN trivial correspondant au réseau de Petri d'entrée (voir la proposition 3.1) est plat (sa hauteur étant une, voir la définition 3.6) et unit-sauf (voir la proposition 3.3).

7.4.2 Multiplicité des solutions

Dans le cas général, il existe de multiples solutions pour le problème de décomposition. Les deux propositions suivantes montrent deux manières de produire, à partir d'une solution valide, une nouvelle solution valide.

Proposition 7.3 *Dans un NUPN plat et unit-sauf, la scission de toute unité feuille en deux unités distinctes produit un nouveau NUPN plat et unit-sauf.*

Proposition 7.4 *Dans un NUPN unit-sauf, une place p telle que $\forall p' \in P \setminus \{p\}, p \nparallel p'$ peut être située au sein de toute unité feuille, sans enfreindre la propriété unit-sauf.*

La relation de concurrence étant quasi-réflexive (voir la proposition 6.1), les places mortes vérifient la pré-condition de la proposition 7.4.

Voici, à titre d'exemple, d'autres places vérifiant la pré-condition de la proposition 7.4 : (i) une place seule à avoir un jeton dans le marquage initial et n'étant la place de sortie d'aucune transition non-morte ; (ii) toute place dans un réseau qui est une machine à états.

7.4.3 Critère idéal d'optimalité

Dans la mesure où le problème de décomposition admet généralement plusieurs solutions, il se pose le problème de déterminer une solution « optimale ». La solution triviale est

par exemple valide, mais dénuée d'intérêt « pratique », car elle n'apporte aucune information supplémentaire à son réseau de Petri sous-jacent. Divers critères d'optimalité sont envisageables :

— Pour exécuter efficacement un réseau de Petri sur une machine parallèle à n cœurs (ou n nœuds de calcul), une solution optimale aurait vraisemblablement n unités feuilles, de manière à exploiter pleinement les capacités de la machine.

— À l'inverse, sur une machine séquentielle (ou disposant d'un parallélisme disponible limité), il serait judicieux d'avoir des unités avec le plus grand nombre de places possible et de minimiser le nombre d'unités, ce qui permet notamment de limiter le nombre de changements de contexte.

— D'un point de vue théorique et, surtout, pour les besoins de la vérification formelle, un critère approprié est de minimiser le nombre de bits nécessaires pour représenter tout marquage accessible (voir la section 3.4).

Nous optons pour ce dernier critère. Pour nos besoins, nous allons recourir au codage le plus dense pour les réseaux plats figurant dans [Gar19], noté *(b)*, que nous perfectionnons par la détection d'unités ne recevant jamais de jeton ou ne perdant jamais leur jeton, selon la définition suivante.

Définition 7.2 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN :*

— *la* projection *d'un marquage* M *sur une unité* u *est l'opérateur binaire suivant :* $M \rhd u \stackrel{\text{def}}{=} M \cap \mathsf{places}\,(u)$ *;*

— *une unité* u *est dite* inactive *si elle n'a pas de jeton dans le marquage initial, et qu'aucune transition du réseau ne puisse déposer de jeton dans cette unité, en d'autres termes* $((M_0 \rhd u) = \varnothing) \wedge (\forall t \in T)\ ((t^\bullet \rhd u) \neq \varnothing \Rightarrow ({}^\bullet t \rhd u)) \neq \varnothing)$ *;*

— *une unité* u *est dite* permanente *si elle a un jeton dans le marquage initial, et que ce jeton ne peut être retiré de cette unité par aucune transition du réseau, en d'autres termes* $((M_0 \rhd u) \neq \varnothing) \wedge (\forall t \in T)\ (({}^\bullet t \rhd u) \neq \varnothing \Rightarrow ((t^\bullet \rhd u)) \neq \varnothing)$.

Notons que la détermination des unités inactives et permanentes est réalisable en temps linéaire, par une simple itération examinant les places d'entrée et les places de sortie de chaque transition.

Remarquons que, dans un NUPN plat, l'unité racine, si elle n'est pas feuille[3], est toujours inactive, puisqu'elle est vide. En utilisant le codage (b) de [Gar19, sect. 6], l'état de chaque unité feuille ayant n places locales peut être codé en utilisant $\lceil \log_2 (n+1) \rceil$ bits. Ce nombre de bits peut être encore réduit si l'unité est inactive ou permanente.

Définition 7.3 *Soit* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN plat. Le nombre de bits nécessaire à la représentation de tout marquage accessible de* N *est :* $\Sigma_{u \in U} \nu(u)$*, avec :* $\nu(u) \stackrel{\text{def}}{=} 0$ *si l'unité* u *est inactive ; sinon* $\nu(u) \stackrel{\text{def}}{=} \lceil \log_2 (|\mathsf{places}\,(u)|) \rceil$ *si* u *est permanente ; sinon* $\nu(u) \stackrel{\text{def}}{=} \lceil \log_2 (|\mathsf{places}\,(u)|) + 1 \rceil$.

3. Si l'unité racine est feuille, il s'agit de la seule unité du réseau, ainsi, elle contient toutes ses places.

Finalement, nous optons pour la définition 7.3 comme critère de comparaison : plus le nombre de bits requis par une décomposition est faible, meilleure cette dernière sera considérée.

7.4.4 Critère pragmatique d'optimalité

Nous venons de sélectionner le nombre de bits pour coder les marquages accessibles (voir la définition 7.3) comme étant le critère le plus judicieux pour mesurer finement la qualité d'une décomposition. Mais ce critère est délicat à exprimer pour les approches reposant sur la théorie des graphes ou sur la résolution de problèmes logiques, car il fait intervenir la fonction transcendante « $\log_2$ » et la fonction de passage du continu au discret « $x \mapsto \lceil x \rceil$ ». De ce fait, nos approches ne se focalisent pas directement sur la réduction du nombre de bits spécifié dans la définition 7.3, mais visent un autre objectif plus simple à mettre en œuvre : il s'agit de la réduction du nombre d'unités dans le NUPN de sortie.

Le nombre d'unités feuilles de toute solution appartient à un intervalle $\{Min : Max\}$, où :

— *Min* est le plus petit entier tel que pour tout marquage accessible M, $|M| \leq Min$; toute solution valide comporte au moins $|M_0|$ unités feuilles.

  Dans le cas des réseaux (sous-) conservatifs, la valeur de *Min* est toujours égale à $|M_0|$; dans le cas général, il est possible de calculer rapidement (une sous-approximation de) *Min*, sans pour autant devoir explorer entièrement l'ensemble des marquages accessibles, au moyen d'une exploration partielle des marquages accessibles, en partant de M_0, puis en tirant chaque transition au plus une fois, en privilégiant les transitions ayant plus de places de sortie que d'entrée.

  Pour admettre une solution valide, certains réseaux doivent comporter plus d'unités feuilles que *Min*. Par exemple, le réseau (P, T, F, M_0) tel que $P = \{p_0, p_1, p_2, p_3\}$, $T = \{t_1, t_2, t_3\}$, $M_0 = \{p_0\}$ et $F = (\{p_0\} \times T) \cup \{(t_i, p_j) \in T \times P \mid (i \neq j) \wedge (p_j \neq p_0)\}$ a au plus deux jetons, mais requiert au minimum trois unités feuilles.

— *Max* est le nombre de places du réseau (ceci résulte du dernier point de la définition 3.1). La limite supérieure *Max* est atteinte par, et seulement par, la solution triviale. Des approches destinées à réduire la valeur de la limite supérieure *Max* sont exposées ci-après, à la sous-section 7.4.6.

Généralement, avec un nombre fixé de places dans le réseau d'entrée et un nombre choisi $n \in \{Min : Max\}$ d'unités feuilles dans la décomposition de sortie, les places peuvent être réparties entre ces n unités de nombreuses manières différentes.

Notons que la réduction du nombre d'unités n ne coïncide pas toujours avec la réduction du nombre de bits. Considérons, par exemple, un réseau de dix places : une décomposition en deux unités permanentes de cinq places chacune nécessitera six bits, tandis qu'une autre décomposition en trois unités permanentes avec, respectivement, deux, deux et six places ne requerra que cinq bits. La même observation vaut pour les unités non-permanentes, par exemple $4 + 5$ contre $7 + 1 + 1$ places.

Mais une étude statistique pour toutes les distributions de places parmi n unités montre que

les valeurs moyennes et médianes du nombre de bits requis sont des fonctions croissantes de n. Ainsi, une recherche orientée vers la réduction du nombre d'unités devrait aussi, en règle générale, réduire le nombre de bits pour le codage des marquages accessibles.

7.4.5 Recherche dichotomique ou linéaire

Nous venons de voir, à la sous-section précédente, que toutes les solutions au problème de la décomposition ont leur nombre d'unités feuilles n comprises dans un intervalle $\{Min : Max\}$. Il semble donc tentant d'effectuer une recherche dichotomique dans cet intervalle en vue de trouver la solution avec le plus petit nombre d'unités (et, espérons-le, de bits).

Cependant, la dichotomie n'est optimale que si le coût de détermination d'une solution valide est constant sur toute la plage considérée. Nous avons observé que cette hypothèse ne peut être valable dans notre contexte. Par exemple, la détermination d'une solution avec beaucoup d'unités (n proche de Max) peut se révéler rapide, dans la mesure où il existe en général une multitude de solutions valides et qu'un outil n'a besoin que d'en trouver une seule. Inversement, la vérification d'une solution avec peu d'unités (n proche de Min) peut être plus lente si l'outil doit examiner et réfuter de nombreuses possibilités avant d'en trouver une valide ou de conclure à l'absence de solution — ce dernier cas étant fort coûteux en pratique, notamment avec les résolveurs de formules logiques.

Après avoir mis en œuvre la dichotomie, la recherche linéaire ascendante (partant de Min et incrémentant n jusqu'à ce qu'une solution soit trouvée) et la recherche linéaire descendante (partant de Max et décrémentant n, tant qu'une solution est trouvée), nous avons constaté que cette dernière est en règle générale plus efficiente. L'un des principaux avantages de cette dernière stratégie est qu'elle produit toujours une solution valide (mais éventuellement sous-optimale), même si la recherche est interrompue en raison d'une limite de temps spécifiée par l'utilisateur.

Chaque fois que possible, nous accélérons la convergence en demandant à l'outil s'il existerait une solution ayant au plus n unités feuilles (plutôt qu'une solution ayant exactement n unités feuilles). Si une une solution ayant m unités feuilles est trouvée, avec $m < n - 1$, l'itération suivante utilisera m plutôt que $n - 1$.

7.4.6 Réduction de la borne supérieure

Ayant opté pour une recherche linéaire descendante, nous présentons maintenant trois façons de réduire la valeur de la borne supérieure Max, valant au départ le nombre de places. Cela permet d'augmenter l'efficacité des approches de décomposition en restreignant l'espace de recherche, sans pour autant exclure les solutions pertinentes (la borne inférieure Min définie à la sous-section 7.4.4 demeurant intacte).

Premièrement, les places mortes, si elles sont présentes, peuvent être logées dans n'importe quelle unité feuille du NUPN de sortie, tout en préservant la propriété unit-sauf (voir la proposition 7.4).

Mais créer une unité supplémentaire pour contenir toutes les places mortes, ou pire encore, une unité supplémentaire par place morte, ne serait pas judicieux pour le coût en bits (voir la sous-section 7.4.3). En effet, de telles unités ne seraient pas toujours détectées comme inactives par la définition 7.2, qui est une simple approximation n'effectuant pas d'analyse d'accessibilité approfondie.

Au lieu de cela, notre approche consiste à retirer au préalable les places mortes du réseau d'entrée, avant de procéder à sa décomposition. De ce fait, nous menons une recherche linéaire descendante depuis $|P \setminus P_\top|$ au lieu depuis $|P|$. Finalement, nous réintégrons dans la décomposition finale les places mortes, en les distribuant dans les unités déjà existantes, en tirant profit des *emplacements libres*, autrement dit des valeurs binaires inutilisées dans le codage logarithmique de certaines unités : par exemple, une unité non-permanente ayant n places a $m = n + 1 - \lceil \log_2(n+1) \rceil$ emplacements libres, ce qui signifie qu'au plus m places mortes peuvent être ajoutées à cette unité sans augmenter son coût en bits. Lorsque plus aucun emplacement libre n'est disponible, l'unité comprenant le plus de places est choisie.

Deuxièmement, après avoir écarté les places mortes, nous pouvons toujours amorcer une telle recherche linéaire descendante à partir d'une valeur inférieure à $|P \setminus P_\top|$.

Proposition 7.5 *Soient les nombres* $Sum_p \stackrel{\text{def}}{=} |\{p' \in P \mid (p = p') \vee (\{p, p'\} \in R_\bot \cup R_\top)\}|$ *et* Sum*, le maximum de* Sum_p*, pour toutes les places* p*. Il existe une solution ayant* Sum *unités feuilles.*

Preuve. Soit Δ le degré maximal du graphe de concurrence. Sum est égal à $\Delta+1$. L'incrément « $+1$ » provient de la présence du terme « $p = p'$ » dans la définition de Sum_p, et du fait que P est non-nul (voir la définition 2.17). D'après le théorème de Brooks (étendu aux graphes éventuellement non-connexes), le nombre de couleurs nécessaires pour le graphe de concurrence est au plus égal à $\Delta + 1$, il existe donc au moins une solution ayant au plus Sum unités feuilles. □

Puisque $Min \leq Sum \leq |P \setminus P_\top|$, nous pouvons commencer la recherche à partir de Sum (plutôt que $|P \setminus P_\top|$), une solution optimale en terme de nombre d'unités feuilles ne pouvant pas en avoir plus de Sum. Lorsque $Sum < |P|$, ceci exclut délibérément des solutions sous-optimales, telles que le NUPN trivial (voir la définition 3.5).

Troisièmement, si $R_\bot = \varnothing$ et si le NUPN d'entrée N contient la directive « `!unit_safe` » (voir la section 3.6), alors, d'après la proposition 3.2, il existe au moins une solution avec $\mathsf{largeur}(N)$ unités feuilles. Ceci implique que la recherche linéaire descendante peut être lancée à partir de $\mathsf{min}(\mathsf{largeur}(N), Sum)$ plutôt que Sum, en excluant les solutions potentielles ayant plus d'unités feuilles que le NUPN d'entrée.

Néanmoins, si $R_\bot \neq \varnothing$ il est impossible de réutiliser $\mathsf{largeur}(N)$, car les paires de $R_\bot$ sont abstraites comme les paires de $R_\top$, induisant des contraintes supplémentaires, que la décomposition pré-existante est susceptible de ne pas satisfaire, y compris après application de la proposition 3.2.

7.5 Approches par coloriage de graphe

Comme indiqué par la proposition 7.1, le problème de décomposition de réseaux de Petri en automates communicants peut se ramener au problème de coloriage de graphes [WKAK14, WKA+18]. Notre première famille d'approches repose sur une telle réduction.

7.5.1 Principes

Partant d'une solution calculée au problème des places concurrentes (voir la section 6.11), un graphe de concurrence (voir la définition 7.1) est produit. Sur la base de ce graphe de concurrence, un outil de coloriage de graphe est invoqué (en gardant à l'esprit que le coloriage de graphe est un problème NP-complet). Enfin, une partition des sommets (autrement dit, des places) est déduite de la coloration obtenue (voir la proposition 7.1), ceci permettant de produire une décomposition valide.

Conséquence logique de la proposition 7.1, si $R_{\mathbb{I}} = \varnothing$, et si le graphe de concurrence est colorié en utilisant le plus petit nombre de couleurs possible (soit le nombre chromatique du graphe), la décomposition issue du coloriage minimise le nombre d'unités. Par exemple, le graphe de concurrence à gauche de la figure 7.2 possède un nombre chromatique de trois et est coloré avec trois couleurs. Ce coloriage est repris par le NUPN à droite de cette même figure, qui possède trois unités feuilles : il s'agit donc d'une solution optimale en termes d'unités (voir la sous-section 7.4.4).

7.5.2 Implantation et expérimentations

Pour implanter la décomposition de réseaux de Petri reposant sur le coloriage de graphes, notre choix s'est porté sur le logiciel Color6 [4] [ZLHX14], développé à l'Université de Picardie Jules Verne (France) et qui est l'un des plus récents outils de coloriage de graphes.

Color6 ne calcule pas nécessairement une solution optimale (autrement dit, avec le nombre chromatique), mais renvoie en réalité une solution ayant au plus un nombre de couleurs spécifié par l'utilisateur. En conséquence cet outil doit être invoqué de manière répétée, en utilisant la stratégie de recherche linéaire décroissante décrite dans la sous-section 7.4.5.

Nous avons développé des scripts écrits dans les langages shell et Awk, chargés de convertir le graphe de concurrence au format DIMACS [JT96], d'invoquer Color6 de manière itérative en utilisant la recherche linéaire descendante, d'analyser la sortie de Color6 et enfin d'attribuer des places aux unités une fois les itérations terminées. Les résultats obtenus sont présentés, comme pour toutes nos autres approches de décomposition, à la section 7.10.

Nous avons également expérimenté la dichotomie et la recherche linéaire ascendante (voir la sous-section 7.4.5), lesquelles se sont avérées, en moyenne, 8 % et 18 % plus lentes que la recherche linéaire descendante. En effet, Color6 nécessite, dans la plupart des cas,

4. home.mis.u-picardie.fr/~cli/EnglishPage.html

davantage de temps pour conclure à l'absence de solution (dans les cas où on lui donne un nombre de couleurs strictement inférieur au nombre chromatique) que pour faire émerger une solution lorsqu'il en existe au moins une.

7.6 Approches par calcul itéré de clique maximum

Notre deuxième famille d'approches (qui est originale, à notre connaissance) au problème de décomposition repose, comme pour la famille reposant sur le coloriage de graphes sur un graphe non-orienté et sans boucle.

7.6.1 Principes

La définition ci-dessous introduit la notion de *graphe séquentiel* [SY95] [WWJ19], qui est un graphe paramétré selon les places du réseau (qui sont ses sommets) et selon $(R_\perp, R_\top, R_\perp)$, la solution des places concurrentes calculée au chapitre précédent (voir la définition 6.4).

Définition 7.4 *Soit un réseau de Petri (P, T, F, M_0) et son triplet $(R_\perp, R_\top, R_\perp)$ des places concurrentes. Le* graphe séquentiel *est le graphe $(P, R_\perp \setminus \{\{p, p\} \mid p \in P\})$, autrement dit le graphe non-orienté, sans boucle, ayant pour sommets les places du réseau et dont les arêtes représentent les paires de places non-concurrentes de la matrice.*

Remarquons que deux sommets distincts du graphe de concurrence (voir la définition 7.1) sont adjacents *ssi* ces sommets ne sont pas adjacents dans le graphe séquentiel.

Dans le graphe séquentiel, une *clique* (voir la définition 2.9) est un ensemble de places ayant au plus un jeton dans tout marquage accessible, autrement dit, une clique représente une unité potentielle.

Remarquons que les places d'une clique du graphe séquentiel sont les places d'un *stable* (voir la définition 2.9) du graphe de concurrence. De ce fait, une *couverture (minimale) en cliques* du graphe séquentiel, à savoir, un ensemble (minimal) de cliques telle que leur union forme l'ensemble des sommets du graphe séquentiel, est un coloriage (optimal en nombre de couleurs) du graphe de concurrence.

Ici, nous allons utiliser un autre problème de graphe, consistant en la détermination de la, ou d'une des cliques du graphe ayant le plus de sommets, autrement dit, le problème de détermination d'une *clique maximum*[5] (voir la définition 2.9). Bien que ce problème soit également NP-complet, il est généralement plus rapide à résoudre en pratique que le coloriage de graphes.

Déterminer une clique maximum dans le graphe séquentiel permet de construire la plus grande unité possible dans le réseau. Afin d'obtenir une décomposition valide, il est possible d'itérer ce calcul : lorsqu'une clique maximum a été déterminée, le graphe séquentiel est simplifié en lui retirant les places de la clique. Ce processus continue tant que le graphe

5. À ne pas confondre avec une *clique maximale*, qui est une clique n'étant le sous-ensemble d'aucune autre clique. Une clique maximum est une clique maximale, mais la réciproque est fausse.

séquentiel contient au moins une place. L'ensemble des cliques calculées servent alors à construire les unités de la décomposition.

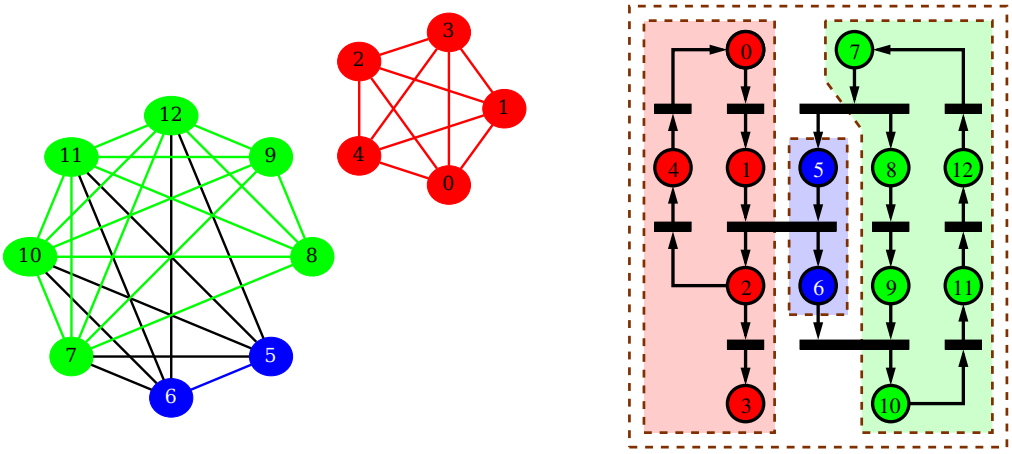

FIGURE 7.3 – Exemple de graphe séquentiel (à gauche) et de sa décomposition associée (à droite)

Par exemple, le graphe séquentiel à gauche de la figure 7.3 comporte deux cliques maximums, qui sont $\{7:12\}$ (en vert) et $\{5:7\} \cup \{10:12\}$. Supposons que pour produire la première unité, nous optons pour la clique verte. Une fois avoir retiré dans le graphe les sommets et arêtes associés à cette clique, il existe cette fois une unique clique maximum, correspondant à $\{0:4\}$ (en rouge), ce qui nous donne une deuxième unité. En retirant les sommets et arêtes de cette clique, nous obtenons $\{5:6\}$ (en bleu). Ceci nous donne alors une troisième unité. Le graphe n'a plus de sommet restant, la décomposition est alors terminée : elle correspond au NUPN en partie droite de la figure 7.3.

Notons que ces approches itérant la recherche d'une clique maximum ne sont pas nécessairement optimales concernant le nombre des unités.

7.6.2 Implantation et expérimentations

Les expérimentations ont été réalisées avec quatre outils de calcul de clique maximum : MAXCLIQUEDYN [6] [KJ07], BBMC [7] [SMRH13], MAXCLIQUEPARA [8] [DKR+13] — tous développés au sein de l'institut Jožef Stefan, à Ljubljana (Slovénie), et MOMC [9] [LJM17] développé au sein de l'université de Picardie Jules Verne (France).

Nous avons développé des scripts écrits dans les langages shell et Awk, qui génèrent automatiquement les graphes séquentiels au format DIMACS, qui invoquent ensuite les

6. insilab.org/maxclique
7. L'implantation originale n'étant pas disponible, nous avons utilisé la réimplantation suivante : commsys.ijs.si/~matjaz/maxclique/BBMC
8. commsys.ijs.si/~matjaz/maxclique/MaxCliquePara
9. home.mis.u-picardie.fr/~cli/EnglishPage.html

outils de graphes, puis extraient les cliques maximums de leurs sorties, et retirent les cliques maximums obtenues de leurs graphes.

La recherche linéaire ou dichotomique sur le nombre d'unités n'ayant pas lieu d'être dans cette famille d'approches, les remarques de la sous-section 7.4.5 ne la concernent pas.

En pratique, lorsque la clique calculée a moins de trois places, les unités restantes sont déduites directement d'une lecture du graphe, sans recours à l'outil de calcul de cliques.

L'implantation de cette famille d'approches est « à tout moment » : si la décomposition est interrompue, une solution partielle est obtenue en générant une nouvelle unité par place restante dans le graphe séquentiel.

7.7 Approches fondées sur la logique propositionnelle

Nous présentons maintenant, notre troisième famille d'approches, qui ramènent le problème de décomposition au problème de validité booléenne (problème de satisfaisabilité, *SAT*). En ayant recours à la logique propositionnelle, plutôt qu'à des problèmes de graphes (voir les sections 7.5 et 7.6), nous estimons proposer une approche novatrice de décomposition.

7.7.1 Principes

Les problèmes de validité booléenne seront ici exprimés à l'aide de formules sous *forme normale conjonctive* (en abrégé *formules FNC*). Il s'agit d'une restriction des formules en logique propositionnelle, où seules les conjonctions de clauses sont autorisées. Cette restriction provient du fait que la plupart des résolveurs modernes de problèmes SAT reposent sur des variantes de l'algorithme *CDCL* (acronyme anglais de *Conflict-Driven Clause Learning*), qui ne résout que des formules FNC. Bien qu'il existe des procédures polynomiales permettant de réduire tout problème SAT en problème SAT-FNC, il est plus efficace de générer directement les formules sous FNC lorsque le problème en entrée s'y prête bien. Nous verrons par la suite que ceci est le cas pour la décomposition de NUPN, d'où le choix des formules FNC seulement.

Les formules de ces approches sont paramétrées selon : (i) l'ensemble des places P ; (ii) la solution pour les places concurrentes $(R_\perp, R_\top, R_\amalg)$ de la définition 6.4 et (iii) un nombre maximal d'unités n.

Une formule est vraie *ssi* il existe une partition de P en au plus $n \in \{1 : |P|\}$ unités feuilles $\{u_1, \ldots, u_n\}$ telle que, pour toute unité u_i, pour toutes places p_1 et p_2 distinctes et localisées dans u_i, $\{p_1, p_2\} \in R_\perp$ (ceci implique[10] $p_1 \nparallel p_2$). Si tel est le cas, d'un modèle validant la formule, nous pouvons extraire une allocation de chaque place au sein d'une unique unité, avec au plus n unités, qui respectent la propriété unit-sauf ; ceci revient à dire que l'on peut extraire du modèle une décomposition valide.

10. La réciproque est vraie lorsque $R_\amalg = \varnothing$.

Ce problème peut également se voir sous la forme d'une instance du problème de coloriage de graphes, où au plus n couleurs peuvent être utilisées pour colorier le graphe de concurrence (voir la définition 7.1). Une formule est valide (autrement dit, satisfaisable) *ssi* la valeur de n est suffisamment grande (en fait, au moins égale au nombre chromatique du graphe) de sorte qu'il existe au moins une décomposition avec n unités.

Remarquons que, si une décomposition ayant n unités existe, il est possible de générer $n! - 1$ autres décompositions identiques, à la permutation des numéros d'unités près. Intuitivement, nous remarquons que nous pouvons pallier ce problème en imposant à la première place d'être localisée dans la première unité, puis en imposant à la deuxième place d'être localisée dans la première ou la deuxième unité, et ainsi de suite. Formellement, au lieu d'ajouter des contraintes dans les formules pour exprimer le fait que chaque place p est une place locale d'une certaine unité de $\{u \in U \mid 1 \leq \#_U(u) \leq n\}$, nous brisons la symétrie entre les unités en contraignant chaque place p à être une place locale d'une unité de $\{u \in U \mid 1 \leq \#_U(u) \leq \mathsf{min}(\#_P(p), n)\}$. Cette rupture de symétrie permet, d'une part d'éviter de déclarer des variables et des contraintes superflues (ce qui permet de réduire la taille des formules) et, d'autre part, de réduire (parfois sensiblement) le temps de traitement des formules invalides, en empêchant ainsi au résolveur d'effectuer de multiples retours en arrière lors de la découverte de contradictions, pour vainement tenter de découvrir un modèle.

Plus précisément, chaque formule est générée de la manière suivante. Pour chaque place p et chaque unité u telles que $1 \leq \#_U(u) \leq \min(n, \#_P(p))$, nous créeons une variable x_{pu}, qui soit vraie *ssi* la place p est une place locale de l'unité u. La condition $\min(n, \#_P(p))$ est utilisée à la place de n, car elle permet de casser la symétrie évoquée au paragraphe ci-dessus, et ainsi d'économiser $n \times (n+1)/2$ (car $n \leq |P|$) variables superflues.

Nous ajoutons ensuite les contraintes suivantes sur ces variables (x_{pu}) :

- nous exprimons le fait que deux places concurrentes ne peuvent pas être les places propres de la même unité : pour toute paire de places $\{p_1, p_2\} \in R_\perp \cup R_\top$ telle que $p_1 <_P p_2$ [11] et pour toute unité u telle que $\#_U(u) \leq \min(n, \#_P(p_2))$, $\neg x_{p_1 u} \vee \neg x_{p_2 u}$;
- nous exprimons le fait que chaque place soit la place propre d'au moins une unité : pour chaque place p, nous ajoutons la contrainte $\bigvee_{1 \leq \#_U(u) \leq \min(n, \#_P(p))} x_{pu}$.

Remarquons que la formule ainsi produite ne possède aucune contrainte requérant que chaque place ne soit la place propre d'au plus une unité. En effet, une telle contrainte nécessiterait de soit générer des formules FNC de taille $O(|U| \times |P|^2)$ au lieu de $O(|P|^2)$, soit de générer des formules sans la restriction FNC, pour les convertir en formules FNC, ce qui ne résoudrait en rien la problématique de départ. Ainsi, un modèle valide pour la formule produite n'est pas, à proprement parler, une solution au problème de décomposition, autrement dit un problème de partitionnement, où chaque place appartient à une unique unité, mais une réponse au problème plus général de couverture, où une place appartient à

11. La relation $\|$ étant symétrique, la condition $p_1 <_P p_2$ permet d'éviter d'exprimer des contraintes redondantes.

au moins une unité.

C'est pourquoi il est impératif d'effectuer un post-traitement sur le modèle satisfaisant la formule, afin de pouvoir en déduire une décomposition valide : il faut, pour chaque place, sélectionner une seule unité. Il serait possible de procéder à une telle opération, naïvement, comme suit : pour toute place p, $\mathsf{unité}(p) := \mathsf{choix}(\{u \in U \mid x_{pu} = \top\})$. À la sous-section suivante, plutôt que de recourir à une telle affectation avec la fonction choix, nous exploiterons cette liberté de sélection afin de réduire le nombre de bits requis pour représenter les marquages accessibles.

7.7.2 Implantation et expérimentations

Nos expérimentations ont été réalisées avec deux résolveurs : CaDiCaL 1.3.0 [Bie19] et MiniSAT 2.2.0 [ES03]. MiniSAT a été choisi pour sa popularité et pour son importance historique. CaDiCaL est quant à lui celui qui a résolu le plus de problèmes lors de la compétition SAT Race 2019 [12].

Nous optons, à l'instar des décompositions reposant sur le coloriage de graphes (voir la section 7.5), pour une recherche linéaire décroissante (voir la sous-section 7.4.5).

Techniquement, nous avons développé des scripts en langage Python, qui itérativement : (i) génèrent une formule DIMACS-CNF ayant au plus $n = Max$ unités ; (ii) invoquent un résolveur ; (iii) analysent sa sortie ; (iv) génèrent une allocation des places vers les unités ; (v) abaissent la borne supérieure Max (selon la décomposition obtenue), avant de recommencer.

Lorsque le modèle produit par le résolveur affecte une même place dans plusieurs unités distinctes, nous devons choisir une unique unité qui la contiendra dans la décomposition finale. Plutôt que de définir une affectation quelconque (par exemple, aléatoire avec la fonction choix), nous définissons une heuristique visant à réduire le nombre de variables utilisées pour coder les marquages accessibles. Notre stratégie prend son inspiration dans l'algorithme de remplissage de sacs par sélection systématique du pire sac par ordre décroissant (en anglais, l'algorithme *worst-fit-decreasing bin-packing*), où les unités sont vues comme des sacs :

— les places sont d'abord triées selon la fonction $p \mapsto |\{x_{pu} \mid x_{pu} = \bot\}|$ par ordre décroissant (autrement dit selon $p \mapsto |\{x_{pu} \mid x_{pu} = \top\}|$ suivant un ordre croissant, qui donne, pour chaque place, le nombre d'unités pouvant la contenir dans le modèle) ;

— ensuite, chaque place est mise dans l'unité ayant le plus grand nombre de d'emplacements libres (voir la sous-section 7.4.6), ou en cas d'égalité, le plus de places locales ;

— enfin, toute unité n'ayant pas (ou plus) de place propre est supprimée.

Notons que le coût de remplissage de chaque sac est la taille du codage de son unité

12. sat-race-2019.ciirc.cvut.cz

correspondante, qui est logarithmique par rapport à sa taille, et non linéaire, contrairement au problème classique de remplissage de sacs. De plus, la taille des sacs n'est pas bornée. D'où le choix de l'heuristique de sélection systématique du pire sac, qui donne en pratique de meilleurs résultats que, par exemple, l'heuristique de sélection systématique du premier sac possible.

Il convient de noter que chaque résolveur a une influence sur la forme des modèles calculés, notamment concernant le nombre de décompositions valides que l'on peut extraire. Par exemple, MiniSAT essaye, par défaut, d'affecter les variables encore indéterminées de la formule à $\bot$, cela signifie que les modèles calculés vont présenter un nombre minimal de variables à $\top$, ce qui revient à dire qu'elles ne disposent que d'une seule décomposition sous-jacente. Ceci n'est pas le cas de CaDiCaL, qui, en première tentative, affecte les variables encore indéterminées selon une heuristique.
Mais certains résolveurs permettent de modifier leur comportement : par exemple, MiniSAT dispose d'une option `-polarity-mode = {true, false, rnd}` permettant d'opter pour une première affectation à $\top$, à $\bot$, ou bien aléatoire. Pour ne pas complexifier davantage ce présent travail, abordant un problème déjà vaste, nous utiliserons systématiquement les options par défaut des outils.

7.8 Approches fondées sur la logique du premier ordre

Notre quatrième famille d'approches traduisent le problème de décomposition en formules logiques du premier ordre (autrement dit, des logiques *SMT*, acronyme de « SATisfaisabilité Modulo des Théories »), plus expressive que la logique propositionnelle de la section 7.7.

7.8.1 Principes

Ces formules sont paramétrées selon : (i) l'ensemble des places P ; (ii) la solution pour les places concurrentes $(R_\bot, R_\top, R_\top)$ de la définition 6.4 ; (iii) un nombre maximal d'unités n et (iv) le fragment de logique du premier ordre choisi parmi les six possibles, à savoir QF_BV, QF_UFBV, QF_DT, QF_UFDT, QF_IDL ou QF_UFIDL.

Quelque soit le fragment de logique utilisé, à toute place du réseau est associée au moins une unité et deux places distinctes dans la même unité sont non-concurrentes.

Les formules utilisant QF_DT, QF_UFDT, QF_IDL ou QF_UFIDL expriment un problème de partition des places, comme pour la famille à la section 7.5 : le typage des variables contraint chaque place d'un modèle à être affectée à au plus une unité. Les formules utilisant QF_BV ou QF_UFBV expriment au contraire un problème de couverture des places, à l'instar des formules de la section 7.7 : dans un même modèle, une place peut être affectée à plusieurs unités.

Nous prenons aussi en considération la présence de symétrie des numéros d'unités évoqués à la section 7.7, pour produire des formules plus courtes et plus faciles à résoudre.

Plus précisément, chaque formule est générée de la manière suivante, selon le fragment de logique choisi :

QF_BV. Ce fragment correspond à la logique *quantifier-free bit-vector*, qui apporte les vecteurs booléens de taille fixe, ainsi que les opérateurs logiques, relationnels et arithmétiques sur ces vecteurs. Notre codage pour QF_BV crée, pour chaque place p, un vecteur de bits b_p de taille n tel que $b_p[u]$ est vrai *ssi* la place p peut être une place locale de l'unité u (à la section 7.7, nous avions $|P| \times n$ variables propositionnelles, soit un nombre quadratique de variables ; nous passons ici à $|P|$ vecteurs de bits, soit un nombre linéaire de vecteurs, tous de taille n). Ensuite, pour empêcher que deux places concurrentes ne soient situées dans la même unité, la contrainte suivante est ajoutée : pour chaque paire de places p_1 et p_2 telle que $(\{p_1, p_2\} \notin R_\perp) \wedge (p_1 <_P p_2)$, $b_{p_1} \,\&\, b_{p_2} = \perp_{(n)}$. Finalement, pour chaque place p, nous ajoutons la contrainte suivante, qui exprime le fait que p soit la place propre d'au moins une unité, tout en cassant la symétrie des unités : $\bigvee_{1 \leq u \leq \mathsf{min}\,(p,n)} (b_p[1:u] \neq \perp_{(u)})$. La formule ainsi obtenue ne présente aucune contrainte requérant que chaque place soit la place propre d'au plus une unité : elle décrit un problème de couverture et non de partition.

QF_UFBV. Ce fragment correspond à la logique *quantifier-free uninterpreted-function bit-vector*. La théorie des fonctions non-interprétées rend possible la déclaration de symboles de fonctions qui ne sont donnés que par leur signatures (autrement dit par les types de leurs arguments et des résultats). Notre codage pour QF_UFBV est basé sur celui de QF_BV mais, au lieu des variables b_p, nous définissons une fonction non-interprétée u, de l'ensemble de vecteurs de bits de taille $\lceil \log_2(|P|) \rceil$ vers l'ensemble de vecteurs de bits de taille n ; chaque occurrence de b_p est remplacée par $u(\lambda(\#_P(p)))$ dans les contraintes, où λ est une fonction injective de $\{1, \ldots, |P|\}$ vers l'ensemble des vecteurs de bits de taille $\lceil \log_2(|P|) \rceil$. Comme pour QF_BV, un modèle satisfaisant une formule représente une couverture des places, et non une partition des places.

QF_DT. Ce fragment correspond à la logique *quantifier-free data-type*, qui permet la définition de types algébriques, tels que les types énumérés, les enregistrements, les listes, les arbres, etc. Notre codage pour QF_DT définit un type énuméré *Unité* contenant une valeur par unité. Il crée également, pour chaque place p, une variable x_p de type *Unité*. Ensuite, pour empêcher que deux places concurrentes ne soient situées dans la même unité, la contrainte suivante est ajoutée : pour chaque paire de places (p_1, p_2) telle que $(\{p_1, p_2\} \notin R_\perp) \wedge (p_1 < p_2)$, $x_{p_1} \neq x_{p_2}$. Finalement, la symétrie est cassée en ajoutant, pour chaque variable x_p dont le numéro de places $\#_P(p)$ est plus petit que n, la contrainte $\bigvee_{1 \leq u \leq p} (x_p = u)$.

QF_UFDT. Ce fragment correspond à la logique *quantifier-free uninterpreted-function data-type*. Notre codage pour QF_UFDT est basé sur celui de QF_DT mais, au lieu des variables x_p, nous définissons à la fois un type énuméré *Places*, contenant une valeur par place, et une fonction non-interprétée $u : \mathit{Place} \to \mathit{Unité}$, chaque occurrence de x_p étant remplacée par $u(p)$ dans les contraintes.

QF_IDL. Ce fragment correspond à la logique *quantifier-free integer-difference*, qui apporte les variables entières et les contraintes arithmétiques de la forme $(x - y)$ *op* c, où x est une variable entière, y est une variable entière ou une constante, *op* est un opérateur de comparaison, et c est une constante entière. Notre codage pour QF_IDL est basé sur celui de QF_DT, où les variables x_p ne sont plus de type *Unité* mais des entiers. Puisque les entiers ne sont pas bornés, chaque variable x_p ayant un numéro de place $\#_P(p)$ supérieur ou égal à n n'est sujette à aucune contrainte de casse de symétrie : elle doit être contrainte en ajoutant $\bigvee_{1 \leq u \leq n}(x_p = u)$.

QF_UFIDL. Ce fragment correspond à la logique *quantifier-free uninterpreted-function integer-difference*. Notre codage pour QF_UFIDL est basé sur celui de QF_IDL, avec les mêmes changements que pour passer de QF_BV à QF_UFBV.

7.8.2 Implantation et expérimentations

Notre chaîne d'outils repose sur le standard SMT-LIB 2.6 [13], qui fait référence dans la communauté SMT, aussi bien auprès des développeurs d'outils que les organisateurs des compétitions. Ce standard définit à la fois des fragments de logique du premier ordre avec leurs axiomes associés et un langage permettant d'interagir avec les résolveurs, aussi bien pour la génération de formules que pour l'interprétation du modèle calculé.

L'adoption de ce standard rend la chaîne d'outils indépendante des résolveurs sous-jacents utilisés. Notre chaîne d'outils peut générer des formules en utilisant les six fragments standards de logique du premier ordre, décrits ci-dessus.

Il est possible d'employer au sein de notre chaîne d'outils n'importe quel résolveur de formules logiques du premier ordre acceptant (au moins) l'un des fragments de logique SMT-LIB que nous utilisons. Nous avons expérimenté quatre résolveurs SMT : Z3 [dMB08] et CVC4 [BCD+11], lesquels sont suffisamment généraux pour traiter tous les fragments logiques susmentionnés, ainsi que Boolector [NPB14] et Yices [Dut14], qui étaient parmi les plus rapides de la compétition SMT-COMP 2019 [14], pour les fragments de logiques qu'ils étaient capables de traiter.

Comme pour les familles d'approches des sections 7.5 et 7.7, nous adoptons une recherche linéaire décroissante (voir la sous-section 7.4.5).

Techniquement, nous avons développé des scripts en langage Python, qui itérativement : (i) génèrent une formule SMT-LIB 2 ayant au plus $n = Max$ unités ; (ii) invoquent un résolveur ; (iii) analysent sa sortie ; (iv) génèrent une allocation des places vers les unités ; (v) abaissent la borne supérieure *Max* (selon la décomposition obtenue), avant de recommencer.

Nous avons testé 14 combinaisons (fragment de logique, solveur), auxquelles nous ajoutons une quinzième combinaison en traitant le fragment QF_IDL avec les capacités d'optimisation

13. smtlib.cs.uiowa.edu
14. smt-comp.github.io/2019

linéaire de Z3 : ceci est fait en enrichissant la formule QF_IDL avec une directive (spécifique à Z3) « min » demandant à Z3 de calculer une solution comportant le plus petit nombre d'unités ; ainsi, aucune itération n'est requise pour cette approche, que nous appelons « z3opt » [15]. Pour minimiser le nombre d'unités, une nouvelle variable entière, *Bound* est créée, et pour chaque place p, une contrainte $x_p \leq Bound$ est ajoutée. Enfin, une directive d'optimisation min (*Bound*) est donnée.

Lorsque les fragments QF_BV et QF_UFBV sont utilisés, si le modèle calculé n'est pas une partition des places, il est transformé en une partition à l'aide de l'algorithme de remplissage de sacs décrit pour la troisième famille d'approches (voir la section 7.7).

Notons que pour QF_IDL et QF_UFIDL, nous pourrions remplacer certaines clauses disjonctives par des contraintes de différence (par exemple, $x_3 = 1 \vee x_3 = 2 \vee x_3 = 3$ serait remplacée par $0 \leq x_3 \leq 3$) ; mais cette variante s'est avérée 14 fois plus lente lors de nos expérimentations préliminaires avec le résolveur Z3, et par conséquent, abandonnée par la suite. Ce résultat n'est pas surprenant, puisque les résolveurs SMT semblent traiter efficacement les clauses disjonctives en les transmettant à leurs résolveurs SAT sous-jacents.

7.9 Implantations logicielles et formats de fichiers

Notre chaîne d'outils accepte en entrée un fichier « `.nupn` », contenant un réseau de Petri représenté dans le format NUPN, ou un fichier « `.pnml` » dans le format normalisé PNML (voir la section 3.6) ; elle produit en sortie un fichier « `.nupn` » contenant le résultat de la décomposition.

Pour les besoins de nos expériences, les réseaux d'entrée qui ne sont pas triviaux sont traités comme s'ils l'étaient, en ignorant délibérément leurs structures NUPN. La seule exception à cette règle est faite lors de l'application des stratégies de réduction de la borne supérieure pour la recherche linéaire descendante (voir la sous-section 7.4.6).

Le réseau décomposé est (dans une grande mesure) identique au réseau « `.nupn` » d'entrée, mais comporte de nouvelles informations concernant les unités. Puisque le format « `.nupn` » exige que les places locales de toute unité forment un intervalle, la décomposition requiert une permutation des numéros des places, la correspondance entre les numéros de place d'entrée et de sortie est conservée dans la section « `labels` » du fichier de sortie. Son réseau de Petri sous-jacent est isomorphe au réseau d'entrée (voir la définition 4.3) : le marquage initial, les transitions et les arcs sont identiques dans les fichiers d'entrée et de sortie, après renumérotation des places.

Notre chaîne d'outils est modulaire et rassemble des composants logiciels d'origines diverses. Certains sont dédiés aux réseaux de Petri (voir la section 3.7). Par exemple, la décomposition triviale d'un modèle NUPN (voir la sous-section 7.4.1) s'obtient en invoquant Nupn_Info avec l'option « `-trivial-units` ». Les tailles des codages (voir la sous-section 7.4.3)

15. Les formules pour z3opt conservent les contraintes cassant la symétrie : les formules restent paramétrisées par la borne supérieure n. Ces contraintes imposent aussi à toute solution de vérifier $Bound \leq n$.

et les bornes *Min* et *Max* (voir la sous-section 7.4.4) peuvent être calculées par l'outil CÆSAR.BDD, pour tout modèle NUPN, respectivement avec les options « `-encodings` », « `-min-concurrency` » et « `-max-concurrency` ».

Mais beaucoup de composants logiciels de notre chaîne d'outils ne sont pas spécifiques aux réseaux de Petri (voir les sections 7.5 à 7.8).

Enfin, notre chaîne d'outils est composée d'une collection de scripts, écrits dans les langages Awk, shell et Python ; ces scripts permettent l'exécution des décompositions, en implantant (notamment) les étapes des sous-sections 7.4.5 à 7.4.6 et en orchestrant l'invocation des outils susmentionnés.

7.10 Résultats expérimentaux

Les expérimentations présentées ci-après proviennent de [BGdL20] et ont été réalisées en janvier 2020. Pour cela, nous avons utilisé les modèles du jeu de tests présenté à la section 3.8. Il contenait à l'époque 12 728 modèles, mais les 3 472 modèles ajoutés depuis ne changent guère ses caractéristiques. Les chiffres de [BGdL20, Table 1] (respectivement [BGdL20, Table 2]) sont proches de ceux de la table 3.3 (respectivement de la table 3.4).

Nos expérimentations ont été réalisées sur des ordinateurs individuels équipés de processeurs Intel Xeon W3550 (3,07 GHz), de 12 Go de RAM et exécutant la distribution Linux Debian 10.

Ces expérimentations sont d'ampleur considérable : le produit des 12 728 NUPN de notre collection en entrée par les 22 approches de décomposition représentent 280 016 décompositions à procéder, nécessitant au total 32 jours de temps processeur, sans compter les temps de vérification pour les 265 121 décompositions ayant réussi (l'écart s'expliquant soit par les exécutions dépassant le délai imparti, celles faisant l'objet de plantages des outils de graphes ou des résolveurs de formules logiques, soit par les rares modèles de trop grande taille, n'ayant pas de matrice des places concurrentes (lorsque ces expérimentations ont été réalisées, l'outil CÆSAR.BDD n'était pas en mesure de produire les matrices de douze réseaux).

La table 7.1 synthétise les résultats expérimentaux de nos approches de décomposition, lesquelles sont énumérées dans la colonne 1 de ce tableau. Toutes les approches reposent sur les matrices de places concurrentes, préalablement calculées pour chaque modèle NUPN d'entrée de notre collection. Les différentes approches ont été appliquées sur tous les modèles, avec un délai imparti de deux minutes par modèle. En cas d'échec d'une approche (en raison, par exemple, d'un modèle de taille trop importante, de l'absence de sa matrice des places concurrentes, de contraintes trop complexes à traiter par un résolveur, et ainsi de suite), la solution triviale est retenue en guise de résultat. Lorsqu'un dépassement de délai se produit pour une approche recourant à une recherche linéaire descendante, la dernière solution intermédiaire calculée est retenue.

— La colonne 2 (*% de succès*) donne le pourcentage de modèles décomposés par

approche de décomposition	% de succès	nombre d'échecs	hors délais	temps total (HH:MM:SS)	ratio des bits total	ratio des bits moyen	ratio des unités total	ratio des unités moyen
col-color6	97,8 %	13	338	14:20:01	76,4 %	60,0 %	71,9 %	28,3 %
clq-bbmc	99,1 %	15	170	17:48:34	66,2 %	57,7 %	61,7 %	27,8 %
clq-maxcl	99,2 %	15	157	17:11:48	68,1 %	57,8 %	63,5 %	28,1 %
clq-mcqd	99,1 %	15	178	18:26:56	69,9 %	57,8 %	65,5 %	27,9 %
clq-momc	98,7 %	16	213	19:00:01	73,7 %	58,3 %	69,0 %	28,5 %
sat-cadical	96,2 %	12	697	26:57:30	73,1 %	59,9 %	69,4 %	29,5 %
sat-minisat	95,5 %	12	795	31:59:00	77,9 %	61,1 %	74,3 %	30,1 %
smt-bv-boolector	95,1 %	12	728	29:34:34	77,4 %	60,5 %	74,0 %	30,4 %
smt-bv-cvc4	94,2 %	12	855	35:04:50	78,7 %	61,4 %	75,5 %	31,3 %
smt-bv-yices	95,4 %	12	685	26:05:16	73,7 %	61,3 %	70,0 %	30,1 %
smt-bv-z3	94,0 %	12	881	37:19:22	78,3 %	61,7 %	75,1 %	31,4 %
smt-dt-cvc4	92,9 %	12	1104	48:50:21	81,9 %	62,7 %	79,0 %	32,4 %
smt-dt-z3	92,7 %	12	1100	41:12:21	83,3 %	62,4 %	80,6 %	32,7 %
smt-idl-cvc4	92,5 %	12	1220	47:59:45	84,6 %	63,6 %	81,7 %	33,5 %
smt-idl-yices	93,2 %	12	1082	40:52:12	83,7 %	63,0 %	80,7 %	32,2 %
smt-idl-z3	95,0 %	12	848	33:30:06	80,0 %	61,3 %	76,5 %	30,6 %
smt-idl-z3opt	87,3 %	12	1921	73:01:03	87,2 %	65,9 %	85,1 %	37,2 %
smt-ufdt-cvc4	92,4 %	12	1167	44:39:34	84,2 %	63,5 %	81,5 %	33,1 %
smt-ufdt-z3	92,9 %	12	1078	40:36:39	82,9 %	62,4 %	80,2 %	32,6 %
smt-ufidl-cvc4	90,8 %	12	1468	57:03:43	85,7 %	64,7 %	83,1 %	34,6 %
smt-ufidl-yices	94,9 %	12	823	30:41:15	77,5 %	60,9 %	74,1 %	30,6 %
smt-ufidl-z3	94,0 %	12	952	36:46:32	80,7 %	61,8 %	77,5 %	31,4 %

TABLE 7.1 – Résultats comparatifs de nos 22 approches de décomposition

l'approche correspondante, qu'un dépassement de délai se soit produit ou non.

— La colonne 3 (*nombre d'échecs*) donne le nombre de modèles pour lesquels la décomposition a échoué, sans survenue d'un dépassement de délai.

— La colonne 4 (*hors délais*) donne le nombre de modèles pour lesquels un dépassement de délai s'est produit.

— La colonne 5 (*temps total*) donne le temps passé par l'approche sur tous les modèles, en excluant le temps passé au pré-calcul de la matrice des places concurrentes et à la validation des résultats.

— La colonne 6 (*ratio des bits*) mesure la qualité de la décomposition en donnant le quotient du nombre de bits du modèle décomposé divisé par le nombre de bits du modèle d'entrée. Si un modèle d'entrée est non-trivial, il est remplacé par le NUPN trivial équivalent ; la sous-colonne « *total* » donne le quotient entre les sommes de bits pour tous les modèles, tandis que la sous-colonne « *moyen* » donne la moyenne des quotients obtenus pour chaque modèle.

— La colonne 7 (*ratio des unités*) fait la même chose que la colonne 6, en considérant le nombre d'unités (feuilles) plutôt que le nombre de bits.

De nombreuses observations peuvent être dégagées de la table 7.1. Tout d'abord, les approches reposant sur la recherche itérée de clique maximum sont les plus performantes : elles figurent parmi les approches les plus rapides, avec le moins de dépassement de délais, et produisent les décompositions les plus compactes — certainement parce qu'elles parviennent à traiter de grands modèles pour lesquels les gains potentiels en bits et en unités sont les plus conséquents. Hormis les approches reposant sur les cliques, les meilleures approches sont, lorsque l'on considère les rangs cumulés pour les deux colonnes des bits, « sat-cadical », « col-color6 », « smt-bv-boolector », « smt-bv-yices » et « smt-ufidl-yices ».

Nous constatons que la variante « smt-idl-z3opt » est l'approche la moins efficace de notre chaîne de décomposition, quelque soit le critère considéré.

Les expériences montrent clairement que les approches les plus rapides (en termes de nombre de délais dépassés) donnent les meilleurs résultats, à la fois en termes de nombre de bits et de nombre d'unités. De plus, les expériences montrent que les nombres d'unités sont fortement corrélés aux nombres de bits, ce qui valide notre approche. Cependant, il ne suffit pas d'atteindre la meilleure décomposition en termes d'unités, le critère idéal d'optimalité étant le nombre de bits (voir la sous-section 7.4.3). Cela explique pourquoi, par exemple, « col-color6 » a presque deux fois moins de dépassements de délais que « sat-cadical », ce qui donne de meilleurs ratios d'unités, mais a des résultats légèrement moins bons en terme de ratios de bits.

Considérer le temps cumulé d'exécution sans considérer le nombre de succès, ou bien prendre en compte les ratios moyens d'unités ou de bits sans prendre en compte les ratios totaux d'unités ou de bits, (et ce faisant, accorder trop d'attention aux petits modèles) est un piège à éviter. Par exemple, l'approche par coloriage de graphes « col-color6 » a certes le mérite d'être (de loin) la plus rapide approche pour les plus petits réseaux, mais elle est moins performante que les approches utilisant les cliques pour les réseaux de grande taille, plus difficiles à décomposer.

Nous constatons que les approches reposant sur la recherche itérée de clique maximum donnent des résultats homogènes, que ce soit en nombre d'échecs et délais échus, temps, ratios des bits et des unités. Au contraire, les approches recourant aux résolveurs de formules logiques, présentent de très fortes disparités dans les résultats obtenus, ces approches étant particulièrement sensibles au résolveur choisi. Les approches employant des formules propositionnelles ou des formules du premier ordre avec des fragments de logiques de plus bas niveau, à l'instar de « smt-bv », sont les plus efficaces, alors que les approches mettant en œuvre des formules du premier ordre avec des fragments de logiques de plus haut niveau, telles que « smt-dt » et « smt-ufdt », sont les moins efficaces. Il faut néanmoins nuancer cette explication : dans les compétitions de résolveurs [16], nous pouvons constater une corrélation inverse entre la popularité d'un fragment de logique et de son niveau d'abstraction (et

16. Voir smt-comp.github.io.

conséquemment, de sa simplicité). Il est donc vraisemblable que les résolveurs soient davantage optimisés pour les fragments les plus populaires. En outre, nous constatons que le meilleur résolveur dans un fragment de logique donné n'est pas nécessairement le meilleur dans un autre fragment de logique : par exemple, « smt-idl-z3 » est plus performant que « smt-idl-yices », mais « smt-ufidl-yices » s'avère meilleur que « smt-udidl-z3 ».

Expérimentalement, nous avons mesuré que parmi les décompositions que l'on peut extraire d'un même modèle calculé par « sat-cadical », le nombre de bits requis pour coder les marquages accessibles varie d'en moyenne 10 % entre la meilleure et la plus mauvaise solution. Ceci justifie l'intérêt de l'algorithme inspiré du remplissage de sacs décrit dans la section 7.7 (aussi utilisé par les approches « smt-bv-* », qui ont également de bons résultats).

Il est intéressant de noter que nos résultats réfutent l'affirmation selon laquelle la taille des diagrammes de décision serait moindre lorsque les unités ont un nombre de places *équilibré* [SY95] [BD98]. Au contraire, nous observons que les codages les plus compacts sont obtenus en utilisant les approches de recherche itérée de clique maximum, lesquelles produisent des décompositions fondamentalement déséquilibrées dans la mesure où elles calculent de très grandes cliques lors des premières itérations et de très petites cliques lors des dernières. Notre codage utilise pour chaque unité un nombre logarithmique de variables en fonction de son nombre de places. Or, le logarithme étant une fonction concave, plus une décomposition est « déséquilibrée », plus elle permet de réduire le nombre de variables. Il semblerait que la réduction de la taille des diagrammes évoquée par les deux travaux susmentionnés soit plus liée à l'emploi du « degré d'orthogonalité » des unités pour les trier plutôt qu'à une répartition *équilibrée* des places au sein des unités. Ce tri des unités s'apparente à un ordonnancement initial des variables et est à mettre en perspective avec [ABD17].

7.11 Validation des résultats

Nous avons systématiquement contrôlé chaque fichier NUPN produit par notre chaîne d'outils de décomposition, afin de nous assurer que les propriétés suivantes sont vérifiées :

— Le réseau de sortie est syntaxiquement et sémantiquement correct (en exécutant CÆSAR.BDD avec l'option « `-check` »).

— Il existe une bijection des numéros de places du réseau d'origine vers celles du réseau restructuré de sorte qu'après permutation des places, le marquage initial et les arcs des deux réseaux soient identiques.

— Le réseau de sortie est plausiblement unit-sauf, autrement dit, toutes les paires distinctes de places contenues dans $R_\top$ appartiennent à des unités distinctes ; si de plus, toutes les paires de places distinctes contenues dans $R_\bot$ appartiennent à des unités distinctes et que le réseau d'origine est unit-sauf, alors le réseau de sortie est lui aussi garanti unit-sauf.

— Nous explorons également (au moins une partie des) marquages accessibles du réseau de sortie, afin de nous assurer qu'ils respectent la propriété unit-sauf (au moyen de CÆSAR.BDD, avec l'option « `-check` »), en imposant un temps imparti d'une minute pour cette exploration. Si cette dernière est complète, le réseau est prouvé unit-sauf et nous vérifions également qu'il possède le même nombre de marquages accessibles que le réseau d'origine.

— Le réseau de sortie a les mêmes propriétés numériques, structurelles et comportementales que le réseau NUPN d'origine, y compris celles figurant dans la table 3.3 (hormis « non trivial et unit-sauf ») et la table 3.4 (à l'exception des trois dernières lignes consacrées aux unités). Ceci est fait en comparant les sorties de CÆSAR.BDD, pour l'option « `-mcc` », toujours avec un temps imparti d'une minute.

Pour réaliser cette procédure de validation rigoureuse, en plus des ordinateurs de la section 7.10, nous avons fait un usage intensif de serveurs équipés de processeurs Intel Xeon E5-1650 v4 (3,60 GHz), de 128 Go de RAM et exécutant OpenIndiana, ainsi que des serveurs de la grille de calcul Grid'5000.

Grâce à ces validations, nous avons détecté en 2019 et en 2020 des bogues dans deux outils de détermination de clique maximum (MoMC et BBMC), dans l'outil de coloriage de graphes (Color6) et dans deux résolveurs de formules logiques (CVC4 et Z3), ainsi que divers problèmes de divergence de formats de sortie entre certains outils. Ces problèmes ont été signalés à leurs auteurs et corrigés dans la mesure du possible, ou à défaut, contournés.

7.12 Comparaison avec l'outil Hippo

En septembre 2019, nous avons comparé nos approches de décomposition avec les trois approches (« graphes de comparabilité », « invariants heuristiques » et « coloration d'hypergraphes ») de l'outil Hippo, développé au sein de l'université de Zielona Góra (Pologne).

Nous avons effectué cette comparaison sur la totalité des 223 réseaux de Petri saufs disponibles sur le portail web d'Hippo[17] (après suppression des modèles isomorphes, voir le chapitre 4). Notre chaîne d'outils a révélé que plusieurs de ces réseaux étaient incorrects et que les deux dernières approches de décomposition pouvaient produire des résultats invalides ; nous avons signalé ces problèmes (il y avait trois fichiers XML syntaxiquement incorrects, neuf paires de modèles isomorphes et un modèle marqué par erreur comme non-sauf), résolus postérieurement par les développeurs d'Hippo.

En poursuivant notre évaluation, nous avons mesuré que les trois approches d'Hippo ont pris, respectivement, 65 secondes, 25 minutes et 1 heure 53 minutes pour traiter la collection de modèles de référence. Le portail Hippo attribue un temps limite d'environ 22 minutes, dans la mesure où, pour cinq des modèles, [WWJ19] rapporte que la décomposition à l'aide d'hypergraphes prend plus d'une heure. Dans les faits, les trois approches ont pu traiter, respectivement, 70,5 %, 92,5 %, et 91,1 % de la collection.

17. hippo.iee.uz.zgora.pl.

En comparaison, notre chaîne d'outils n'a mis que 16 secondes pour traiter la totalité de la collection, en utilisant l'outil Color6 sur un ordinateur portable de 2011, équipé d'un processeur Intel Core i5-3210M (2,50 GHz), de 8 Go de mémoire, d'un disque dur mécanique et fonctionnant sous Debian Linux 10. Les modèles décomposés produits par notre chaîne d'outils sont, en outre, plus compacts, avec un rapport de bits moyen de 60,6 %, contre respectivement 80,6 %, 92,2 % et 76,21 % pour les trois approches de l'outil Hippo.

7.13 Contribution à la compétition MCC

L'édition 2022 du MCC (voir la section 3.8) comporte 1 628 réseaux de Petri, dont 705 peuvent recevoir une structure NUPN (ils sont non-colorés, ordinaires et saufs). Notre chaîne d'outils a été employée sur les 208 réseaux qui ne contenaient aucune structure NUPN, ainsi que sur deux réseaux ayant une information NUPN fortement sous-optimale (« IBM319 » et « IBM703 »).

Chaque réseau a été décomposé une première fois, avec une matrice des places concurrentes éventuellement incomplète. Si tel est le cas, une seconde matrice des places concurrentes a été calculée, mais cette fois depuis le réseau décomposé, l'algorithme C_1 étant (potentiellement) accéléré par la structure NUPN (voir la section 3.4) apportée par la décomposition. Si l'algorithme C_1 ne peut toujours pas parcourir exhaustivement les marquages accessibles, la seconde matrice contient les mêmes paires de places non-concurrentes que la première matrice (ces paires venant des algorithmes C_2 et C_4) : il serait vain d'essayer de procéder à une seconde décomposition. Sinon, l'algorithme C_1 se termine, la seconde matrice est complète et nous avons entrepris une seconde décomposition.

Souvent, la seconde décomposition a un coût en bits et un nombre d'unités identiques à ceux de la première décomposition. Ceci montre que les algorithmes C_2 et C_4 repèrent suffisamment de paires de places non-concurrentes pour produire des décompositions de bonne qualité, est-ce malgré des matrices des places concurrentes incomplètes.

Notre chaîne d'outils a été en mesure de décomposer 208 de ces 210 instances (soit 99,0 %). Ce chiffre comprend les six instances du modèle « Eratosthenes », qui n'ont que des décompositions triviales : pour cette raison, ce modèle est resté non-décomposé dans la collection du MCC ; ce chiffre comprend également deux instances du modèle « EGFr » qui étaient isomorphes : pour éliminer ce doublon du MCC, une des deux instances a été laissée sans décomposition.

Par rapport aux réseaux non-décomposés, les réseaux décomposés produits par notre chaîne d'outils ont un ratio moyen d'unités de 35,4 % et un ratio moyen de bits de 47,2 %.

7.14 La suite de jeux de tests VLSAT

En décomposant notre collection de réseaux de Petri (voir la section 3.8), des centaines de milliers de formules en logique propositionnelle ou du premier ordre ont été produites. Nous

avons minutieusement sélectionné ces formules pour produire une suite de jeux de tests, intitulée VLSAT, acronyme de « Very Large Boolean SATisfiability ». L'objectif de VLSAT est de proposer un ensemble de formules de complexité croissante et difficiles à résoudre en raison de leurs tailles. Cette suite est proposée en trois collections :

— VLSAT-1 [BG20] contient cent formules en logique propositionnelle, au format DIMACS CNF, toutes valides (c.-à-d. satisfaisables) et présentant un nombre élevé de modèles, ce qui les rend intéressantes pour les compétitions de comptage de modèles. Les formules de VLSAT-1 ont été utilisées pour l'édition 2020 de la compétition internationale sur le comptage de modèles [18].

— VLSAT-2 [BG21c] contient cent formules en logique propositionnelle, au format DIMACS CNF, dont cinquante valides et cinquante invalides. Ces formules sont vouées aux compétitions de validité booléenne, telles que la compétition internationale sur la résolution de formules propositionnelles [19]. Cette compétition cherche des formules difficiles pour le résolveur MiniSAT, mais faisables pour les résolveurs dans l'état de l'art. En 2020 et 2021, nous avons envoyé un sous-ensemble de VLSAT-2 à ses organisateurs [BG21b], qui les ont classées comme intéressantes. Au total, les éditions 2020 et 2021 de cette compétition ont utilisé le quart des tests de VLSAT-2.

— VLSAT-3 [Bou21] contient 1200 formules au format SMT-LIB, dans six fragments de logiques du premier ordre (QF_BV, QF_DT, QF_IDL, QF_UFBV, QF_UFDT, QF_UFIDL), chacun de ces six fragments ayant cent formules valides et cent invalides. Les tests de VLSAT-3 ont été intégrés dans la collection de tests de la compétition internationale de satisfaisabilité modulo des théories. Plus de 90 % des tests de VLSAT-3 ont été utilisés lors de la seizième édition de cette compétition, qui s'est déroulée en 2021.

Les trois collections VLSAT sont disponibles sous une licence Creative Commons à l'adresse suivante : `cadp.inria.fr/resources/vlsat`.

7.15 Bilan et perspectives

Pour le problème de la décomposition, qui est assez ancien puisqu'il remonte à près de cinquante ans, nous avons présenté diverses approches permettant de traduire de manière automatisée tout réseau de Petri ordinaire et sauf en un NUPN plat et unit-sauf. Ces traductions préservent toutes les propriétés structurelles et comportementales du modèle d'entrée ; en particulier, elles ne créent ni ne suppriment aucune place, transition ou arc ; seule la topologie se voit remaniée, en regroupant les places non-concurrentes dans des unités. Pour ce problème, les principes et les résultats issus de la théorie des NUPN se sont avérés être remarquablement bien adaptés.

Nos approches ont été implantées dans une chaîne d'outils cohérente et complète, acceptant

18. `mccompetition.org`
19. `satcompetition.github.io`

et produisant des modèles au format normalisé PNML. La chaîne d'outils tire profit des avancées récentes concernant les algorithmes de graphes et les résolveurs de formules logiques. Sur une vaste collection de 12 728 modèles d'origine très diverse, nos approches ont fait preuve de taux de réussite élevés, allant de 87,3 % à 99,2 %. Notre chaîne d'outils a été mise en œuvre sur les modèles exigeants de l'édition 2022 du MCC et a obtenu un taux de réussite de 99 %, ce qui constitue à la fois une preuve de pertinence en théorie et une démonstration de son efficacité en pratique.

La chaîne d'outils est utile pour restructurer automatiquement des réseaux de Petri « anciens » et les « mettre à niveau » en NUPN, en inférant leur structure concurrente (cachée ou perdue) ; ces informations structurelles, qui existaient lorsque les modèles ont été conçus mais ont été oubliées depuis, augmentent considérablement l'efficacité des outils de vérification formelle. Il est également probable que notre chaîne d'outils puisse être utile pour d'autres applications, comme l'extraction de processus au sein des réseaux de flux de travaux [MCvdA13].

Comme le problème de décomposition est ardu et que plusieurs étapes de notre chaîne d'outils sont NP-complets ou PSPACE-complets, il subsistera toujours des réseaux de Petri de taille arbitrairement grande, qui ne pourront pas être entièrement décomposés. Il s'agit toutefois d'une complexité asymptotique, n'infirmant pas les résultats de nos approches en pratique.

À ce propos, bien que calcul de la matrice des places concurrentes (voir le chapitre 6) soit un ingrédient essentiel pour nos décompositions, et que l'exploration des marquages accessibles (voir l'algorithme C_1) puisse s'avérer coûteuse pour certains réseaux (car PSPACE-complète), cette difficulté n'est pas rédhibitoire : à la section 7.12, les réseaux du MCC ont été décomposés en deux fois, la première à l'aide d'une matrice pouvant être incomplète, et la seconde à l'aide d'une matrice complète.

Nous pouvons envisager plusieurs pistes pour des travaux ultérieurs.

La modularité de notre chaîne d'outils permet des évolutions futures, dans lesquelles certains composants pourraient être substitués par d'autres composants, plus efficaces.

Nos approches de décomposition se prêtent bien à des approches de type « portefeuille », qui consistent à lancer plusieurs approches de décomposition en parallèle, pour ne garder que le meilleur résultat.

Les approches reposant sur la coloration de graphes ou la résolution de formules logiques peuvent aussi être parallélisées : il est possible d'appeler en parallèle plusieurs instances de l'outil de résolution sous-jacent, avec différents nombres maximaux d'unités, plutôt que de l'appeler séquentiellement (en utilisant la stratégie de recherche linéaire décroissante de la sous-section 7.4.5). En outre, il est possible d'invoquer en parallèle plusieurs algorithmes de recherche de clique maximum.

Actuellement, nous n'utilisons que des algorithmes donnant des solutions exactes pour les décompositions reposant sur le coloriage de graphes ou la recherche itérée de clique maximum. Nous pourrions envisager des algorithmes approchés, en remplacement ou en

complément des algorithmes exacts existants. Par exemple, il est possible d'accélérer les décompositions reposant sur le coloriage de graphes en commençant la recherche linéaire avec un algorithme approché, puis en basculant vers un algorithme exact.

Enfin, lorsqu'une approche reposant sur de la recherche itérée de clique maximum a consommé tout son temps imparti, les places qui ne sont pas encore traitées sont ajoutées à la décomposition en créant une unité par place. Plutôt que de recourir à un tel comportement, il est possible d'envisager ici un algorithme de type glouton, calculant des cliques maximales (en temps polynomial) pour allouer les dernières unités.

Une optimisation de type « diviser pour régner », permettant à la fois des gains en temps de traitement et en qualité de décomposition, pourrait être envisagée pour les réseaux ayant un graphe de concurrence non connexe (ce qui est le cas de 6,3 % des réseaux de notre collection). Dans ces réseaux, il est possible de décomposer séparément chaque *composante connexe* du graphe de concurrence, autrement dit chaque sous-graphe connexe maximal (ces réseaux non connexes ont en moyenne 60,2 places et ont en moyenne 3,1 composantes connexes). Dès lors que toutes les composantes connexes ont été décomposées en unités, il est possible de regrouper les places locales d'unités appartenant à des composantes connexes distinctes dans la même unité. Une telle opération permet de construire une décomposition pour l'ensemble du réseau de Petri. Une heuristique bien choisie pour cette étape de fusion permet en outre d'améliorer la qualité de décomposition finale en terme de codage des marquages accessibles.

Chapitre 8

Décomposition de réseaux de Petri en NUPN hiérarchiques

Ce chapitre traite de la décomposition d'un réseau de Petri ordinaire et sauf en un NUPN unit-sauf, mais contrairement au chapitre 7, non nécessairement plat : la décomposition produite peut avoir une hauteur supérieure à une. Deux approches de décomposition reposant sur la matrice des places concurrentes (voir le chapitre 6) sont proposées.

8.1 Motivations

Le chapitre 7 a proposé des approches pour la traduction automatisée d'un réseau de Petri ordinaire et sauf en un réseau équivalent d'automates s'exécutant de manière concurrente. Ces approches présentent plusieurs avantages, les principales sont qu'elles sont automatisées (pour déterminer quelles places peuvent être situées dans le même automate, elles s'appuient sur les places concurrentes du chapitre 6), rapides et offrent des taux élevés de réussite (99,2 % des réseaux de la section 3.8 peuvent être décomposés en moins d'une minute). Mais ces décompositions sont plates, et les trois raisons suivantes motivent l'étude des décompositions hiérarchiques.

Premièrement, avec leurs unités imbriquées formant des arbres, les NUPN sont intrinsèquement hiérarchiques. Ce fait est exploité pour la conception de codages réduisant le nombre de variables booléennes requises pour représenter les marquages accessibles (voir la section 3.4). Les NUPN plats ne bénéficient pas de la hiérarchie et n'ont donc pas nécessairement des codages optimaux.

Deuxièmement, une décomposition hiérarchique est plus lisible pour les humains, car les unités imbriquées permettent d'exprimer adéquatement la relation entre les unités mères et filles. Ainsi, la figure 8.1 montre deux NUPN unit-saufs ayant le même réseau de Petri sous-jacent. Seul le NUPN de la partie droite représente une décomposition hiérarchique et permet de distinguer convenablement les unités mères et filles des transitions T et T', qui

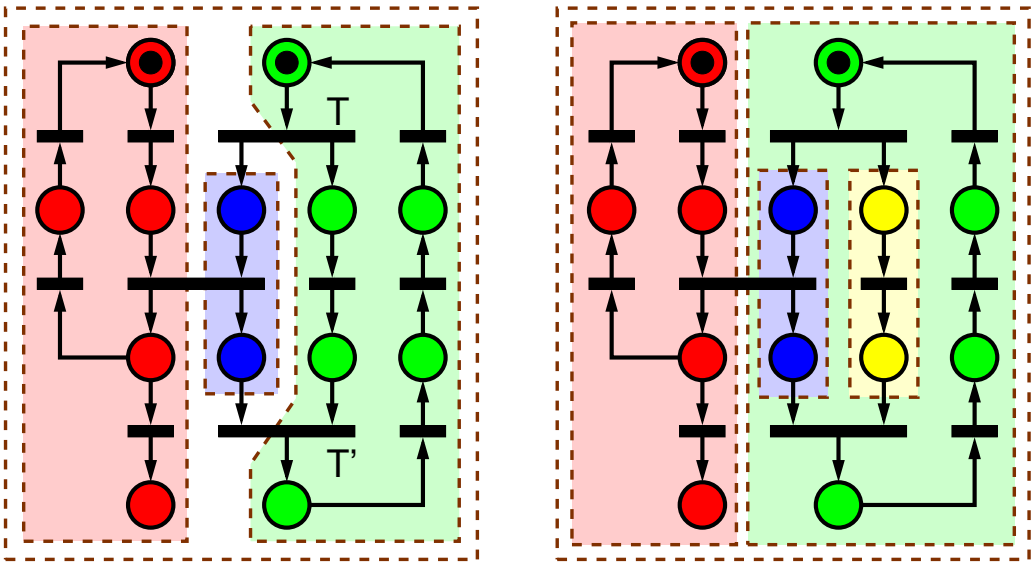

FIGURE 8.1 – NUPN plat (à gauche) versus NUPN hiérarchique (à droite)

sont des transitions « fourchette » et « jointure ».

Troisièmement, les unités du formalisme NUPN permettent de représenter concrètement des processus, en faisant le lien entre réseaux de Petri et algèbres de processus. Or, les algèbres de processus usuelles permettent d'entrelacer librement les opérateurs séquentiels et parallèles : une procédure de décomposition compatible avec les réseaux de Petri devrait donc être en mesure de produire des NUPN avec des unités imbriquées à une profondeur arbitraire.

Pour toutes ces raisons, nous cherchons ici, contrairement aux décompositions en automates communicants (où le mélange des automates pères et fils était acceptable) des approches alternatives distinguant correctement les processus parents de leurs enfants.

La suite de ce chapitre est organisée comme suit. La section 8.2 présente le problème de décomposition hiérarchique et discute de la forme des réseaux acceptés en entrée et produits en sortie. La section 8.3 situe ce chapitre par rapport aux travaux existants. La section 8.4 expose un critère pour une solution « optimale ». Les sections 8.5 et 8.6 proposent deux approches de décomposition hiérarchiques. La section 8.7 présente nos résultats expérimentaux sur une collection de 16 200 modèles. La section 8.8 aborde les démarches entreprises pour la validation des résultats. Finalement, la section 8.9 formule des observations conclusives et évoque les perspectives ouvertes.

8.2 Énoncé du problème

Dans ce chapitre, nous cherchons à apporter simultanément la modularité (voir la section 1.5) et la hiérarchie (voir la section 1.6) aux réseaux de Petri.

Nous considérons le problème de décomposition comme une opération dans le domaine des réseaux de Petri, acceptant en entrée un réseau et produisant un réseau décomposé en sortie. Ce dernier doit avoir un réseau de Petri sous-jacent isomorphe (voir la définition 4.3) au réseau d'entrée, autrement dit, préserver toutes les places, transitions, arcs et le marquage initial du réseau d'entrée. La décomposition doit ainsi permettre d'ajouter au réseau d'entrée une structure en processus imbriqués hiérarchiquement, mais en préservant toutes les propriétés structurelles et comportementales du réseau existant.

Ce problème consiste alors à transformer tout réseau de Petri ordinaire et sauf, ou tout réseau d'automates communicants (voir le chapitre 7), en un NUPN unit-sauf (voir la section 1.7).

Évidemment, si le réseau d'entrée correspond simplement à une machine à états ou à un réseau d'automates, le NUPN résultant sera vraisemblablement plat.

Comme pour les décompositions plates, le problème de décomposition de ce présent chapitre possède toujours au moins une solution (certes, de faible utilité pratique) : il s'agit du NUPN trivial (voir la figure 3.3).

Comme pour les décompositions en automates communicants, il peut exister plusieurs décompositions hiérarchiques pour un même réseau de Petri d'entrée ; cette multiplicité est même augmentée par la nature combinatoire de l'imbrication des unités. Dans le pire cas, le nombre de décompositions valides peut être exponentiel par rapport au nombre de places : il faut, là encore, chercher une décomposition optimisant un critère de qualité approprié.

8.3 Travaux voisins

Au-delà des approches de décomposition plates mentionnées à la section 8.1, plusieurs travaux abordent le problème de la décomposition hiérarchique.

L'algorithme proposé par [Esh09] permet de convertir des réseaux de Petri en diagrammes d'états UML (voir la section 3.5). Comme les NUPN, ces diagrammes permettent de modéliser des systèmes parallèles et sont intrinsèquement hiérarchiques. Mais bien que cet algorithme soit correct et possède une complexité polynomiale, il ne peut traiter que les réseaux ayant des *couvertures emboîtables*. La classe des réseaux respectant cette propriété est trop restrictive (elle n'inclut pas, par exemple, les réseaux à choix libres) pour notre besoin de traiter l'ensemble des réseaux de Petri ordinaires et saufs.

Le vérificateur de modèles ITS-Tools peut décomposer de manière hiérarchique [ABC+19, sect 2.2, « modularité de Louvain »] le réseau de Petri en entrée, afin de représenter plus efficacement ses marquages accessibles lors de l'utilisation de diagrammes de décision d'ensembles hiérarchiques [TMPHK09]. Mais ces décompositions, issues de la modularité de Louvain, enfreignent la propriété unit-sauf.

L'*extraction de processus* (en anglais, « process mining ») consiste en la synthèse d'un modèle (généralement un réseau de Petri) décrivant un ensemble de traces, également

appelé *journal*. Le contrôle de conformité, à savoir la comparaison du réseau généré et du journal, est une étape cruciale pour vérifier la correction du modèle, mais difficile en raison de coûts de calcul prohibitifs découlant de l'explosion de l'espace d'états. Afin de palier ce problème, [MCvdA13] propose une décomposition hiérarchique, réalisée en identifiant dans un graphe orienté (dont les sommets sont les places et les transitions du réseau, et les arcs sont ceux du réseau) les régions à « entrée unique et sortie unique » et leurs imbrications. Mais une telle approche de décomposition ne garantit pas la propriété unit-sauf, en dehors de classes de réseaux très rudimentaires, comme celle des *graphes marqués*.

Un algorithme de décomposition hiérarchique pour les réseaux ordinaires, vivants et saufs est proposé par [KA06, Kar07]. Cet algorithme repose sur la construction itérée de macro-places, permettant ainsi la construction d'une décomposition hiérarchique de « bas en haut ». La découverte des macro-places est réalisée par la détection de sous-réseaux intitulés *blocs-P*, lesquels possèdent des places d'entrée ou de sortie, ce qui les rapproche des réseaux ouverts (voir notamment [MOW09]), bien que, dans ces derniers, une place ne peut être à la fois d'entrée et de sortie. Mais, si le réseau d'entrée n'est pas vivant, l'algorithme doit recourir à une exploration des marquages accessibles. De plus, la décomposition produite par l'algorithme ne respecte pas la propriété unit-sauf.

8.4 Critère d'optimalité

La proposition suivante montre que la minimisation du nombre d'unités (utilisée au chapitre 7 pour les décompositions plates) n'est pas pertinente pour évaluer des approches de décompositions hiérarchiques.

Proposition 8.1 *Pour un réseau de Petri donné, une décomposition minimisant le nombre d'unités est toujours une décomposition plate.*

Preuve. Soit un NUPN N. Soit le NUPN N' produit par application de la proposition 3.4 à N. Par construction : (i) les réseaux de Petri sous-jacents de N et N' sont isomorphes (voir la définition 4.3) ; (ii) N' est un NUPN plat et (iii) soit $N = N'$, soit N' a strictement moins d'unités que N. □

Pour évaluer la « qualité » d'une décomposition hiérarchique, le critère du nombre minimal de bits pour coder tout marquage accessible (voir la section 3.4) est plus pertinent. Nous adoptons le codage le plus compact de [Gar19, Sect. 6] pour les NUPN unit-saufs, à savoir le *schéma (b) avec chevauchement*, dont le coût est exprimé par la proposition suivante :

Proposition 8.2 *Soit un NUPN unit-sauf* $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$. *Le coût en bits du codage de* N *vaut* $\nu(u_0)$, *où la fonction* $\nu : U \to \mathbb{N}$ *est définie comme suit :*

$$\forall u \in U, \nu(u) \stackrel{\text{def}}{=} \begin{cases} \text{si } u \textit{ est une unité feuille alors } \lceil \log_2(|\mathsf{places}(u)| + 1) \rceil \\ \text{sinon si } u \textit{ n'a pas de place locale alors } \sum_{u' \in \mathsf{sous\text{-}unités}(u)} \nu(u') \\ \text{sinon } 1 + \max\left(\lceil \log_2 |\mathsf{places}(u)| \rceil, \sum_{u' \in \mathsf{sous\text{-}unités}(u)} \nu(u') \right) \end{cases}$$

Plus précisément, $\nu(u)$ donne le nombre de bits utilisés pour coder l'intersection de tout marquage accessible M avec les places globales de l'unité u, autrement dit, $M \cap \mathsf{places}^*(u)$. Notons $m_u = |\mathsf{places}(u)|$ le nombre de places locales de u :

- Si u est une unité feuille, dans tout marquage accessible, au plus une place locale de u peut comporter un jeton (N est unit-sauf), ou bien elle peut ne pas posséder de jeton du tout ; ces $m_u + 1$ possibilités peuvent être codées en utilisant $\lceil \log_2(m_u + 1) \rceil$ bits.
- Si u n'est pas une unité feuille et que $m_u = 0$, pour tout marquage accessible M, l'intersection $M \cap \mathsf{places}^*(u)$ peut être exprimée comme l'union disjointe $\biguplus_{u' \in \mathsf{sous\text{-}unités}(u)}(M \cap \mathsf{places}^*(u'))$; chaque marquage $M \cap \mathsf{places}^*(u')$ étant lui-même unit-sauf, le nombre de bits $\nu(u)$ est la somme de tous les $\nu(u')$.
- Si u n'est pas une unité feuille et que $m_u > 0$, soit il y a exactement un jeton dans une place locale de u et aucun jeton dans les sous-unités (de manière transitive) de u, soit il n'y a aucun jeton dans les places locales de u et éventuellement des jetons dans les sous-unités (de manière transitive) de u ; pour représenter ce choix, le codage utilise une variable supplémentaire.

8.5 Approche structurelle

Cette approche prend en entrée un réseau de Petri ordinaire et sauf, ainsi que les paires de places que l'on sait avec certitude être non-concurrentes (à savoir, l'ensemble $R_\perp$ calculé au chapitre 6).

Nous définissons d'abord deux relations binaires sur les places, $\preceq_P$ et $\approx_P$. Intuitivement, $\approx_P$ exprime que deux places peuvent être dans la même unité et $\preceq_P$ exprime que deux places peuvent être dans des unités imbriquées.

Définition 8.1 *Soit $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ un NUPN et $R_\perp$ un sous-ensemble des paires de places non-concurrentes. Les relations $\preceq_P$ et $\approx_P$ sont alors définies sur P^2 par :*

- $p_1 \preceq_P p_2 \stackrel{\text{def}}{=} (p_1 = p_2) \vee ((\{p_1, p_2\} \in R_\perp) \wedge (\forall p_3 \in P \setminus \{p_1, p_2\}, (\{p_2, p_3\} \notin R_\perp) \Rightarrow (\{p_1, p_3\} \notin R_\perp))$;
- $p_1 \approx_P p_2 \stackrel{\text{def}}{=} (p_1 \preceq_P p_2) \wedge (p_2 \preceq_P p_1)$.

La décomposition hiérarchique reposant sur les relations $\preceq_P$ et $\approx_P$ est plus exigeante que la décomposition plate, présentée au chapitre 7 : dans une décomposition plate, deux places p_1 et p_2 peuvent être mises dans la même unité *ssi* $\{p_1, p_2\} \in R_\perp$; la relation $\approx_P$ exige, de surcroît, que p_1 et p_2 soient pareillement concurrentes (ou non-concurrentes) par rapport à toute autre place p_3.

Proposition 8.3 *La relation $\preceq_P$ (voir la définition 8.1) est réflexive et transitive.*

Preuve. Par construction, $\forall p \in P, p \preceq_P p$. De ce fait, la relation $\preceq_P$ est réflexive. Montrons que $\preceq_P$ est transitive. Soient p_1, p_2, p_3 trois places, deux à deux distinctes, telles que $p_1 \preceq_P p_2$

et $p_2 \preceq_P p_3$. Montrons que $p_1 \preceq_P p_3$. Nous avons :

(i) $\{p_1, p_2\} \in R_\perp$;

(ii) $\forall p \in P \setminus \{p_1, p_2\}, (\{p_2, p\} \notin R_\perp) \Rightarrow (\{p_1, p\} \notin R_\perp)$;

(iii) $\{p_2, p_3\} \in R_\perp$;

(iv) $\forall p' \in P \setminus \{p_2, p_3\}, (\{p_3, p'\} \notin R_\perp) \Rightarrow (\{p_2, p'\} \notin R_\perp)$.

En prenant $p' = p_1$ dans (iv), nous obtenons $(\{p_1, p_3\} \notin R_\perp) \Rightarrow (\{p_1, p_2\} \notin R_\perp)$. Puisque (i), nous avons $\{p_1, p_3\} \in R_\perp$. En prenant $p = p'$ dans (ii) et (iv), nous obtenons $\forall p \in P \setminus \{p_1, p_2, p_3\}, (\{p_3, p\} \notin R_\perp) \Rightarrow (\{p_1, p\} \notin R_\perp)$. Comme (i) et (iii), nous pouvons autoriser $p = p_2$ dans l'équation précédente, et obtenons $\forall p \in P \setminus \{p_1, p_3\}, (\{p_3, p\} \notin R_\perp) \Rightarrow (\{p_1, p\} \notin R_\perp)$. Ainsi, $\preceq_P$ est transitive. □

La relation $\preceq_P$ n'est pas un ordre partiel, car, bien qu'elle soit réflexive et transitive (voir la proposition 8.3), elle n'est pas antisymétrique. Remarquons en outre que $R_\perp$ n'est pas, elle-même, une relation d'équivalence, puisqu'il s'agit d'un sous-ensemble de la relation $\nparallel$, qui n'est pas nécessairement réflexive et transitive.

Proposition 8.4 *La relation $\approx_P$ (voir la définition 8.1) est une relation d'équivalence.*

Preuve. Par définition, $\forall p_1, p_2 \in P, (p_1 \approx_P p_2) \Leftrightarrow (p_2 \approx_P p_1)$. Conséquemment, la relation $\approx_P$ est symétrique. La réflexivité et la transitivité de la relation $\approx_P$ découlent de la réflexivité et la transitivé de la relation $\preceq_P$ (voir la proposition 8.3). □

Nous introduisons désormais la notion de *structure hiérarchique*, qui formalisme la notion de décomposition hiérarchique basée sur les relations $\preceq_P$ et $\approx_P$.

Définition 8.2 *Soit (P, T, F, M_0) un réseau de Petri et $R_\perp$ un sous-ensemble des paires de places non-concurrentes. Sa* structure hiérarchique *correspondante est un quadruplet $(U, u_0, \preceq_U, \mathsf{unité})$ tel que :*

- *$U \stackrel{\text{def}}{=} (P/\approx_P) \cup \{u_0\}$ est un ensemble (fini) d'unités, défini comme étant l'ensemble quotient de P par rapport à $\approx_P$, duquel l'unité racine u_0 est ajoutée si elle est absente ;*
- *$\mathsf{unité} : P \to U$ est une fonction associant à chaque place sa classe d'équivalence selon la relation $\approx_P$;*
- *$\preceq_U$ est une relation sur U, définie par :*
  (i) $\forall u \in U, u \preceq_U u_0$; autrement dit, l'unité racine est la plus grande unité ;
  (ii) $\forall u \in U, (u_0 \preceq_U u) \Rightarrow (u = u_0)$; autrement dit, la plus grande unité est unique ;
  (iii) $\forall (p_1, p_2) \in P \times P, (p_1 \preceq_P p_2) \Leftrightarrow (\mathsf{unité}(p_1) \preceq_U \mathsf{unité}(p_2))$; autrement dit, la relation $\preceq_P$ est transposée des places vers les unités (ses classes d'équivalences) pour donner la relation $\preceq_U$.

Proposition 8.5 *Étant donné (P, T, F, M_0) un réseau de Petri et $R_\perp$ un sous-ensemble des paires de places non-concurrentes, la structure hiérarchique $(U, u_0, \preceq_U, \mathsf{unité})$ de la définition 8.2 existe toujours et est unique.*

Preuve. Puisque $\approx_P$ est une relation d'équivalence sur P, l'ensemble U et la fonction $\mathsf{unité}$ existent. En raison de l'unicité de la relation $\approx_P$, il existe une unique solution pour U et $\mathsf{unité}$ (modulo le choix de u_0). En raison de l'unicité de la relation $\preceq_P$ et étant donné les définitions de U et $\mathsf{unité}$, il existe une unique relation $\preceq_U$. La définition de cette relation assure que u_0 est unique et est la plus grande unité de U. Puisque U, u_0, $\preceq_U$ et $\mathsf{unité}$ existent et sont uniques, il en est de même pour $(U, u_0, \preceq_U, \mathsf{unité})$. □

Proposition 8.6 *Soit (P, T, F, M_0) un réseau de Petri, $R_\perp$ un sous-ensemble des paires de places non-concurrentes et $(U, u_0, \preceq_U, \mathsf{unité})$ la structure hiérarchique de la définition 8.2. La relation $\preceq_U$ est un ordre partiel sur l'ensemble des unités U.*

Preuve. Par définition, $\forall u \in U, u \preceq_U u$, ainsi, la relation $\preceq_U$ est réflexive. Étant donné que U est l'ensemble quotient $P/\approx_P$, nous avons : $\forall u_1, u_2 \in U, ((u_1 \preceq_U u_2) \wedge (u_2 \preceq_U u_1)) \Rightarrow u_1 = u_2$. Par conséquent, la relation $\preceq_U$ est antisymétrique. Montrons que la relation $\preceq_U$ est transitive. Soient u_1, u_2, u_3 trois unités, deux à deux distinctes, telles que $u_1 \preceq_U u_2$ et $u_2 \preceq_U u_3$, nous avons :

(i) $\forall p_1, p_2 \in P, (\mathsf{unité}(p_1) = u_1) \wedge (\mathsf{unité}(p_2) = u_2) \Rightarrow (\{p_1, p_2\} \in R_\perp)$;

(ii) $\forall u \in U \setminus \{u_1, u_2\}, \forall p, p_1, p_2 \in P, (\mathsf{unité}(p) = u) \wedge (\mathsf{unité}(p_1) = u_1) \wedge (\mathsf{unité}(p_2) = u_2) \wedge (\{p_2, p\} \notin R_\perp) \Rightarrow (\{p_1, p\} \notin R_\perp)$;

(iii) $\forall p_2, p_3 \in P, (\mathsf{unité}(p_2) = u_2) \wedge (\mathsf{unité}(p_3) = u_3) \Rightarrow (\{p_2, p_3\} \in R_\perp)$;

(iv) $\forall u' \in U \setminus \{u_2, u_3\}, \forall p', p_2, p_3 \in P, (\mathsf{unité}(p') = u') \wedge (\mathsf{unité}(p_2) = u_2) \wedge (\mathsf{unité}(p_3) = u_3) \wedge (\{p_3, p'\} \notin R_\perp) \Rightarrow (\{p_2, p'\} \notin R_\perp)$.

En prenant $u' = u_1$ dans (iv), nous obtenons $\forall p_1, p_2, p_3 \in P, (\mathsf{unité}(p_1) = u_1) \wedge (\mathsf{unité}(p_2) = u_2) \wedge (\mathsf{unité}(p_3) = u_3) \wedge (\{p_1, p_3\} \notin R_\perp) \Rightarrow (\{p_1, p_2\} \notin R_\perp)$. Puisque (i), nous avons également $\forall p_1, p_3 \in P, (\mathsf{unité}(p_1) = u_1) \wedge (\mathsf{unité}(p_3) = u_3) \Rightarrow (\{p_1, p_3\} \in R_\perp)$.

En prenant $u = u'$ dans (ii) et (iv), nous obtenons $\forall u \in U \setminus \{u_1, u_2, u_3\}), \forall p, p_1, p_3 \in P, (\mathsf{unité}(p) = u) \wedge (\mathsf{unité}(p_1) = u_1) \wedge (\mathsf{unité}(p_3) = u_3) \wedge (\{p_3, p\} \notin R_\perp) \Rightarrow (\{p_1, p\} \notin R_\perp)$. Puisque (iii), nous pouvons autoriser $u = u_2$ dans l'équation précédente ; en conséquence $\forall u \in U \setminus \{u_1, u_3\}, \forall p, p_1, p_3 \in P, (\mathsf{unité}(p) = u) \wedge (\mathsf{unité}(p_1) = u_1) \wedge (\mathsf{unité}(p_3) = u_3) \wedge (\{p_3, p\} \notin R_\perp) \Rightarrow (\{p_1, p\} \notin R_\perp)$. Ainsi, $\preceq_U$ est transitive et est donc un ordre partiel. □

La définition suivante exprime la notion de *place isolée*, qui généralise les places mortes. Il s'agit des places qui ne peuvent être concurrentes qu'avec elles-mêmes, si elles ne sont pas mortes : elles peuvent donc être situées dans toute unité d'un NUPN unit-sauf.

Définition 8.3 *Soit (P, T, F, M_0) un réseau de Petri. Une place p est* isolée ssi $\forall p' \in P, \Re(\{p, p'\}) \Rightarrow (p = p')$, *autrement dit* ssi *$p$ est morte ou le seul marquage accessible ayant un jeton dans p est le singleton $\{p\}$.*

Notons qu'au moins 94,1 % de nos réseaux (voir la section 3.8) ont des places isolées, que nos réseaux comportent en moyenne 34,1 % de places isolées et qu'au total, au moins 27,6 % des places de notre collection sont isolées.

La définition ci-dessous précise la condition nécessaire et suffisante pour qu'une place soit

une place locale de u_0 dans la structure hiérarchique (voir la définition 8.2). Cette définition permet d'expliquer pourquoi, dans la définition de U, l'union des termes $(P/\approx_P)$ et $\{u_0\}$ n'est pas nécessairement disjointe (dans le cas où une place isolée est connue dans $R_\perp$), mais qu'elle est requise (notamment, en cas d'absence de place isolée).

Définition 8.4 *Les places qui peuvent être déterminées comme isolées par une lecture de $R_\perp$ sont l'ensemble des places locales de l'unité u_0 produite par la structure hiérarchique. Autrement dit,* $\mathsf{places}\,(u_0) = \{p \in P \mid (P \setminus \{p\}) \otimes \{p\} \subseteq R_\perp\}$.

Preuve. Une place connue (par une lecture de $R_\perp$) pour être isolée est la plus grande place selon la relation $\preceq_P$. □

La structure hiérarchique peut contenir plusieurs décompositions possibles, qui sont toutes exprimées par la même relation d'ordre partiel $\preceq_U$. Nous devons choisir une décomposition particulière parmi celles-ci.

Proposition 8.7 *Soit (P, T, F, M_0) un réseau de Petri, $R_\perp$ un sous-ensemble des paires de places non-concurrentes, et $(U, u_0, \preceq_U, \mathsf{unité})$ la structure hiérarchique sur $(N, R_\perp)$. Pour toute relation $\sqsubseteq$ telle que $(U, \sqsupseteq)$ est un arbre couvrant de $(U, \succeq_U)$, le NUPN $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ est unit-sauf.*

Preuve. Montrons, dans un premier temps, que le 8-uplet $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ est un NUPN (voir la définition 3.1) : (i) seule l'unité u_0 peut être vide ; (ii) l'ensemble des unités non-vides forment une partition des places ; (iii) le couple $(U, \sqsupseteq)$ forme un arbre des unités ; (iv) l'ensemble U possède un unique plus grand élément selon $\sqsubseteq$, qui est u_0. Ainsi, N est un NUPN. Pour chaque paire de places $\{p_1, p_2\}$ appartenant à des unités non-disjointes, nous avons $(p_1 \preceq_P p_2) \vee (p_2 \preceq_P p_1)$, et ainsi $\{p_1, p_2\} \in R_\perp$ (d'après la définition 8.1). Ainsi, N est un NUPN unit-sauf. □

Lors du choix d'un arbre couvrant particulier, nous recourons à des heuristiques qui sélectionnent, pour chaque unité, une unique unité parente parmi celles de $\preceq_U$. S'il existe une unité parente n'ayant qu'une seule sous-unité, elle est sélectionnée en priorité. Sinon, une unité parente ayant le plus grand nombre de places locales est choisie. De plus, pour réduire le nombre d'arbres couvrants, nous optimisons la structure hiérarchique en appliquant de façon répétée les deux règles suivantes, jusqu'à saturation : (i) chaque unité ayant une seule unité parente est fusionnée avec celle-ci ; (ii) toute paire d'unités non-concurrentes ayant les mêmes unités parentes et les mêmes sous-unités sont fusionnées.

8.6 Approche gloutonne

Cette approche prend en entrée un NUPN unit-sauf $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$, ainsi que les paires de places que l'on sait avec certitude être non-concurrentes (à savoir, l'ensemble $R_\perp$ calculé au chapitre 6).

Tout d'abord, le NUPN d'entrée est transformé en déplaçant vers l'unité racine toutes les places isolées (voir la définition 8.3). Cette transformation préserve la propriété unit-sauf

du NUPN (l'unité racine étant la seule unité non-disjointe avec toutes les autres unités, elle est la seule à ne pouvoir contenir que des places isolées). La définition suivante formalise cette transformation.

Définition 8.5 *Soit* $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN et* $R_\perp$ *un sous-ensemble des paires de places non-concurrentes. La première itération de l'approche gloutonne produit un NUPN* $(P, T, F, M_0, U', u_0, \sqsubseteq', \mathsf{unité}')$*, tel que :*

— $U' \stackrel{\text{def}}{=} \{\mathsf{unité}'(p) \mid p \in P\} \cup \{u_0\}$;

— $\forall p_1 \in P, \mathsf{unité}'(p_1) \stackrel{\text{def}}{=} \begin{cases} \mathsf{si}\ \forall p_2 \in P, (p_1 = p_2) \vee (\{p_1, p_2\} \in R_\perp)\ \mathsf{alors}\ u_0 \\ \mathsf{sinon}\ \mathsf{unité}\,(p_1) \end{cases}$;

— $\sqsubseteq' \stackrel{\text{def}}{=} (U' \times U') \cap \sqsubseteq$.

Proposition 8.8 *Si le NUPN d'entrée est unit-sauf, alors le NUPN produit par la définition 8.5 est également unit-sauf.*

Preuve. La première itération de l'approche gloutonne (voir la définition 8.5) déplace toutes les places isolées (voir la définition 8.3) dans l'unité racine. Le reste de la structure NUPN est inchangée. De ce fait, puisque le NUPN d'entrée est unit-sauf, alors le NUPN de sortie l'est également. □

L'approche gloutonne procède ensuite par itérations successives, en transformant, lors de chaque itération, le NUPN unit-sauf courant en vue de construire une meilleure structure NUPN. Dans la mesure où plusieurs transformations sont possibles à chaque étape, nous utilisons une fonction d'objectif η pour décider quelle transformation sera optée.

Définition 8.6 *Soit* $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN et* $R_\perp$ *un sous-ensemble des paires de places non-concurrentes. Pour toutes unités* $u_1, u_2 \in U$*, nous définissons :*

— $\theta\,(u_1, u_2) \stackrel{\text{def}}{=} \{p_1 \in \mathsf{places}\,(u_1) \mid \forall p_2 \in \mathsf{places}^*(u_2) \setminus \{p_1\}, \{p_1, p_2\} \in R_\perp\}$;

— $\eta\,(u_1, u_2) \stackrel{\text{def}}{=} \theta\,(u_1, u_2) \cup \theta\,(u_2, u_1)$.

Ici, $\eta\,(u_1, u_2)$ désigne toutes les places locales de u_1 non-concurrentes avec toute place distincte transitivement contenue dans u_2, et vice-versa en permutant u_1 et u_2.

À chaque itération, l'approche gloutonne sélectionne deux unités distinctes u_1 et u_2, toutes deux directement imbriquées dans la même unité u et telles que $\eta\,(u_1, u_2) \neq \varnothing$. Si aucun couple (u_1, u_2) n'existe, l'algorithme s'arrête. Si plusieurs couples (u_1, u_2) existent, l'algorithme sélectionne, comme premier critère, le couple ayant la plus petite hauteur (en notant bien que les deux unités u_1 et u_2 ont la même hauteur), et comme second critère, la paire maximisant la fonction $(u_1, u_2) \mapsto |\eta\,(u_1, u_2)|$. En conséquence, les paires d'unités sont sélectionnées de manière descendante, en commençant par les sous-unités de l'unité racine, puis en progressant vers les unités feuilles.

Une fois une paire (u_1, u_2) sélectionnée, possédant la même unité parente v, le NUPN est transformé comme suit. Une nouvelle unité u_3 est créée, qui est directement imbriquée dans v et contient directement u_1 et u_2. Toutes les places de $\eta\,(u_1, u_2)$ sont ôtées des unités u_1

et u_2 et deviennent des places locales de la nouvelle unité u_3 ; la condition $\eta(u_1, u_2) \neq \varnothing$ garantit que u_3 comporte au moins une place locale, empêchant ainsi l'ajout d'une unité vide non-racine dans le NUPN ainsi obtenu. La figure 8.2 illustre une itération de l'approche gloutonne, qui partant du NUPN à gauche, produit le NUPN à droite.

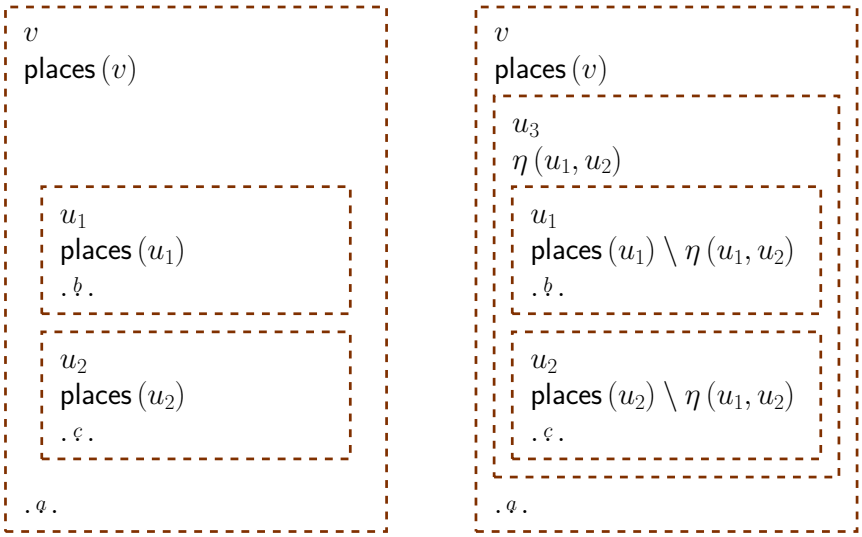

FIGURE 8.2 – Représentation graphique d'une itération de l'approche gloutonne

Si u_1 (respectivement u_2) ne comporte plus aucune place locale, elle est éliminée et par conséquent, toutes ses sous-unités, si elles existent, deviennent des sous-unités de u_3 ; si u_3 (respectivement v) ne comporte qu'une seule sous-unité u' (respectivement u_3) après ce changement, alors u' (respectivement u_3) est éliminée et toutes ses places locales et sous-unités sont transférées au sein de u_3 (respectivement v) ; ceci permet d'éviter l'introduction d'unités dites *redondantes*, car n'ayant pas d'unité sœur.

Combinées, ces deux propriétés rendent inutile la définition de η pour un nombre variable d'unités, dans la mesure où l'algorithme peut produire des unités ayant plus de deux sous-unités, moyennant plusieurs itérations.

La définition suivante formalise les transformations opérées par les itérations de l'approche gloutonne (en dehors de la première itération, qui correspond à la définition 8.5).

Définition 8.7 *Soit* $(P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$ *un NUPN dont la première itération (voir la définition 8.5) a déjà été appliquée. Une nouvelle itération de l'approche gloutonne produit un nouveau NUPN* $(P, T, F, M_0, U', u_0, \sqsubseteq', \mathsf{unité}')$*, tel que :*

— $\exists v \in U \mid \exists u_1, u_2 \in \mathsf{sous\text{-}unités}(v) \mid (u_1 \neq u_2) \wedge (\eta(u_1, u_2) \neq \varnothing) \wedge$
$(\forall v' \in U, \forall u'_1, u'_2 \in \mathsf{sous\text{-}unités}(v'),$
$((u'_1 \neq u'_2) \wedge (|\eta(u'_1, u'_2)| > |\eta(u_1, u_2)|)) \Rightarrow (\mathsf{hauteur}(u'_1) > \mathsf{hauteur}(u_1)))$;

— $U' \stackrel{\text{def}}{=} (\{\mathsf{unité}(p) \mid p \in P \setminus \eta(u_1, u_2)\} \cup \{u_0\}) \uplus \{u_3\}$;

— $\forall p \in \eta(u_1, u_2), \mathsf{unité}'(p) \stackrel{\text{def}}{=} u_3$;

— $\forall p \in P \setminus \eta\,(u_1, u_2), \mathsf{unité}'(p) \stackrel{\text{def}}{=} \mathsf{unité}\,(p)$;

— $\sqsubseteq' \stackrel{\text{def}}{=} \{(\mathsf{unité}'(p_1), \mathsf{unité}'(p_2)) \in (U' \times U') \mid \exists p_1, p_2 \in P \mid (\mathsf{unité}\,(p_1) \sqsubset \mathsf{unité}\,(p_2)) \vee (p_1 = p_2)\} \cup (\{u \in U' \mid (u \sqsubseteq u_1) \vee (u \sqsubseteq u_2)\} \times \{u_3\})$.

Proposition 8.9 *L'approche gloutonne se termine toujours.*

Preuve. L'approche gloutonne transforme un NUPN d'entrée en appliquant itérativement la définition 8.7, jusqu'à saturation. Il est possible d'appliquer cette définition *ssi* l'ensemble suivant est non-vide : $\{\{u_1, u_2\} \subseteq \mathsf{sous\text{-}unités}\,(u) \mid (u \in U) \wedge (u_1 \neq u_2) \wedge (\eta\,(u_1, u_2) \neq \varnothing)\}$. Or, cet ensemble est strictement décroissant, à chaque itération de l'algorithme. □

Proposition 8.10 *Si le NUPN d'entrée N est unit-sauf, alors le NUPN produit par la définition 8.7 est également unit-sauf.*

Preuve. Soit $p_1, p_2 \in P \mid p_1 \parallel p_2$. On a $(\mathsf{unité}\,(p_1) \notin \{u_1, u_2\}) \wedge (\mathsf{unité}\,(p_2) \notin \{u_1, u_2\})$, donc $\{p_1, p_2\} \subseteq P \setminus \eta\,(u_1, u_2)$. Or, N est unit-sauf et $p_1 \parallel p_2$. Donc $\mathsf{unité}\,(p_1) \not\sqsubseteq \mathsf{unité}\,(p_2)$ et conséquemment, $\mathsf{unité}'(p_1) \not\sqsubseteq' \mathsf{unité}'(p_2)$. Par symétrie, on a $\mathsf{disjoint}'(\mathsf{unité}'(p_1), \mathsf{unité}'(p_2))$.

□

8.7 Résultats expérimentaux

Les approches structurelle et gloutonne ont été implantées dans un outil prototype (600 lignes de code en langage Python), qui a été ensuite appliqué sur les 16 200 réseaux de Petri de la section 3.8, desquels nous avons exclu 80 réseaux de très grande taille (soit 0,5 % de la collection) car leurs matrices des places concurrentes ne pouvaient pas être générées.

Nous avons utilisé des serveurs de la grille de calcul Grid'5000, équipés de processeurs Intel Xeon Gold 5220 (2,2 GHz), de 96 Go de RAM et de la distribution Linux Debian 11. Nous avons alloué, pour la décomposition de chaque réseau, un délai maximal d'une minute.

Nous avons décidé d'appliquer systématiquement nos deux approches depuis un NUPN trivial (en écartant la structure NUPN, si elle était présente dans le réseau d'entrée). L'algorithme de l'approche gloutonne calcule des solutions incrémentales, permettant d'améliorer la décomposition courante jusqu'à la survenue d'un dépassement de délai ; notre implantation exploite une telle possibilité. En outre, nous avons choisi d'appliquer, à la fin de l'approche structurelle, l'algorithme glouton afin d'améliorer encore la décomposition produite par l'approche structurelle.

Voici les principales informations concernant les résultats expérimentaux :

— L'approche structurelle s'est exécutée avec succès sur 15 864 réseaux (soit 98,4 % de la collection). L'exécution a été interrompue pour 208 réseaux, suite à l'expiration du délai d'attente, sans que la moindre décomposition ne puisse être produite. L'exécution a été interrompue pour 48 réseaux, suite à l'expiration du délai d'attente, conduisant à une décomposition de qualité moindre que celle qui aurait été obtenue

sans délai d'attente. Le temps total passé dans ces décompositions a été de 6 heures, soit 1,3 seconde par réseau en moyenne.

— L'approche gloutonne s'est exécutée avec succès sur 15 757 réseaux (soit 97,7 % de la collection). L'exécution a été interrompue pour 62 réseaux, suite à l'expiration du délai d'attente, sans que la moindre décomposition ne puisse être produite. L'exécution a été interrompue pour 301 réseaux, suite à l'expiration du délai d'attente, conduisant à une décomposition de qualité moindre que celle qui aurait été obtenue sans délai d'attente. Le temps total passé dans ces décompositions a été de 8 heures 38 minutes, soit 1,9 seconde par réseau en moyenne.

Plus les modèles sont grands, plus la décomposition hiérarchique des unités est susceptible d'apporter des gains importants en terme de codage. En pratique, l'approche gloutonne est très rapide sur les petits modèles, mais elle est plus lente à converger sur les grands modèles, qui sont mieux traités et plus rapidement par l'approche structurelle : c'est pourquoi, de manière globale, l'approche structurelle est supérieure à l'approche gloutonne.

Notons que 76 % des modèles n'ont qu'un seul arbre couvrant possible, mais que, pour certains modèles, le nombre d'arbres couvrants est trop grand (par exemple, supérieur à 10^{500}) pour qu'une énumération exhaustive soit envisageable : de ce fait, il est nécessaire de définir des stratégies efficientes pour la sélection de l'arbre couvrant.

En ce qui concerne le codage de la définition 8.2, les NUPN hiérarchiques provenant des approches structurelle et gloutonne sont statistiquement plus compacts que les NUPN plats produits. Par exemple, en considérant le nombre total de bits requis pour l'ensemble des NUPN générés, l'approche gloutonne (respectivement structurelle) permet 3 % (respectivement 1 %) de bits en moins que l'approche de décomposition plate la plus performante du chapitre 7 (d'après la section 7.10), à savoir celle reposant sur la recherche itérée de clique maximum (voir la section 7.6). Cette réduction peut sembler mesurée, mais une réduction logarithmique du nombre de bits peut mener à une réduction exponentielle de la taille de représentation mémoire des marquages accessibles dans les diagrammes de décision binaire (voir la section 3.4).

8.8 Validation des résultats

La démarche de validation des résultats de ce présent chapitre est identique à celle présentée à la section 7.11 pour les décompositions plates. Brièvement, cette démarche nous permet de nous assurer que les décompositions produites préservent les réseaux de Petri sous-jacents et que les structures NUPN calculées sont unit-sauf.

Aussi, nos deux approches de décomposition hiérarchique seront réutilisées au chapitre 9 afin de traduire les réseaux de Petri en algèbres de processus. Ceci effectuera, indirectement, une validation des décompositions, lorsque le réseau d'entrée est comparé au code LNT généré (voir la section 9.6).

8.9 Bilan et perspectives

Alors que les décompositions plates du chapitre 7 produisaient une forme restreinte de NUPN (de hauteur un), ce présent chapitre lève cette restriction et utilise toutes les possibilités du formalisme.

Nous avons proposé deux approches complémentaires, permettant, soit de créer à partir de rien une structure NUPN, soit d'améliorer une structure existante. Dans les deux cas, le réseau de Petri sous-jacent reste inchangé, à l'isomorphisme près (voir la définition 4.3) : autrement dit, les places, transitions, arcs et le marquage initial du réseau d'entrée sont préservés.

Nos deux approches permettent de traiter environ 98 % d'une vaste collection comprenant 16 200 modèles (voir la section 3.8), d'origine diverse, avec un temps moyen d'environ deux secondes par modèle.

Par rapport aux décompositions plates, les décompositions hiérarchiques permettent une réduction de 1 % à 3 % du nombre de bits nécessaires pour représenter les marquages accessibles. Mais ces chiffres sont perfectibles, dans la mesure où le codage actuel des marquages accessibles pour les NUPN hiérarchiques est améliorable.

Les décompositions hiérarchiques pourraient être améliorées de différentes manières :

— Lorsque une décomposition est déjà produite, nous pouvons envisager des heuristiques permettant d'optimiser la taille du codage des marquages accessibles, exploitant ses propriétés non-linéaires (voir la section 3.4), et s'appuyant éventuellement sur la matrice des places concurrentes associée au réseau.

— L'approche gloutonne (voir la section 8.6) peut également s'appliquer à une décomposition pré-existante (par exemple, plate), avec pour objectif d'améliorer un NUPN, sans pour autant détruire sa structure existante.

— À ce propos, l'approche gloutonne pourrait être utilisée pour transformer les décompositions plates du MCC (voir la section 7.12) en décompositions hiérarchiques.

— Une autre manière de réduire la complexité du problème serait de pré-calculer localement certaines unités, au moyen de règles structurelles reposant sur les transitions ou d'invariants de places (en réutilisant, notamment [MCvdA13]). Cette opération produirait des décompositions intermédiaires, qui seraient ensuite données aux approches de ce chapitre, lesquelles produiraient (modulo quelques adaptations) une décomposition finale telle que deux places locales dans la décomposition intermédiaire soient dans une même unité au sein de la décomposition finale.

— À l'inverse, nous pourrions employer une stratégie de type « diviser pour régner », générant une couverture du réseau en sous-composants, pouvant éventuellement contenir plusieurs jetons (par exemple au moyen des localisations géographiques [Zai04]). Ces sous-composants peuvent être décomposés individuellement avec les approches de ce chapitre ; la décomposition finale peut alors être obtenue par l'union

des unités des sous-composants (modulo une déduplication des places éventuellement partagées entre les sous-composants).

Chapitre 9

Traduction des NUPN en algèbres de processus

Ce chapitre présente une approche visant à traduire les réseaux de Petri ordinaires et saufs en programmes concurrents écrits dans des algèbres de processus telles que LNT, CSPm, LOTOS ou mCRL2. L'approche proposée découvre automatiquement la concurrence présente dans les réseaux de Petri et l'exprime sous forme d'un ensemble de processus composés en parallèle et potentiellement imbriqués les uns dans les autres à une profondeur arbitraire, tout en préservant l'équivalence forte entre les marquages accessibles du réseau d'entrée et l'espace d'états du programme de sortie.

9.1 Énoncé du problème

En laissant de côté toutes les approches reposant sur la mémoire partagée, il existe trois principaux modèles de concurrence fondés sur le passage de messages : les réseaux d'automates communicants, les réseaux de Petri et les algèbres de processus. Les deux premiers sont généralement considérés comme des formalismes de bas niveau, tandis que le dernier est constitué de langages de plus haut niveau.

La traduction entre les deux premiers formalismes a été traitée au chapitre 7. Aussi, de nombreuses recherches ont été menées en vue de déterminer les liens entre les algèbres de processus et les réseaux de Petri, par exemple, [vV87] [BDH92, KEB94] [DDM88, DDM90] [Gar89, GS90] [Old86, Old91] [Tau89, Tau90] [LRR03] pour n'en mentionner que quelques unes. En ce qui concerne l'implémentation des langages concurrents modernes, cette thématique est toujours d'actualité ; voir par exemple [FH19]. Mais ces travaux considèrent surtout la traduction des algèbres de processus en réseaux de Petri, et non la traduction inverse, qui est sans doute plus difficile. Ce chapitre aborde ce problème inverse, à savoir la traduction des réseaux de Petri (ordinaires et saufs) en algèbres de processus.

9.2 Motivations

Il y a au moins cinq motivations qui sous-tendent une telle étude :

- Obtenir une meilleure compréhension de la relation théorique entre les réseaux de Petri et les algèbres de processus, en élaborant de nouvelles passerelles entre ces deux modèles de concurrence.
- Pouvoir réutiliser des techniques de vérification opérant sur des formalismes de plus haut niveau que les réseaux de Petri.
- Réaliser une ingénierie inverse transformant automatiquement des réseaux de Petri de grande taille, non-structurés et de bas niveau en programmes textuels concis, bien structurés et de haut niveau, lisibles et compréhensibles par des humains.
- Autoriser la conversion des collections existantes de réseaux de Petri en suites de tests utilisables pour vérifier les différents outils développés pour les calculs de processus.
- Implanter aisément de multiples extensions de réseaux de Petri (notamment les réseaux colorés, temporisés, avec des arcs inhibiteurs, avec des arcs de réinitialisation, etc.). Plutôt que de devoir gérer ces extensions diverses, il peut être opportun de les traduire systématiquement en utilisant des langages algébriques suffisamment souples pour toutes les représenter.

Pour un réseau de Petri comprenant m places et n transitions, il existe toujours une traduction directe de ce réseau en un programme non-déterministe équivalent, en implantant directement la sémantique opérationnelle des réseaux de Petri : le programme consiste simplement en, d'une part, m variables booléennes dont la valeur initiale est déterminée par le marquage initial, et d'autre part, n règles condition-action (une par transition) contrôlant si les places d'entrée d'une transition t sont marquées et, si tel est le cas, exécutant le franchissement de la transition en retirant le jeton de chaque place d'entrée de t et en déposant un jeton dans chaque place de sortie de t. Cependant, une telle traduction n'exploite pas les caractéristiques intrinsèques des calculs de processus. Le programme résultant est séquentiel et n'utilise que des choix non-déterministes : il n'est rien de plus qu'un programme à commandes gardées de Dijkstra ou un système de réécriture de termes.

Au lieu de cela, nous cherchons une traduction alternative préservant fidèlement la concurrence présente dans les réseaux de Petri, en exprimant cette concurrence à l'aide des opérateurs de composition parallèle disponibles dans les calculs de processus. Naturellement, si le réseau d'entrée est simplement une machine à états, sa traduction devrait consister en un simple automate ou une expression régulière. Dans des cas plus complexes, la traduction devrait faire intervenir des processus concurrents et, éventuellement, des processus imbriqués dans d'autres processus à une profondeur d'imbrication arbitraire.

La suite de ce chapitre s'articule comme suit. La section 9.3 esquisse un inventaire de l'état de l'art. En s'appuyant sur les résultats des chapitres précédents (en particulier ceux des chapitres 6 à 8), la section 9.4 propose un schéma élégant pour traduire les NUPN en programmes LNT fortement équivalents et exploitant la composition parallèle. La section 9.5

présente nos résultats expérimentaux sur une collection de 16 200 modèles. La section 9.6 aborde les démarches entreprises pour la validation des résultats. La section 9.7 discute des possibilités de transposition du schéma de traduction pour d'autres langages que LNT, notamment CSPm, LOTOS et mCRL2. Enfin, la section 9.8 formule des observations conclusives et donne des orientations pour les travaux ultérieurs.

9.3 Travaux voisins

À notre connaissance, il n'existe que peu de travaux antérieurs relatifs à ce sujet. Outre les articles précités à la section 9.1, qui expriment la sémantique des algèbres de processus en termes de réseaux de Petri, nous pouvons mentionner [BB93, BB01], qui donne une traduction (préservant la « step bisimulation ») de la notation des réseaux de Petri en une variante non-entrelacée, d'ordre partiel de l'algèbre de processus ACP dans laquelle la communication est remplacée par un opérateur d'état causal et [GMR+07, RPU+07], qui explique comment les *multiactions* ont été introduites dans mCRL2 pour permettre une traduction compositionnelle des réseaux de Petri colorés.

Notre approche est différente, dans le sens où nous prenons des algèbres de processus existantes (LNT, LOTOS, CSPm, etc.) telles qu'elles sont, plutôt que de les modifier pour traiter les réseaux de Petri.

Plus récemment, [ST18] propose une traduction des réseaux de Petri en ACP, qui produit des fragments de code épars (à savoir, une définition de processus ACP par place dans le réseau de Petri), alors que notre approche produit un code plus compact (avec une seule définition de processus par composant séquentiel dans le réseau de Petri).

9.4 Traduction vers l'algèbre de processus LNT

Il existe de multiples façons de traduire un NUPN unit-sauf (qu'il soit trivial, plat, ou hiérarchique) en un programme écrit dans une algèbre de processus. Parmi celles-ci, et à l'exclusion de la traduction purement séquentielle mentionnée dans la section 9.2, nous présentons ici une approche simple reposant sur les unités NUPN. En ce qui concerne l'algèbre de processus, nous adoptons le langage LNT [CCG+21, GLS17] pour sa lisibilité et son opérateur de composition parallèle n-aire (plutôt que binaire) [GS99].

Notre traduction doit préserver une équivalence forte entre, d'une part, le graphe des marquages accessibles du NUPN et d'autre part, le système de transitions étiquetées du programme LNT généré. Pour cela, le graphe des marquages accessibles doit être converti en un système de transition étiqueté en effaçant toutes les informations relatives aux noms de places et en ne gardant que les informations concernant les noms de transitions ; en effet, les états ne portent aucune information attachée dans les systèmes de transitions étiquetées, hormis l'état initial qui peut être distingué de tous les autres états.

Étant donné un NUPN $N = (P, T, F, M_0, U, u_0, \sqsubseteq, \mathsf{unité})$, notre traduction comporte deux

volets, décrits ci-après.

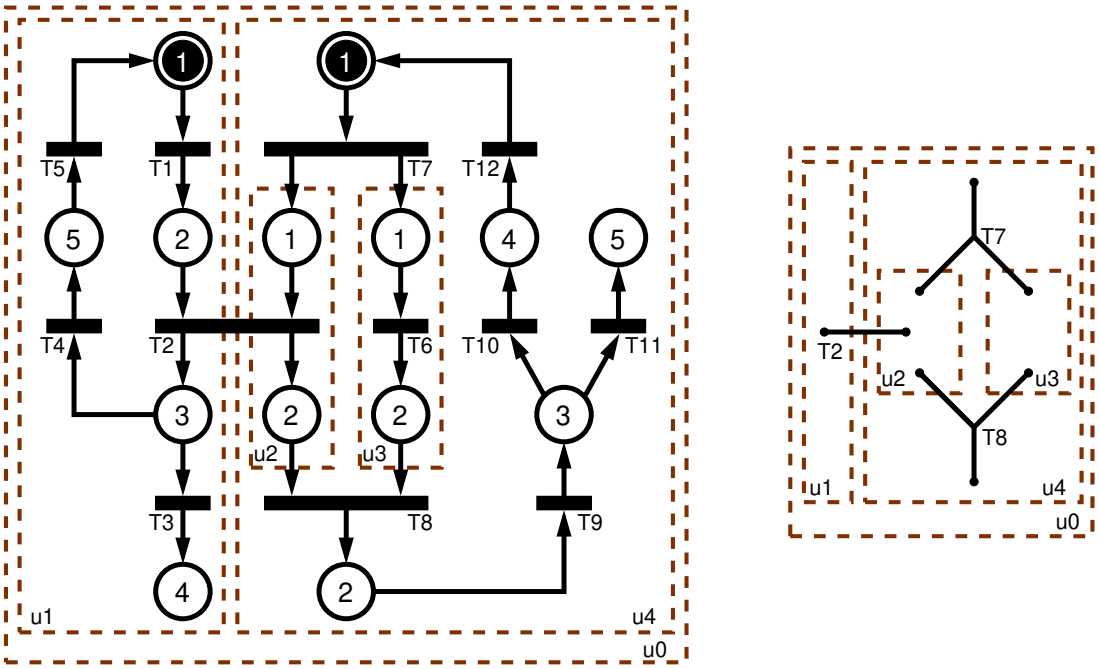

FIGURE 9.1 – Exemple de NUPN (gauche) et de son réseau de synchronisation (droite)

9.4.1 Traduction des unités vers les processus

Premièrement, chaque unité $u \in U$ ayant au moins une place locale (ceci a pour conséquence d'exclure l'unité racine u_0 lorsqu'elle est vide) est traduite en un processus LNT, noté u, qui est défini comme suit.

Définition 9.1 *La fonction* $\nu_u : P \mapsto \mathbb{N}$ *est définie de la manière suivante :*

$$\nu_u(p) \stackrel{\text{def}}{=} \begin{cases} |\{p' \in \mathsf{places}(u) \mid \#(p') \leq \#(p)\}| & \text{si } p \in \mathsf{places}(u) \\ 0 & \text{sinon} \end{cases}.$$

Les places locales de u sont traduites par des entiers naturels compris dans l'intervalle $\{1 : |\mathsf{places}(u)|\}$, comme illustré par la partie gauche de la figure 9.1. Puisque le NUPN est unit-sauf, il existe au plus un jeton dans les places locales de u. Conséquemment, le processus u possède une variable locale entière $P \in \{0 : |\mathsf{places}(u)|\}$, telle que $P = 0$ lorsque l'unité u n'a pas de jeton, et telle que $P = \nu_u(p)$ lorsque l'unité u a un jeton dans sa place locale p (dans ce cas, $P > 0$, voir la définition 9.1).

Si u a une place p initialement marquée (autrement dit, si $\exists p \in \mathsf{places}(u) \cap M_0$), P est initialisé à $\nu_u(p)$, sinon, P est initialisé à 0.

Le processus se comporte ensuite comme une boucle infinie, vérifiant de manière répétée la valeur de P, afin de pouvoir décider de franchir une transition et de mettre à jour la valeur de P (comme illustré par la partie gauche de la figure 9.2).

```
process u4 [T7, ..., T12: none] is
  var P: nat in
    P := 1;
    loop
      case P in
        0 → T8; P := 2
      | 1 → T7; P := 0
      | 2 → T9; P := 3
      | 3 →
        select
          T10; P := 4
        [] T11; P := 5
        end select
      | 4 → T12; P := 1
      | 5 → stop
      end case
    end loop
  end var
end process
```

```
...

process MAIN [T1, ..., T12: none] is
  par
    T2 → u1 [T1, ..., T5]
  || T2 →
    par
      T7, T8 → u4 [T7, ..., T12]
    || T7, T8 →
      par
        T7, T8 → u2 [T2, T7, T8]
      || T7, T8 → u3 [T6, T7, T8]
      end par
    end par
  end par
end process
```

FIGURE 9.2 – Code LNT généré pour l'unité u_4 (gauche) et pour l'unité principale (droite)

Nous introduisons, dans la définition ci-dessous, la fonction σ_u. Elle donne, pour chaque valeur courante de P, les transitions susceptibles d'être franchies.

Définition 9.2 *La fonction* $\sigma_u : \mathbb{N} \mapsto \{T\}$ *est définie ainsi :*

- $\Delta \stackrel{\text{def}}{=} \{t \in T \mid \exists u \in U \mid (|{}^\bullet t \cap \mathsf{places}(u)| > 1) \vee (|t^\bullet \cap \mathsf{places}(u)| > 1)\}$;
- $\sigma_u(0) \stackrel{\text{def}}{=} \{t \in T \setminus \Delta \mid ({}^\bullet t \cap \mathsf{places}(u) = \varnothing) \wedge (t^\bullet \cap \mathsf{places}(u) \neq \varnothing)\}$;
- $\forall p \in \mathsf{places}(u), (\sigma_u \circ \nu_u)(p) \stackrel{\text{def}}{=} \{t \in p^\bullet \setminus \Delta\}$.

Les transitions de Δ*, qui possèdent deux places d'entrée dans la même unité, ou deux places de sortie dans la même unité, sont mortes dans un NUPN unit-sauf (voir la proposition 5.7). Pour faciliter la traduction, elles sont omises de* σ_u.

En utilisant la définition 9.2, pour chaque valeur de $P \in \{0 : |\mathsf{places}(u)|\}$, nous générons une branche correspondant dans l'instruction « **case** P **in** ... **end case** » du processus u :

- Si $\sigma_u(P) = \varnothing$, le processus u s'interrompt alors avec une instruction « **stop** » (voir le cas $P = 5$ de la figure 9.2).
- Sinon, si $|\sigma(P)| = 1$, le processus u exécute alors la transition $t \in \sigma(P)$ et met à jour la valeur de P suivant $(\nu_u \circ \mathsf{oneof})(t^\bullet \cap \mathsf{places}(u))$ (voir le cas $P = 0$ de la figure 9.2). L'assignation finale de P est naturellement omise *ssi* ${}^\bullet t \cap \mathsf{places}(u) = t^\bullet \cap \mathsf{places}(u)$.

— Sinon, nous avons $|\sigma(P)| > 1$, le processus u offre alors un choix non-déterministe (noté « **select** ... **end select** ») parmi toutes les transitions de $\sigma(P)$ (voir le cas $P = 3$ de la figure 9.2).

Le cas $P = 0$ est omis si l'unité considérée possède initialement un jeton (P n'est pas initialisé à 0) et ne peut le perdre (l'instruction « **case** P **in** ... **end case** » ne contient aucune affectation « P := 0 »).

9.4.2 Composition parallèle et hiérarchique des processus

Le second volet de notre traduction consiste en la génération du processus LNT principal. Pour générer ce processus principal, nous construisons d'abord un *réseau de synchronisation* représentant toutes les unités du NUPN et toutes les transitions reliant les différentes unités, comme le montre la partie droite de la figure 9.1. Ce réseau de synchronisation permet ensuite de définir un prédicat $\varphi(t, V) \stackrel{\text{def}}{=} (({}^\bullet t \cup t^\bullet) \cap \bigcup_{u \in V}(u)) \neq \varnothing$, qui est vrai *ssi* la transition t est reliée à une place contenue dans une unité parmi l'ensemble des unités V.

Les processus u sont ensuite composés en parallèle au moyen de l'opérateur n-aire « **par** ... **end par** » du langage LNT, qui permet de synchroniser les processus deux par deux, trois par trois, et ainsi de suite, comme illustré par la partie droite de la figure 9.2. Dans le code LNT, à gauche de la flèche « → » et avant chaque processus u, le traducteur liste toutes les transitions sur lesquelles u doit se synchroniser avec d'autres processus. Il existe plusieurs façons d'exprimer la composition parallèle : notre traduction est effectuée récursivement, de manière ascendante, en composant les unités filles entre elles avant de les composer avec leurs unités parentes. Lors de la composition des unités filles de l'unité parente u, chaque unité fille u' doit se synchroniser sur les transitions t telles que $\varphi(t, \{v \in \mathsf{sous\text{-}unités}^*(u) \mid v \sqsubseteq u'\})$ et $\varphi(t, \{v \in \mathsf{sous\text{-}unités}^*(u) \mid v \not\sqsubseteq u'\})$. Ensuite, le bloc « **par** ... **end par** » résultant et l'unité parente u sont synchronisés ensemble sur les transitions t telles que $\varphi(t, \{u\})$ et telles que $\varphi(t, \mathsf{sous\text{-}unités}^*(u))$.

Les transitions $t \in T$ telles que ${}^\bullet t = \varnothing$ ne sont pas mortes (par contraposition de la proposition 5.5), ne figurent dans aucun des processus u, et peuvent exister dans un réseau sauf *ssi* $t^\bullet = \varnothing$ (voir la contraposée de la proposition 5.6). Le tir de telles transitions est possible dans tout marquage accessible, mais n'a pas d'influence sur le marquage courant (puisque ${}^\bullet t = t^\bullet$). Ces transitions sont d'objet d'un traitement particulier : elles sont ajoutées dans le processus principal au moyen d'une composition parallèle « **par** ... **end par** », sans effectuer de synchronisation.

9.5 Résultats expérimentaux

Nous avons implémenté cette traduction dans un outil prototype (430 lignes de code en langage Python), que nous avons appliqué aux NUPN générés par les approches structurelle et gloutonne du chapitre 8. En réutilisant les mêmes ordinateurs qu'à la section 8.7, la traduction est rapide et nécessite généralement moins d'une seconde par NUPN, bien que

quelques NUPN de grande taille prennent beaucoup plus de temps (ce problème serait résolu en réécrivant le traducteur en C plutôt qu'en Python). Actuellement, 99,6 % des NUPN peuvent être traduits en LNT en moins d'une minute. En outre, le code LNT généré est, en moyenne, deux fois plus petit que le fichier PNML d'entrée.

9.6 Validation des résultats

Nous avons validé la correction de bout en bout de notre traduction PNML vers LNT de la manière suivante : (i) premièrement, en générant, à l'aide de l'outil Sift [1] le système de transitions étiquetées correspondant à chaque modèle PNML d'entrée ; (ii) secondement, en générant, à l'aide de la boîte à outils CADP, le système de transitions étiquetées correspondant à chaque programme LNT généré ; (iii) finalement, en vérifiant, à l'aide de l'outil Bcg_Cmp [2], que les deux systèmes de transitions étiquetées sont fortement équivalents.

Sur nos 16 200 modèles, nous avons ainsi réussi à prouver l'équivalence forte pour 13 108 (respectivement 13 151) programmes LNT générés par l'approche gloutonne (respectivement structurelle), ce qui représente un taux de succès de plus de 80 %. Les marquages accessibles des modèles restants n'ont pu être exhaustivement explorés avec Sift, et donc comparés avec les espaces d'états de leurs traductions en LNT.

Cette étape de validation nous a permis de remarquer qu'initialement, nous ne prenions pas en compte les transitions t telles que ${}^\bullet t = t^\bullet = \varnothing$, car elles ne sont reliées à aucune unité. Cette omission a été corrigée en incorporant ces transitions dans le processus principal (voir la sous-section 9.4.2).

9.7 Extensions à d'autres langages

Nous avons également élaboré des schémas de traduction similaires pour d'autres langages que LNT, à savoir LOTOS [ISO89], CSPm [Ros10] et mCRL2 [GM14].

Les figures 9.3 à 9.5 illustrent la traduction du NUPN de la figure 9.1 pour ces trois algèbres de processus, sur le modèle de la figure 9.2 qui l'illustrait pour le langage LNT.

Notons que la traduction en CCS [Mil80] serait, dans le cas général, impossible, car cette algèbre ne permet que les synchronisations binaires, alors que les réseaux de Petri permettent les synchronisations atomiques entre plus de deux processus.

1. projects.laas.fr/tina/manuals/sift.html
2. cadp.inria.fr/man/bcg_cmp.html

```
channel T0, ..., T12

u4 (P) =
  (P == 0) & T8 → u4 (2)
  []
  (P == 1) & T7 → u4 (0)
  []
  (P == 2) & T9 → u4 (3)
  []
  (P == 3) &
    (T10 → u1 (4) []
     T11 → u1 (5))
  []
  (P == 4) & T12 → u4 (1)
  []
  (P == 5) & STOP
```

```
...

MAIN =
   u1 (1)
   |[{T2}]|
   (
      u4 (1)
      |[{T7, T8}]|
      (
            u2 (0)
            |[{T7, T8}]|
            u3 (0)
      )
   )
```

FIGURE 9.3 – Code CSPm pour l'unité u_4 (gauche) et pour l'unité principale (droite)

```
act

T0, ..., T12,
T2sync, T7sync, T8sync;

proc

u4 (P:Nat) =
   (P == 0) → T8sync . u4 (2) +
   (P == 1) → T7sync . u4 (0) +
   (P == 2) → T9 . u4 (3) +
   (P == 3) →
      (T10 . u4 (4) +
       T11 . u4 (5)) +
   (P == 4) → T12 . u4 (1) +
   (P == 5) → delta;

...
```

```
init

allow ({T0, ..., T12},
   comm ({T2sync|T2sync → T2},
      u1 (1)
      ||
      comm ({T7sync|T7sync|T7sync
               → T7,
             T8sync|T8sync|T8sync
               → T8},
         u4 (1)
         ||
         u2 (0)
         ||
         u3 (0)
)));
```

FIGURE 9.4 – Code mCRL2 pour l'unité u_4 (gauche) et pour l'unité principale (droite)

```
specification MAIN [T1, ..., T12] :
noexit

library
   BOOLEAN, NATURAL
endlib

behaviour
   MAIN [T1, ..., T12]
where

process MAIN [T1, ..., T12] :
noexit :=
   u1 [T1, T1, T2, T3, T5] (1)
   |[T2]|  (
      u4 [T7, ..., T12] (1)
      |[T7, T8]| (
         u2 [T2, T7, T8] (0)
         |[T7, T8]|
         u3 [T6, T7, T8] (0)
   ))
endproc
```

```
...

process u4 [T7, ..., T12] (P : NAT) :
noexit :=
   [P = 0] → T8; u4 [T7, ..., T12] (2)
[]
   [P = 1] → T7; u4 [T7, ..., T12] (0)
[]
   [P = 2] → T9; u4 [T7, ..., T12] (3)
[]
   [P = 3] →
      (T10; u4 [T7, ..., T12] (4) []
       T11; u4 [T7, ..., T12] (5))
[]
   [P = 4] → T12; u4 [T7, ..., T12] (1)
[]
   [P = 5] → stop
endproc

endspec
```

FIGURE 9.5 – Code LOTOS pour l'unité principale (gauche) et pour l'unité u_4 (droite)

9.8 Bilan et perspectives

Le présent chapitre présente une approche originale et automatisée pour traduire les réseaux de Petri saufs (ainsi que les NUPN unit-saufs) en algèbre de processus. Cette approche exploite pleinement le parallélisme des algèbres de processus et s'appuie sur les NUPN comme modèle sémantique intermédiaire pour la traduction.

Notre traduction a été implantée dans une chaîne d'outils complète fonctionnant en quatre étapes successives : (i) un réseau de Petri ordinaire et sauf au format normalisé PNML est converti au format NUPN par l'intermédiaire d'une traduction automatisée, mentionnée à la section 3.6 ; (ii) à partir de ce réseau d'entrée, une matrice (complète ou partielle) des places concurrentes est calculée en utilisant l'outil CÆSAR.BDD [3], qui repose sur les algorithmes du chapitre 6, éventuellement avec l'aide de l'outil Kong [4] lorsque le réseau d'entrée est très grand ; (iii) sur la base de la matrice des places concurrentes (éventuellement incomplète), le réseau d'entrée est décomposé en un NUPN hiérarchique unit-sauf en utilisant les algorithmes du chapitre 8 ; (iv) le NUPN hiérarchique est finalement traduit (voir la

3. cadp.inria.fr/man/caesar.bdd.html
4. github.com/nicolasAmat/Kong

section 9.4) en un programme LNT (ou CSPm, LOTOS, mCRL2, etc.), dont le système de transitions étiquetées est fortement équivalent au graphe des marquages accessibles du réseau d'entrée. Cette chaîne d'outils a été implantée en langage Python et validée avec succès sur 80 % des 16 200 modèles présentés dans la section 3.8.

Ce travail de recherche pourrait être poursuivi dans différentes directions.

Il serait possible d'améliorer la traduction en considérant chacune des unités, non pas comme étant un automate exécutant une instruction d'aiguillage englobée au sein d'une boucle infinie (c.-à-d. « P := n; **loop**; **case** P **in** ... **end case**; **end loop** » dans le langage LNT), mais comme des processus ré-entrants comprenant des points d'entrée multiples. Chaque point d'entrée pourrait être traduit sous la forme d'une expression régulière, en reconnaissant des motifs usuels tels que des séquences, des cycles, des bifurcations (en anglais, fork), des jonctions (en anglais, join), et ainsi de suite. Dans certains cas, les instructions de boucle infinie ou d'aiguillage (en anglais, switch) pourraient être même éliminées du processus généré. D'autre part, les programmes générés pourraient être optimisés en détectant les (sous-) processus identiques ou très similaires, puis en les factorisant par des appels à un code LNT (paramétré) commun. Ces améliorations permettraient de produire des programmes plus concis et plus faciles à comprendre et à maintenir pour des humains.

Une autre voie d'amélioration serait de généraliser notre approche aux réseaux de Petri non-saufs. Mais la plupart des compilateurs existants pour nos langages d'algèbres de processus cibles ne gèrent pas la récursion de processus par composition parallèle, qui est pourtant le moyen naturel pour instancier un ensemble non limité de processus concurrents. Notre approche pourrait également être généralisée aux réseaux de Petri colorés, car tous nos langages cibles fournissent les fonctionnalités requises pour définir et manipuler des structures de données.

Conclusion

Chapitre 10

Conclusion

Bilan

Les réseaux de Petri possèdent une sémantique d'exécution très riche, leur permettant de décrire fidèlement les principes de la concurrence asynchrone. Ce formalisme repose sur un petit nombre de définitions mathématiques élégantes et précises. Ces réseaux ont, de plus, une représentation graphique simple et intuitive. Ces atouts font que les réseaux de Petri ont acquis, depuis leur introduction en 1962, une place incontournable pour la modélisation et la vérification de systèmes asynchrones, et qu'ils constituent toujours un champ de recherche actif.

Mais ces réseaux manquent de structure et ne permettent pas de représenter la notion de processus. Pour palier cette lacune, les NUPN ont été introduits en 2015. Ils complètent les réseaux de Petri, mais sans les remplacer : l'intégralité des propriétés topologiques et comportementales des réseaux de Petri ordinaires et saufs est préservée par les NUPN. Les NUPN étendent les réseaux de Petri en leur apportant deux notions essentielles : la *modularité* (exprimant la notion de processus) et la *hiérarchie* (permettant l'organisation de ces processus sous la forme d'arbres). À l'instar des réseaux de Petri, les NUPN reposent sur un nombre restreint de définitions mathématiques et ont une représentation graphique intuitive. Pour toutes ces raisons, les NUPN sont bien adaptés pour représenter les systèmes concurrents hiérarchiques et leur popularité s'accroît rapidement dans la communauté de la vérification de modèles, là où les réseaux de Petri ordinaires et saufs étaient utilisés.

Cette thèse s'inscrit dans ce mouvement, en apportant des contributions au formalisme des NUPN. Chacun de ses trois volets a permis de mieux comprendre et traiter les modèles NUPN (et les réseaux de Petri).

Concernant l'**équivalence** des NUPN, nous avons défini le principe de NUPN isomorphes, puis apporté des solutions viables pour identifier de tels modèles ou déterminer les modèles uniques, dans de vastes collections de jeux de tests (la plus importante comportant plus de 241 000 réseaux, dont de nombreux doublons). Nous avons pu détecter des réseaux

isomorphes dans les réseaux de Petri et les NUPN du *Model Checking Contest*[1], une compétition internationale dédiée à l'évaluation des outils de vérification formelle.

Concernant l'**analyse** des réseaux de Petri et des NUPN, nous avons permis le calcul de trois propriétés d'atteignabilité importantes en pratique, car elles permettent (entre autres) de détecter et d'éliminer le code mort de ces réseaux, de déterminer leur concurrence à une échelle locale (celle des places) et de résoudre d'autres problèmes connexes, comme la décision des propriétés sauf et unit-sauf (qui servent notamment pour coder les marquages accessibles des réseaux de Petri et des NUPN). Cette partie constitue un ingrédient indispensable à la partie suivante.

Concernant la **structuration** des réseaux de Petri et des NUPN, nous avons permis d'inférer la structure modulaire et hiérarchique manquante des réseaux de Petri, et ainsi de les traduire en réseaux d'automates communicants, en NUPN, ou en algèbre de processus. Cette approche permet également d'établir un pont entre les communautés des réseaux de Petri, des automates communicants et des algèbres de processus. Nous avons utilisé avec succès cette approche pour structurer les modèles ordinaires et saufs du Model Checking Contest.

Pour ces problèmes qui sont d'une part, difficiles à appréhender par l'esprit humain, et d'autre part, ont de fortes complexités théoriques (notamment NP-complets, ou PSPACE-complets), nous avons proposé des approches pragmatiques (une solution partielle est préférable à l'absence de solution) et complémentaires (en combinant divers algorithmes très différents dans leurs principes).

Afin de valider les performances et la correction de nos propositions, nous avons accordé un soin particulier au volet expérimental, en implantant nos solutions et en les appliquant à une collection de modèles d'ampleur inégalée dans la communauté des réseaux de Petri.

Ces travaux ont débouché sur des publications dans la conférence de référence pour les réseaux de Petri [BGdL20] [BG21a], des applications concrètes pour la préparation du Model Checking Contest, et des jeux de tests utilisés dans des compétitions internationales (Model Checking Contest, *Model Counting*[2], *SAT Competition*[3], *SMT-Comp*[4]).

Perspectives

Il serait possible d'étendre ce travail dans plusieurs directions.

Dans notre travail, la partie contrôle des systèmes concurrents hiérarchiques est modélisée de manière *statique* et non *dynamique*. Il serait possible de généraliser la partie contrôle, pour autoriser la description de certains systèmes pouvant démarrer et arrêter des nombres variables de processus. Mais ceci n'est pas limitant en pratique, car en général, il est possible

1. mcc.lip6.fr
2. mccompetition.org
3. satcompetition.github.io
4. smt-comp.github.io

de borner le nombre maximal de processus pouvant être démarrés en parallèle.

À présent, nous ne traitons que la partie contrôle des systèmes concurrents hiérarchiques. L'étape suivante consisterait à étendre le modèle NUPN pour considérer les structures de données, par exemple, en reprenant certaines idées des réseaux de Petri colorés, des réseaux de Petri interprétés et des réseaux de Petri temporisés.

La partie dédiée à l'équivalence permet, en reposant uniquement sur les informations structurelles des réseaux en entrée, de déterminer leur isomorphisme, ce qui implique l'isomorphisme de leurs espaces d'états. Combinée avec les parties suivantes, visant l'analyse et la structuration des modèles, nous pourrions définir des moyens pour détecter l'équivalence de programmes écrits en algèbres de processus.

Les approches de la partie dédiée à l'équivalence pourraient également être appliquées plus finement, non plus globalement (à l'échelle des réseaux), mais localement (à l'échelle des unités), ce qui aiderait par exemple, à déterminer quels sont les processus équivalents lors de la comparaison de flux de travaux, ou lors du re-déploiement d'applications distribuées.

Nous pourrions définir des fonctions de codage des marquages accessibles plus compactes (voir la section 3.4), en exploitant davantage la hiérarchie apportée par la structure NUPN.

La partie visant la structuration des réseaux de Petri et des NUPN a permis de connecter les réseaux de Petri aux réseaux d'automates communicants, aux NUPN et aux algèbres de processus, en inférant leurs processus. Ceci permettrait d'exécuter les réseaux de Petri de manière efficace, et de les utiliser comme outil de prototypage rapide.

Bibliographie

[ABB+16] Elvio Gilberto Amparore, Gianfranco Balbo, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. 30 Years of GreatSPN. In Lance Fiondella and Antonio Puliafito, editors, *Principles of Performance and Reliability Modeling and Evaluation*, pages 227–254. Springer, 2016.

[ABC+19] Elvio Gilberto Amparore, Bernard Berthomieu, Gianfranco Ciardo, Silvano Dal-Zilio, Francesco Gallà, Lom Messan Hillah, Francis Hulin-Hubard, Peter Gjøl Jensen, Loïg Jezequel, Fabrice Kordon, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Andrew S. Miner, Emmanuel Paviot-Adet, Jirí Srba, Yann Thierry-Mieg, Tom van Dijk, and Karsten Wolf. Presentation of the 9th Edition of the Model Checking Contest. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Proceedings (Part III) of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19 : TOOLympics), Prague, Czech Republic*, volume 11429 of *Lecture Notes in Computer Science*, pages 50–68. Springer, April 2019.

[ABD17] Elvio Gilberto Amparore, Marco Beccuti, and Susanna Donatelli. Gradient-Based Variable Ordering of Decision Diagrams for Systems with Structural Units. In Deepak D'Souza and K. Narayan Kumar, editors, *Proceedings of the 15th International Symposium on Automated Technology for Verification and Analysis (ATVA'17), Pune, India*, volume 10482 of *Lecture Notes in Computer Science*, pages 184–200. Springer, October 2017.

[ABD21] Nicolas Amat, Bernard Berthomieu, and Silvano Dal-Zilio. On the Combination of Polyhedral Abstraction and SMT-Based Model Checking for Petri Nets. In Didier Buchs and Josep Carmona, editors, *Proceedings of the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'21), Paris, France*, volume 12734 of *Lecture Notes in Computer Science*, pages 164–185. Springer, June 2021.

[ABD22] Nicolas Amat, Bernard Berthomieu, and Silvano Dal-Zilio. A Polyhedral Abstraction for Petri Nets and its Application to SMT-Based Model Checking. *Fundamenta Informaticae*, 187(2-4) :103–138, 2022.

[ABG23] Nicolas Amat, Pierre Bouvier, and Hubert Garavel. A Toolchain to Compute

Concurrent Places of Petri Nets. *Transactions on Petri Nets and Other Models of Concurrency*, 2023.

[ABH+01] Rajeev Alur, Robert Brayton, Thomas Henzinger, Shaz Qadeer, and Sriram Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration. *Formal Methods in System Design*, 18(2) :97–116, 2001.

[AC22] Nicolas Amat and Louis Chauvet. Kong : A Tool to Squash Concurrent Places. In Luca Bernardinello and Laure Petrucci, editors, *Proceedings of the 43nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'22), Bergen, Norway*, volume 13288 of *Lecture Notes in Computer Science*, pages 115–126. Springer, June 2022.

[AD22] Elvio G. Amparore and Susanna Donatelli. The Ins and Outs of Petri Net Composition. In Luca Bernardinello and Laure Petrucci, editors, *Proceedings of the 43nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'22), Bergen, Norway*, volume 13288 of *Lecture Notes in Computer Science*, pages 278–299. Springer, June 2022.

[AKW05] Marian Adamski, Andrei Karatkevich, and Marek Wegrzyn. *Design of Embedded Control Systems*. Springer, June 2005.

[Arn82] André Arnold. Synchronized Behaviours of Processes and Rational Relations. *Acta Informatica*, 17 :21–29, 1982.

[AW06] Mariam Adamski and Monika Wiśniewska. Dekompozycja równoległa automatów współbieznych z wykorzystaniem hipergrafów. *Pomiary, Automatyka, Kontrola*, 52(nr 6, wyd. spec.) :8–10, 2006.

[Bab16] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA*, pages 684–697. ACM, June 2016.

[Bab19] László Babai. Canonical form for graphs in quasipolynomial time : preliminary report. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA*, pages 1237–1246. ACM, June 2019.

[BB93] Jos C. M. Baeten and Jan A. Bergstra. Non Interleaving Process Algebra. In Eike Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93), Hildesheim, Germany*, volume 715 of *Lecture Notes in Computer Science*, pages 308–323. Springer, August 1993.

[BB01] Jos C. M. Baeten and Twan Basten. Partial-Order Process Algebra (and its Relation to Petri Nets). In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 769–872. North-Holland / Elsevier, 2001.

[BC79] Kellogg S. Booth and Charles J. Colbourn. *Problems Polynomially Equiva-*

lent to Graph Isomorphism. Computer Science Department, University of Waterloo, Canada, 1979.

[BCD02] Eric Badouel, Benoît Caillaud, and Philippe Darondeau. Distributing Finite Automata Through Petri Net Synthesis. *Formal Aspects of Computing*, 13(6) :447–470, 2002.

[BCD+11] Clark W. Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11) Snowbird, UT, USA*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011.

[BCH10] Sandie Balaguer, Thomas Chatain, and Stefan Haar. A Concurrency-Preserving Translation from Time Petri Nets to Networks of Timed Automata. In Nicolas Markey and Jef Wijsen, editors, *Proceedings of the 17th International Symposium on Temporal Representation and Reasoning (TIME'10), Paris, France*, pages 77–84. IEEE Computer Society, September 2010.

[BCH12] Sandie Balaguer, Thomas Chatain, and Stefan Haar. A Concurrency-Preserving Translation from Time Petri Nets to Networks of Timed Automata. *Formal Methods in System Design*, 40(3) :330–355, 2012.

[BD96] Eric Badouel and Philippe Darondeau. Theory of Regions. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I : Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 529–586. Springer, September 1996.

[BD98] Krzysztof Bilinski and Erik L. Dagless. Efficient Approach to Symbolic State Exploration of Complex Parallel Controllers. In *1st International Conference on Application of Concurrency to System Design (ACSD'98), Fukushima, Japan*, pages 132–142. IEEE Computer Society, March 1998.

[BD11] Eike Best and Philippe Darondeau. Petri Net Distributability. In Edmund M. Clarke, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Revised Selected Papers of the 8th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI'11), Novosibirsk, Russia*, volume 7162 of *Lecture Notes in Computer Science*, pages 1–18. Springer, June–July 2011.

[BD15] Eike Best and Raymond R. Devillers. Synthesis and reengineering of persistent systems. *Acta Informatica*, 52 :35–60, February 2015.

[BdC92] Luca Bernardinello and Fiorella de Cindio. A Survey of Basic Net Models and Modular Net Classes. In Grzegorz Rozenberg, editor, *Advances in Petri Nets – The DEMON Project*, volume 609 of *Lecture Notes in Computer*

Science, pages 304–351. Springer, 1992.

[BDE93] Eike Best, Raymond R. Devillers, and Javier Esparza. General Refinement and Recursion Operators for the Petri Box Calculus. In Patrice Enjalbert, Alain Finkel, and Klaus W. Wagner, editors, *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science (STACS'93), Würzburg, Germany*, volume 665 of *Lecture Notes in Computer Science*, pages 130–140. Springer, February 1993.

[BDH92] Eike Best, Raymond R. Devillers, and Jon G. Hall. The Box Calculus : A New Causal Algebra with Multi-Label Communication. In Grzegorz Rozenberg, editor, *Advances in Petri Nets – The DEMON Project*, volume 609 of *Lecture Notes in Computer Science*, pages 21–69. Springer, 1992.

[BDK99] Eike Best, Raymond R. Devillers, and Maciej Koutny. The Box Algebra – A Model of Nets and Process Expressions. In Susanna Donatelli and H. C. M. Kleijn, editors, *Proceedings of the 20th International Conference on Application and Theory of Petri Nets (ICATPN'99), Williamsburg, VA, USA*, volume 1639 of *Lecture Notes in Computer Science*, pages 344–363. Springer, 1999.

[BDK01a] Eike Best, Raymond R. Devillers, and Maciej Koutny. A Unified Model for Nets and Process Algebras. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 14, pages 873–944. Elsevier, 2001.

[BDK01b] Eike Best, Raymond R. Devillers, and Maciej Koutny. *Petri Net Algebra*. EATCS Monographs in Theoretical Computer Science. Springer, 2001.

[BDK02] Eike Best, Raymond R. Devillers, and Maciej Koutny. The Box Algebra = Petri Nets + Process Expressions. *Information and Computation*, 178(1) :44–100, 2002.

[BDL+11] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 Years. *Software Practice and Experience*, 41(2) :133–142, 2011.

[Bes79] Eike Best. The Relative Strength of K-Density. In Wilfried Brauer, editor, *Net Theory and Applications, Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, Germany*, volume 84 of *Lecture Notes in Computer Science*, pages 261–276. Springer, October 1979.

[BG92] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[BG20] Pierre Bouvier and Hubert Garavel. The VLSAT-1 Benchmark Suite. Technical Report RT-0510, INRIA, Grenoble, France, November 2020. Avai-

lable from https://hal.inria.fr/hal-03007233 and https://arxiv.org/abs/2011.11049.

[BG21a] Pierre Bouvier and Hubert Garavel. Efficient Algorithms for Three Reachability Problems in Safe Petri Nets. In Didier Buchs and Josep Carmona, editors, *Proceedings of the 42nd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'21), Paris, France*, volume 12734 of *Lecture Notes in Computer Science*, pages 339–359. Springer, June 2021.

[BG21b] Pierre Bouvier and Hubert Garavel. SAT-Competition Benchmarks Spawning from Concurrency Theory. In Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2021 – Solver and Benchmark Descriptions*, pages 47–48. Report B-2021-1, University of Helsinki, Department of Computer Science, July 2021.

[BG21c] Pierre Bouvier and Hubert Garavel. The VLSAT-2 Benchmark Suite. Technical Report RT-0514, INRIA, Grenoble, France, September 2021. Available from https://hal.inria.fr/hal-03337115 and https://arxiv.org/abs/2110.06336.

[BGdL20] Pierre Bouvier, Hubert Garavel, and Hernán Ponce de León. Automatic Decomposition of Petri Nets into Automata Networks – A Synthetic Account. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Proceedings of the 41st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'20), Paris, France*, volume 12152 of *Lecture Notes in Computer Science*, pages 3–23. Springer, June 2020.

[BH14] Dines Bjørner and Klaus Havelund. 40 Years of Formal Methods - Some Obstacles and Some Possibilities ? In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014 : Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2014.

[Bie19] Armin Biere. CaDiCaL at the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, November 2019.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37 :77–121, 1985.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker : A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Veri-*

fication (CAV'11), Snowbird, UT, USA, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, July 2011.

[BLJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. (synchronous programming with events and relations : the signal language and its semantics). *Science of Computer Programming*, 16(2) :103–149, 1991.

[BMS17] Nicolas Basset, Jean Mairesse, and Michèle Soria. Uniform Sampling for Networks of Automata. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR'17), Berlin, Germany*, volume 85 of *LIPIcs*, pages 36 :1–36 :16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, September 2017.

[Bou21] Pierre Bouvier. The VLSAT-3 Benchmark Suite. Technical Report RT-0516, INRIA, Grenoble, France, December 2021. Available from https://hal.inria.fr/hal-3468625 and https://arxiv.org/abs/2112.03675.

[BRV04] Bernard Berthomieu, Pierre-Olivier Ribet, and François Vernadat. The Tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14) :2741–2756, 2004.

[Cap11] Lorenzo Capra. A lumped Markov process for a class of dynamic Petri nets. In Saad Biaz and Shaoen Wu, editors, *2011 Spring Simulation Multiconference, SpringSim '11, Boston, MA, USA Volume 2 : Proceedings of the 44th Annual Simulation Symposium (ANSS)*, pages 188–197. ACM, April 2011.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Shmuel Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP'05), Edinburgh, UK*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, April 2005.

[CCG+21] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe, and Gideon Smeding. Reference Manual of the LNT to LOTOS Translator (Version 7.0). INRIA, Grenoble, France, September 2021.

[CCI88] CCITT. Specification and Description Language. Recommendation Z.100, International Consultative Committee for Telephony and Telegraphy, Geneva, March 1988.

[CEP95] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity Results for 1-Safe Nets. *Theoretical Computer Science*, 147(1–2) :117–136, 1995.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3) :95–120, 1988.

[CHVB18] Edmund Clarke, Thomas Henzinger, Helmut Veith, and Roderick Bloem,

editors. *Handbook of Model Checking*. Springer, 2018.

[CMPW09] Gianfranco Ciardo, Galen Mecham, Emmanuel Paviot-Adet, and Min Wan. P-Semiflow Computation with Decision Diagrams. In Giuliana Franceschinis and Karsten Wolf, editors, *Proceedings of the 30th International Conference on Application and Theory of Petri Nets (ICATPN'09), Paris, France*, volume 5606 of *Lecture Notes in Computer Science*, pages 143–162. Springer, June 2009.

[CSV86] José Colom, Manuel Silva, and José Villarroel. On software implementation of Petri Nets and colored Petri Nets using high-level concurrent languages. In Grzegorz Rozenberg, editor, *Proceedings of the 7th European Workshop on Applications and Theory of Petri Nets (APN'87), Oxford, UK*, volume 266 of *Lecture Notes in Computer Science*, pages 207–222. Springer, June 1986.

[DDM88] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A Distributed Operational Semantics for CCS Based on Condition/Event Systems. *Acta Informatica*, 26(1/2) :59–91, 1988.

[DDM90] Pierpaolo Degano, Rocco De Nicola, and Ugo Montanari. A Partial Ordering Semantics for CCS. *Theoretical Computer Science*, 75(3) :223–262, 1990.

[DE95] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[Dev21] Raymond R. Devillers. Articulations and Products of Transition Systems and their Applications to Petri Net Synthesis. *CoRR*, abs/2111.00202, 2021.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9) :569–570, September 1965.

[Dij68] Edsger W. Dijkstra. The structure of "the"-multiprogramming system. *Communcations of the ACM*, 11(5) :341–346, 1968.

[DKKP02] Raymond R. Devillers, Hanna Klaudel, Maciej Koutny, and Franck Pommereau. An Algebra of Non-safe Petri Boxes. In Hélène Kirchner and Christophe Ringeissen, editors, *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology (AMAST'02), Saint-Gilles-les-Bains, Reunion Island, France*, volume 2422 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2002.

[DKKP03] Raymond R. Devillers, Hanna Klaudel, Maciej Koutny, and Franck Pommereau. Asynchronous Box Calculus. *Fundamenta Informaticae*, 54(4) :295–344, 2003.

[DKR+13] Matjaz Depolli, Janez Konc, Kati Rozman, Roman Trobec, and Dusanka Janezic. Exact Parallel Maximum Clique Algorithm for General and Protein Graphs. *Journal of Chemical Information and Modeling*, 53(9) :2217–2228,

January 2013.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), Budapest, Hungary*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.

[DN89] Srinivas Devadas and Richard Newton. Decomposition and Factorization of Sequential Finite State Machines. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 8 :1206–1217, November 1989.

[DR96] Jörg Desel and Wolfgang Reisig. The Synthesis Problem of Petri Nets. *Acta Informatica*, 33 :297–315, June 1996.

[DS18] Raymond R. Devillers and Uli Schlachter. Factorisation of Petri Net Solvable Transition Systems. In Victor Khomenko and Olivier H. Roux, editors, *Proceedings of the 39st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'18), Bratislava, Slovakia*, volume 10877 of *Lecture Notes in Computer Science*, pages 82–98. Springer, June 2018.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26rd International Conference on Computer Aided Verification (CAV'14), Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[Dyl13] Robert Dylewski. Assigning the cover of Petri nets with subnets of the automatic type. *Przeglad Elektrotechniczny*, 89 :285–289, January 2013.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1) :127–144, 2003.

[EM15] Javier Esparza and Philipp J. Meyer. An SMT-based Approach to Fair Termination Analysis. In Roope Kaivola and Thomas Wahl, editors, *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design (FMCAD'15), Austin, Texas, USA*, pages 49–56. IEEE, September 2015.

[ENN06] Attila Egri-Nagy and Chrystopher Nehaniv. Hierarchical decomposition–coordinate systems for understanding complexity. In *Proceedings of the Workshop on the Evolution of Complexity at Artificial Life X*, 2006.

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Mar-*

gherita Ligure, Italy Selected Revised Papers, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, May 2003.

[Esh09] Rik Eshuis. Translating Safe Petri Nets to Statecharts in a Structure-Preserving Way. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of the 2nd World Congress on Formal Methods (FM'09), Eindhoven, The Netherlands*, volume 5850 of *Lecture Notes in Computer Science*, pages 239–255. Springer, November 2009.

[Esp90] Javier Esparza. Synthesis Rules for Petri Nets, and How they Lead to New Results. In Jos C. M. Baeten and Jan Willem Klop, editors, *Proceedings of CONCUR'90, Theories of Concurrency : Unification and Extension, Amsterdam, The Netherlands*, volume 458 of *Lecture Notes in Computer Science*, pages 182–198. Springer, August 1990.

[FH19] Felix Freiberger and Holger Hermanns. Concurrent Programming from pseuCo to Petri. In Susanna Donatelli and Stefan Haar, editors, *Proceedings of the 40th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'19), Aachen, Germany*, volume 11522 of *Lecture Notes in Computer Science*, pages 279–297. Springer, June 2019.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. PhD thesis, Université Joseph Fourier (Grenoble), November 1989.

[Gar15] Hubert Garavel. Nested-Unit Petri Nets : A Structural Means to Increase Efficiency and Scalability of Verification on Elementary Nets. In Raymond R. Devillers and Antti Valmari, editors, *Proceedings of the 36th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'15), Brussels, Belgium*, volume 9115 of *Lecture Notes in Computer Science*, pages 179–199. Springer, June 2015.

[Gar19] Hubert Garavel. Nested-Unit Petri Nets. *Journal of Logical and Algebraic Methods in Programming*, 104 :60–85, April 2019.

[Gar20] Hubert Garavel. Proposal for Adding Useful Features to Petri-Net Model Checkers. Technical Report abs/2101.05024, arXiv Computing Research Repository, December 2020.

[GG13] Hubert Garavel and Susanne Graf. Formal Methods for Safe and Secure Computers Systems. BSI Study 875, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany, December 2013.

[GKM07] Blaise Genest, Dietrich Kuske, and Anca Muscholl. On Communicating Automata with Bounded Channels. *Fundamenta Informaticae*, 80(1–3) :147–167, 2007.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011 : A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer*

(STTT), 15(2) :89–107, April 2013.

[GLS17] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. From LOTOS to LNT. In Joost-Pieter Katoen, Rom Langerak, and Arend Rensink, editors, *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, volume 10500 of *Lecture Notes in Computer Science*, pages 3–26. Springer, October 2017.

[GM14] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[GMR+07] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck van Weerdenburg. The Formal Specification Language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl, Germany, 2007.

[Gor17] Roberto Gorrieri. *Process Algebras for Petri Nets - The Alphabetization of Distributed Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2017.

[Gre09] Tener Greg. *Attacks On Difficult Instances Of Graph Isomorphism : Sequential And Parallel Algorithms*. PhD thesis, University of Central Florida, 2009.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV'90), Ottawa, Canada*, pages 379–394. North-Holland, June 1990.

[GS99] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'99), Beijing, China*, pages 185–202. Kluwer Academic Publishers, October 1999.

[GS06] Hubert Garavel and Wendelin Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Computer Science*, 351(2) :131–145, February 2006.

[GtBvdP20] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 Expert Survey on Formal Methods. In Maurice H. ter Beek and Dejan Nickovic, editors, *Proceedings of the 25th International Conference Formal Methods for Industrial Critical Systems (FMICS'20), Vienna, Austria*, volume 12327

of *Lecture Notes in Computer Science*, pages 3–69. Springer, September 2020.

[GvHPvdW06] R. Goud, Kees M. van Hee, R. D. J. Post, and Jan Martijn E. M. van der Werf. Petriweb : A repository for petri nets. In Susanna Donatelli and P. S. Thiagarajan, editors, *Proceedings of the 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN'06), Turku, Finland*, volume 4024 of *Lecture Notes in Computer Science*, pages 411–420. Springer, June 2006.

[GWW+14] Iwona Grobelna, Monika Wisniewska, Remigiusz Wisniewski, Michal Grobelny, and Piotr Mroz. Decomposition, validation and documentation of control process specification in form of a Petri net. In *7th International Conference on Human System Interactions, HSI 2014, Costa da Caparica, Portugal, June 16-18, 2014*, pages 232–237, 2014.

[Hac72] Michel H. T. Hack. Analysis of Production Schemata by Petri Nets. Master thesis (computer science), Massachusetts Institute of Technology, Cambridge, MA, USA, February 1972. Report MAC-TR 94 of the MIT Project MAC. See also : Michel Hack, "Corrections to MAC-TR 94", June 1974.

[Hac74] Michel H. T. Hack. Extended State-Machine Allocatable Nets (ESMA), an Extension of Free Choice Petri Net Results. MIT Project MAC, Computation Structures Group, Memo 78–1, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[Ham16] Alexandre Hamez. A Symbolic Model Checker for Petri Nets : pnmc. *Transactions on Petri Nets and Other Models of Concurrency*, 11 :297–306, 2016.

[Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4) :238–241, 1970.

[Han02] Per Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer, May 2002.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proc. IEEE*, 79(9) :1305–1320, 1991.

[HJM+21] Falk Howar, Marc Jasper, Malte Mues, David Schmidt, and Bernhard Steffen. The RERS challenge : towards controllable and scalable benchmark synthesis. *International Journal on Software Tools for Technology Transfer*, 23(6) :917–930, 2021.

[HK98] Martin Hesketh and Maciej Koutny. An Axiomatisation of Duplication Equivalence in the Petri Box Calculus. In Jörg Desel and Manuel Silva Suárez, editors, *Proceedings of the 19th International Conference on Application and Theory of Petri Nets (ICATPN'98), Lisbon, Portugal*, volume

1420 of *Lecture Notes in Computer Science*, pages 165–184. Springer, June 1998.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8) :666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[IEC13a] IEC. GRAFCET specification language for sequential function charts. International Standard 60848 :2013, International Electrotechnical Commission, Geneva, 02 2013.

[IEC13b] IEC. Programmable controllers – Part 3 : Programming languages. International Standard 61131-3, International Electrotechnical Commission, Geneva, 02 2013.

[ISO88] ISO/IEC. ESTELLE – A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1988.

[ISO89] ISO/IEC. LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1989.

[ISO11] ISO/IEC. High-level Petri Nets – Part 2 : Transfer Format. International Standard 15909-2 :2011, International Organization for Standardization – Information Technology – Systems and Software Engineering, Geneva, 2011.

[ISO12] ISO/IEC. Object Management Group Unified Modeling Language (OMG UML). International Standard 19505 :2012-1 and 19505 :2012-2, International Organization for Standardization – Information Technology – Systems and Software Engineering, Geneva, 2012.

[ISO13] ISO/IEC. Object Management Group Business Process Model and Notation. International Standard 19510 :2013, International Organization for Standardization – Information Technology – Systems and Software Engineering, Geneva, 2013.

[ISO17] ISO/IEC. Object Management Group Systems Modeling Language (OMG SysML). International Standard 19514 :2017, International Organization for Standardization – Information Technology – Systems and Software Engineering, Geneva, 2017.

[Jan84] Ryszard Janicki. Nets, Sequential Components and Concurrency Relations. *Theoretical Computer Science*, 29 :87–121, 1984.

[Jen92] Kurt Jensen. *Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use – Volume 1.* EATCS Monographs on Theoretical Computer

Science. Springer, 1992.

[JK07] Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments (ALENEX 2007), New Orleans, Louisiana, USA*. SIAM, January 2007.

[JKM12] Kurt Jensen, Lars Kristensen, and Thomas Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429 :169–179, 2012.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15), Mumbai, India*, pages 247–259. ACM, January 2015.

[JLL77] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of Some Problems in Petri Nets. *Theoretical Computer Science*, 4(3) :277–299, June 1977.

[JM85] J.L. Johnson and Tadao Murata. Structure matrices for petri nets and their applications. *Journal of the Franklin Institute*, 319(3) :299–309, March 1985.

[JMM+19] Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon R. H. Schiffelers, Harco Kuppens, and Frits W. Vaandrager. RERS 2019 : Combining Synthesis with Real-World Models. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19), Part III : TOOLympics, Prague, Czech Republic*, pages 101–115. Springer, April 2019.

[JT96] David S. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability – 2nd DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, January 1996.

[KA06] Andrei Karatkevich and Grzegorz Andrzejewski. Hierarchical Decomposition of Petri Nets for Digital Microsystems Design. In *Proceedings of the 2006 International Conference on Modern Problems of Radio Engineering, Telecommunications, and Computer Science, Lviv-Slavsko, Ukraine*, pages 518–521. IEEE, February–March 2006.

[Kar07] Andrei Karatkevich. *Dynamic Analysis of Petri Net-Based Discrete Systems*, volume 356 of *Lecture Notes in Control and Information Sciences*. Springer,

2007.

[KE96] Andrei Kovalyov and Javier Esparza. A Polynomial Algorithm to Compute the Concurrency Relation of Free-choice Signal Transition Graphs. In *Proceedings of the 3rd Workshop on Discrete Event Systems (WODES'96), Edinburgh, Scotland, UK*, pages 1–6, 1996.

[KEB94] Maciej Koutny, Javier Esparza, and Eike Best. Operational Semantics for the Petri Box Calculus. In Bengt Jonsson and Joachim Parrow, editors, *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR'94), Uppsala, Sweden*, volume 836 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 1994.

[KHHH+21] Fabrice Kordon, Lom Messah Hillah, Francis Hulin-Hubard, Loïg Jezequel, and Emmanuel Paviot-Adet. Study of the efficiency of model checking techniques using results of the MCC from 2015 To 2019. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2021.

[KJ07] Janez Konc and Dusanka Janezic. An Improved Branch and Bound Algorithm for the Maximum Clique Problem. *MATCH – Communications in Mathematical and in Computer Chemistry*, 58(3) :569–590, January 2007.

[KLM+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin : High-Performance Language-Independent Model Checking. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), London, UK*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707. Springer, April 2015.

[Kor92] Fabrice Kordon. *Prototypage de systèmes paralèlles à partir de réseaux de Petri colorés, application au langage Ada dans un environnement centralisé ou réparti*. PhD thesis, Université Pierre et Marie Curie (Paris 6), 1992.

[Kov92] Andrei Kovalyov. Concurrency Relations and the Safety Problem for Petri Nets. In Kurt Jensen, editor, *Proceedings of the 13th International Conference on Application and Theory of Petri Nets (ICATPN'92), Sheffield, UK*, volume 616 of *Lecture Notes in Computer Science*, pages 299–309. Springer, June 1992.

[Kov00] Andrei Kovalyov. A Polynomial Algorithm to Compute the Concurrency Relation of a Regular STG. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, chapter 6, pages 107–126. Springer, Boston, MA, USA, January 2000.

[KSM12] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Graph Symmetry Detection and Canonical Labeling : Differences and Synergies. In Andrei Voronkov, editor, *Turing-100 - The Alan Turing Centenary, Manchester,*

UK, volume 10 of *EPiC Series in Computing*, pages 181–195. EasyChair, June 2012.

[KW15] Andrei Karatkevich and Remigiusz Wiśniewski. Relation Between SM-Covers and SM-Decompositions of Petri Nets. In *Proceedings of the International Conference of Computational Methods in Sciences and Engineering (ICCMSE'15)*, volume 1702-1, page 100012. AIP Publishing, December 2015.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[LAV90] Jean Christophe Lloret, Pierre Azéma, and François Vernadat. Compositional Design and Verification of Communication Protocols, Using Labelled Petri Nets. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV'90), New Brunswick, NJ, USA,*, volume 531 of *Lecture Notes in Computer Science*, pages 96–105. Springer, 1990.

[LCA13] José Luis López-Presa, Luis Núñez Chiroque, and Antonio Fernández Anta. Novel techniques for automorphism group computation. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, pages 296–307. Springer, June 2013.

[LJM17] Chu-Min Li, Hua Jiang, and Felip Manya. On Minimization of the Number of Branches in Branch-and-Bound Algorithms for the Maximum Clique Problem. *Computers & Operations Research*, 84(?) :1 – 15, August 2017.

[LL98] Bilung Lee and Edward Lee. Hierarchical Concurrent Finite State Machines in Ptolemy. In *Proceedings of the 1st International Conference on Application of Concurrency to System Design (ACSD'98), Fukushima, Japan*, pages 34–40. IEEE Computer Society, March 1998.

[LRR03] Kamal Lodaya, D. Ranganayakulu, and K. Rangarajan. Hierarchical Structure of 1-Safe Petri Nets. In Vijay A. Saraswat, editor, *Proceedings of the 8th Asian Computing Science Conference on Programming Languages and Distributed Computation (ASIAN'03), Mumbai, India*, volume 2896 of *Lecture Notes in Computer Science*, pages 173–187. Springer, December 2003.

[MCvdA13] Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Hierarchical Conformance Checking of Process Models Based on Event Logs. In José-Manuel Colom and Jörg Desel, editors, *Proceedings of the 34th International Conference on Applications and Theory of Petri Nets (PETRI NETS'13), Milan, Italy*, volume 7927 of *Lecture Notes in Computer Science*, pages 291–310. Springer, June 2013.

[Mic05] Piotr Miczulski. Reprezentacja hierarchicznego grafu znakowań z wykorzystaniem funkcji monotonicznych. In Grzegorza Andrzejewskiego and Zbigniewa Skowrońskiego, editors, *Informatyka - sztuka czy rzemiosło*. University of Zielona Góra Printing House, Poland, jun 2005.

[Mil79] Gary Miller. Graph Isomorphism, General Remarks. *Journal of Computer and System Sciences*, 18(2) :128–142, 1979.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[MO98] José Monteiro and Arlindo Oliveira. Finite State Machine Decomposition For Low Power. In *Proceedings 2000. Design Automation Conference (IEEE Cat. No.00CH37106)*, pages 758–763, May 1998.

[MO00] José Monteiro and Arlindo Oliveira. FSM decomposition by direct circuit manipulation applied to low power design. In *Proceedings of ASP-DAC 2000, Asia and South Pacific Design Automation Conference 2000, Yokohama, Japan*, pages 351–358, June 2000.

[MO04] José Monteiro and Arlindo Oliveira. FSM Decomposition by Direct Circuit Manipulation Applied to Low Power Design. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, August 2004.

[MOW09] Stephan Mennicke, Olivia Oanea, and Karsten Wolf. Decomposition into open nets. In Thomas Freytag and Andreas Eckleder, editors, *Algorithmen und Werkzeuge für Petrinetze (AWPN'09), Karlsruhe, Germany*, pages 29–34. CEUR-WS.org, September 2009.

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60 :94–112, September 2014.

[MS81] J. Martínez and Manuel Silva. A Simple and Fast Algorithm to Obtain All Invariants of a Generalized Petri Net. In Claude Girault and Wolfgang Reisig, editors, *Application and Theory of Petri Nets, Germany*, volume 52 of *Informatik-Fachberichte*, pages 301–310. Springer, September 1981.

[Mur89] Tadao Murata. Petri Nets : Analysis and Applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[MW04] Thomas Mailund and Michael Westergaard. Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th Tools and Algorithms for the Construction and Analysis of Systems, Barcelona, Spain*, volume 2988 of *Lecture Notes in Computer Science*, pages 177–191. Springer, March–April 2004.

[NP92] Tobias Nipkow and Lawrence C. Paulson. Isabelle-91. In Deepak Kapur,

editor, *Proceedings of the 11th International Conference on Automated Deduction CADE-11 (Saratoga Springs, NY, USA)*, volume 607 of *Lecture Notes in Computer Science*, pages 673–676. Springer, June 1992.

[NPB14] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1) :53–58, 2014.

[Old86] Ernst-Rüdiger Olderog. Operational Petri Net Semantics for CCSP. In Grzegorz Rozenberg, editor, *Proceedings of the 7th European Workshop on Applications and Theory of Petri Nets (APN'87), Oxford, UK*, volume 266 of *Lecture Notes in Computer Science*, pages 196–223. Springer, 1986.

[Old91] Ernst-Rüdiger Olderog. *Nets, Terms, and Formulas : Three Views of Concurrent Processes and Their Relationship*. Cambridge University Press, 1991.

[Pau89] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3) :363–397, 1989.

[Pau16] Loïc Paulevé. Goal-Oriented Reduction of Automata Networks. In Ezio Bartocci, Pietro Liò, and Nicola Paoletti, editors, *Proceedings of the 14th International Conference on Computational Methods in Systems Biology (CMSB'16), Cambridge, UK*, volume 9859 of *Lecture Notes in Computer Science*, pages 252–272. Springer, September 2016.

[PBV11] Florent Peres, Bernard Berthomieu, and François Vernadat. On the composition of time Petri nets. *Discrete Event Dynamic Systems*, 21(3) :395–424, 2011.

[PC98] Enric Pastor and Jordi Cortadella. Efficient Encoding Schemes for Symbolic Analysis of Petri Nets. In Patrick Dewilde, Franz J. Rammig, and Gerry Musgrave, editors, *Proceedings on the International Conference on Design, Automation and Test in Europe (DATE'98), Paris, France*, pages 790–795. IEEE Computer Society, February 1998.

[PCP99] Enric Pastor, Jordi Cortadella, and Marco A. Peña. Structural Methods to Improve the Symbolic Analysis of Petri Nets. In Susanna Donatelli and H. C. M. Kleijn, editors, *Proceedings of the 20th International Conference Application and Theory of Petri Nets (ICATPN'99), Williamsburg, VA, USA*, volume 1639 of *Lecture Notes in Computer Science*, pages 26–45. Springer, June 1999.

[PCR01] Enric Pastor, Jordi Cortadella, and Oriol Roig. Symbolic Analysis of Bounded Petri Nets. *IEEE Trans. Computers*, 50(5) :432–448, 2001.

[Pet77] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3) :223–252, 1977.

[Pet79] Carl A. Petri. Concurrency. In Wilfried Brauer, editor, *Net Theory and Applications, Proceedings of the Advanced Course on General Net Theory*

of Processes and Systems, Hamburg, Germany, volume 84 of *Lecture Notes in Computer Science*, pages 251–260. Springer, October 1979.

[Pet90] Antoine Petit. Distribution and Synchronized Automata. *Theoretical Computer Science*, 76(2–3) :285–308, 1990.

[RE96] Grzegorz Rozenberg and Joost Engelfriet. Elementary Net Systems. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I : Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 12–121. Springer, September 1996.

[Rei85] Wolfgang Reisig. *Petri Nets : An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[Ros10] A. William Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010.

[RPU+07] Ivo Raedts, Marija Petkovic, Yaroslav S. Usenko, Jan Martijn E. M. van der Werf, Jan Friso Groote, and Lou J. Somers. Transformation of BPMN Models for Behaviour Analysis. In Juan Carlos Augusto, Joseph Barjis, and Ulrich Ultes-Nitsche, editors, *Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS-2007), Funchal, Portugal*, pages 126–137. INSTICC PRESS, June 2007.

[SAW13] Łukasz Stefanowicz, Marian Adamski, and Remigiusz Wiśniewski. Application of an Exact Transversal Hypergraph in Selection of SM-Components. In *Technological Innovation for the Internet of Things - 4th IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2013, Costa de Caparica, Portugal, April 15-17, 2013. Proceedings*, volume 394, pages 250–257, April 2013.

[Sch03] Karsten Schmidt. Using Petri Net Invariants in State Space Construction. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), Warsaw, Poland*, volume 2619 of *Lecture Notes in Computer Science*, pages 473–488. Springer, April 2003.

[Sch06] Karsten Schmidt. Automated generation of a progress measure for the sweep-line method. *International Journal on Software Tools for Technology Transfer*, 8(3) :195–203, June 2006.

[Sil85] Manuel Silva. *Las Redes de Petri : en la Automática y la Informática*. Editorial AC Madrid, 1985.

[SJMvdP17] Bernhard Steffen, Marc Jasper, Jeroen Meijer, and Jaco van de Pol. Property-Preserving Generation of Tailored Benchmark Petri Nets. In *Proceedings of the 17th International Conference on Application of Concurrency to System Design (ACSD'17), Zaragoza, Spain*, pages 1–8. IEEE

Computer Society, June 2017.

[SMRH13] Pablo San Segundo, Fernando Matía, Diego Rodríguez-Losada, and Miguel Hernando. An Improved Bit Parallel Exact Maximum Clique Algorithm. *Optimization Letters*, 7(3) :467–479, January 2013.

[Som01] Fabio Somenzi. Efficient manipulation of decision diagrams. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 3(2) :171–181, 2001.

[ST18] Slavomír Simonák and Martin Tomásek. ACP Semantics for Petri Nets. *Computing and Informatics*, 37(6) :1464–1484, 2018.

[Sta90] Peter H. Starke. *Analyse von Petri-Netz-Modellen*. Leitfäden und Monographien der Informatik. Teubner, Stuttgart, Germany, 1990.

[SY95] Alexei Semenov and Alexandre Yakovlev. Combining Partial Orders and Symbolic Traversal for Efficient Verification of Asynchronous Circuits. In Tatsuo Ohtsuki and Steven Johnson, editors, *Proceedings of the 12th International Conference on Computer Hardware Description Languages and their Applications (CHDL'95), Makuhari, Chiba, Japan*. IEEE, August–September 1995.

[TA13] Jacek Tkacz and Marian Adamski. Structured Mapping of Petri Net States and Events for FPGA Implementations. *International Journal of Electronics and Telecommunications*, 59(no. 4) :331–339, December 2013.

[Tau89] Dirk Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, volume 369 of *Lecture Notes in Computer Science*. Springer, 1989.

[Tau90] Dirk Taubner. Representing CCS Programs by Finite Predicate/Transition Nets. *Acta Informatica*, 27(6) :533–565, 1990.

[TM15] Yann Thierry-Mieg. Symbolic Model-Checking Using ITS-Tools. In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), London, UK*, volume 9035 of *Lecture Notes in Computer Science*, pages 231–237. Springer, April 2015.

[TMPHK09] Yann Thierry-Mieg, Denis Poitrenaud, Alexandre Hamez, and Fabrice Kordon. Hierarchical Set Decision Diagrams and Regular Models. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09), York, UK*, volume 5505 of *Lecture Notes in Computer Science*, pages 1–15. Springer, March 2009.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990, Bonn, Germany*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, June

1989.

[vGGS08] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. On Synchronous and Asynchronous Interaction in Distributed Systems. In Edward Ochmanski and Jerzy Tyszkiewicz, editors, *Proceedings of the 33rd International Symposium on the Mathematical Foundations of Computer Science (MFCS'08), Torun, Poland*, volume 5162 of *Lecture Notes in Computer Science*, pages 16–35. Springer, August 2008. Full version available from http://arxiv.org/abs/0901.0048.

[vGGSU12] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke-Uffmann. On Distributability of Petri Nets. In Lars Birkedal, editor, *Proceedings of the 15th International Conference on the Foundations of Software Science and Computational Structures (FoSSaCS'12), Tallinn, Estonia*, volume 7213 of *Lecture Notes in Computer Science*, pages 331–345. Springer, March–April 2012. Full version available from http://arxiv.org/abs/1207.3597.

[vGGSU13] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke-Uffmann. On Characterising Distributability. *Logical Methods in Computer Science*, 9(3), 2013.

[vV87] Rob J. van Glabbeek and Frits W. Vaandrager. Petri Net Models for Algebraic Theories of Concurrency. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *Proceedings of the 1st International Conference on Parallel Architectures and Languages Europe (PARLE'87), Volume II, Eindhoven, The Netherlands*, volume 259 of *Lecture Notes in Computer Science*, pages 224–242. Springer, 1987.

[Wis11] Monika Wisniewska. Autorski system wspomagajacy proces dekompozycji sieci Petriego na podsieci typu automatowego. *Pomiary, Automatyka, Kontrola*, Vol. 57(nr 8) :948–950, 2011.

[Wiś12] Monika Wiśniewska. *Application of Hypergraphs in Decomposition of Discrete Systems*, volume 23 of *Lecture Notes in Control and Computer Science*. University of Zielona Góra Printing House, Poland, 2012.

[Wiś18] Remigiusz Wiśniewski. Dynamic Partial Reconfiguration of Concurrent Control Systems Specified by Petri Nets and Implemented in Xilinx FPGA Devices. *IEEE Access*, 6 :32376–32391, 2018.

[WKA+18] Remigiusz Wiśniewski, Andrei Karatkevich, Marian Adamski, Anikó Costa, and Luís Gomes. Prototyping of Concurrent Control Systems With Application of Petri Nets and Comparability Graphs. *IEEE Transactions on Control Systems Technology*, 26(2) :575–586, 2018.

[WKAK14] Remigiusz Wiśniewski, Andrei Karatkevich, Marian Adamski, and Daniel Kur. Application of Comparability Graphs in Decomposition of Petri Nets. In *Proceedings of the 7th International Conference on Human System*

Interactions (HSI'14), Costa da Caparica, Portugal, pages 216–220. IEEE, June 2014.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods : Practice and experience. *ACM Computer Surveys*, 41(4) :19 :1–19 :36, 2009.

[Wol18] Karsten Wolf. Petri Net Model Checking with LoLA 2. In Victor Khomenko and Olivier H. Roux, editors, *Proceedings of the 39st International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS'18), Bratislava, Slovakia*, volume 10877 of *Lecture Notes in Computer Science*, pages 351–362. Springer, June 2018.

[WWA12] Remigiusz Wiśniewski, Monika Wiśniewska, and Marian Adamski. Wielomianowy algorytm wyznaczania hipergrafu współbiezno'sci w sieciach Petriego swobodnego wyboru. *Pomiary, Automatyka, Kontrola*, 58 :650–652, January 2012.

[WWJ19] Remigiusz Wiśniewski, Monika Wiśniewska, and Marcin Jarnut. C-Exact Hypergraphs in Concurrency and Sequentiality Analyses of Cyber-Physical Systems Specified by Safe Petri Nets. *IEEE Access*, 7 :13510–13522, January 2019.

[Zai04] Dmitry Zaitsev. Decomposition of Petri nets. *Cybernetics and Systems Analysis*, 40 :739–746, January 2004.

[ZLHX14] Zhaoyang Zhou, Chu Min Li, Chong Huang, and Ruchu Xu. An Exact Algorithm with Learning for the Graph Coloring Problem. *Computers & OR*, 51 :282–301, 2014.