THÈSE DE DOCTORAT DE

L'ÉCOLE CENTRALE DE NANTES

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Rémi PARROT

Réseaux de Petri temporisés pour la synthèse de circuits pipelinés

Thèse présentée et soutenue à l'École Centrale de Nantes, le 15 Novembre 2022
Unité de recherche : UMR 6004, Laboratoire des Sciences du Numérique de Nantes (LS2N)

Rapporteurs avant soutenance :

François VERNADAT Professeur des Universités, INSA Toulouse
Florent de DINECHIN Professeur des Universités, INSA Lyon

Composition du Jury :

Présidente :	Isabel DEMONGODIN	Professeure des Universités, Aix-Marseille Université
Dir. de thèse :	Olivier H. ROUX	Professeur des Universités, École Centrale de Nantes
Co-dir. de thèse :	Malek GHANES	Professeur des Universités, École Centrale de Nantes
Co-encadrant de thèse :	Mikaël BRIDAY	Maître de Conférences, École Centrale de Nantes

Invitée :

Miassa TALEB Ingénieure de recherche, Renault, Guyancourt

[...]

Tout fut
Débâcle et dispersion
Turbulences et gaspillage
Avant que le rythme
Ne prenne possession
De l'espace

[...]

Rythmes, Andrée Chedid

Remerciement

Que ce serait prétentieux de prétendre avoir fait tout ce travail seul et solitaire ! J'aimerais adresser ma plus grande reconnaissance à tout mon entourage, toutes ces personnes qui m'ont accompagné, soutenu, conseillé, fait marrer, bref, toutes ces personnes qui ont été présentes ! La liste est longue, alors allons y pour une tentative de remerciements !

Tout d'abord, merci à François Vernadat, Florent de Dinechin et Isabel Demongodin d'avoir accepté de faire partie de mon jury de thèse. Je suis très reconnaissant de l'intérêt qu'ils ont porté à mon travail et je les remercie pour tous leurs conseils, remarques et critiques. J'espère que nous trouverons de nouvelles occasions d'échanger sur nos travaux et peut-être même d'en réaliser ensemble !

Merci à l'équipe Système Temps Réel qui a été ma famille d'accueil au LS2N, merci pour l'accueil chaleureux, les réunions dans la bonne humeur et les nombreuses pauses cafés ! Tout particulièrement, merci à Didier pour m'avoir initié au travail de chercheur et donner envie de continuer.

Bien évidemment, un immense merci à mes papas de doctorat : Olivier et Mikaël ! Je ne compte pas les fois où je suis venu pleurnicher dans leurs bureaux, et ils ont su à chaque fois me remonter avec leur bonne humeur et leur positivité. Je suis très reconnaissant de les avoir eut comme encadrants de thèse et comme amis !

Merci beaucoup à Miassa qui a su assurer le lien avec Renault, merci pour sa patience et sa compréhension.

Merci à toutes les personnes sans qui mon intégration au LS2N n'aurait pas été possible. Je pense tout particulièrement à Virginie, Emily, Christine et Edith qui ont su gérer chacune de mes demandes avec patience et efficacité !

Merci à ma famille qui a toujours été le rocher sur lequel je viens me reposer quand la mer est trop agitée. Merci à mes parents pour leur soutien indéfectible et pour leur patience face à mes tentatives maladroites d'expliquer mon travail ! Merci à mes sœurs, Lise et Marie, pour leur soutien, leur réconfort et pour avoir toujours été mes modèles.

Je voudrais également remercier toutes les autres belles personnes que j'ai eut la chance de rencontrer au laboratoire. Merci à la team des doctorants et post-doctorants pour les pauses, les madeleines et les barres de rire ! Merci à Sam, Valentin, René, Mehdi, Yuxin,

Adrien, Dridri, Houssein, Maël, Nicolas, Antoine, Lolee, Lily, Bastien, Ralph, Ismaila, Hugo, Selma, Devasmito, François, Modan, Oriane et Charlotte. Merci aussi à Seb un ami de longue date qui s'est retrouvé être mon cobureau ! Merci aussi à Dominique pour son accueil sous la neige Montréalaise !

Merci à Ibis, qui a été une belle rencontre pleine de surprise au détour d'une conférence, merci pour son humour, sa bonne humeur et son féminisme sans faille !

Ma vie à Nantes a également été rythmée par la présence de tous mes amis de longue date, que je tiens à remercier. Merci à la team des grimpeurs (et buveurs) nantais : Ken, Nico, Buzz, Fanch, Cam, Julie, Yann, Camille, Manon, Diane et Louis, sans oublier Marie ! Merci à la team du Pouce d'Or : Nico, Manu, Matho et Lukas, pour tous les premiers de l'an et les randos, passés ensemble dans le froid et le brouillard ! Merci au trio inséparable Jrôme, Yaya et Hugo pour les blagues, les pirateries, les histoires de cœur et la grande philosophie !

Merci à Léna, une amie de passage, qui m'a apporté du présent, du soleil et plus de cheddar, en période sombre de confinement.

Je n'oublie pas non plus les amis de plus longue date, le groupe de la Borde Basse : Grégoire, Fifi, Hugo et Sylvain. Merci pour être toujours là, même si l'on ne se voit que rarement !

Merci à tous !

Sommaire

Introduction **13**

1 Prérequis **21**
1.1 Méthodes formelles . . . 21
1.1.1 Ensembles . . . 21
1.1.2 Système de Transition . . . 22
1.1.3 Bisimulation . . . 23
1.1.4 Logiques temporelles . . . 24
1.2 Conception de circuits . . . 24
1.2.1 Formats de données . . . 24
1.2.2 Circuit logique programmable . . . 25

2 Un modèle formel pour les circuits pipelinés **27**
2.1 Modéliser un circuit synchrone . . . 27
2.1.1 Automates temporisés . . . 28
2.1.2 Extension des Réseaux de Petri avec du temps . . . 28
2.1.3 Notre choix . . . 29
2.2 Réseau de Petri . . . 29
2.2.1 Définition . . . 29
2.2.2 Sémantique d'entrelacement . . . 30
2.2.3 Sémantique à pas maximaux . . . 31
2.2.4 Expressivité . . . 32
2.3 Réseau de Petri Temporisé . . . 32
2.3.1 Tir en trois phases . . . 33
2.3.2 Définition . . . 34
2.3.3 Comparaison tir atomique / tir en trois phases . . . 36
2.4 Réseau de Petri Temporisé avec reset et transitions retardables . . . 37
2.4.1 Définition . . . 38
2.4.2 Exemple . . . 41

2.5 Réseau de Petri Temporisé avec transitions retardables 41
2.5.1 Reset explicite 42
2.5.2 Définition 44

3 Propriétés des Réseaux de Petri Temporisés avec transitions retardables (DTPNs) 45
3.1 Expressivité des DTPNs par rapport aux classes à intervalles 45
3.1.1 Définitions des différentes classes 46
3.1.2 Expressivité des classes DTPN et DTPN^{dt} 48
3.1.3 Les classes en temps dense 49
3.1.4 Les classes en temps discret 49
3.2 Une sémantique symbolique pour les DTPNs 52
3.2.1 Le délai dynamique 52
3.2.2 Les états symboliques 53
3.2.3 La sémantique symbolique 55
3.3 Expressivité des RTPNs 55
3.3.1 Sémantique symbolique pour les RTPNs 56
3.3.2 Définition d'un Automate Temporisé 57
3.3.3 Traduction vers un automate temporisé à une horloge 58
3.3.4 Corollaires sur le langage 60
3.4 Complexité théorique et pratique 61
3.4.1 Problèmes d'accessibilité 61
3.4.2 Calcul du successeur symbolique 63
3.5 Algorithmes d'exploration de l'espace d'états symboliques 64
3.5.1 Successeur d'un état 64
3.5.2 Successeurs d'un état symbolique 66
3.5.3 Graphe d'états symboliques 70

4 Synthèse de pipeline d'un circuit synchrone 71
4.1 Définition du problème 71
4.1.1 Qu'est-ce qu'un circuit synchrone ? 71
4.1.2 Le pipeline 74
4.1.3 Le pliage 77
4.2 Synthèse de pipeline 80
4.2.1 Bibliographie 80

4.2.2 Algorithme glouton . . . 81
4.2.3 Leiserson & Saxe . . . 85
4.3 Synthèse de pipeline à partir d'un modèle RTPN . . . 87
4.3.1 Bibliographie . . . 87
4.3.2 Modélisation d'un circuit synchrone par un RTPN . . . 88
4.3.3 Construction du pipeline . . . 92
4.3.4 Résultats expérimentaux . . . 95
4.4 Synthèse de pliage . . . 98
4.4.1 Bibliographie . . . 98
4.4.2 Synthèse de pliage avec RTPN . . . 99
4.4.3 En pratique . . . 104

5 Application : compilation Simulink vers VHDL 107
5.1 État de l'art . . . 107
5.1.1 Choix des formats de données . . . 108
5.1.2 Synthèse de VHDL . . . 110
5.2 Notre outil de compilation . . . 112
5.2.1 Parseur Simulink . . . 113
5.2.2 Évaluation des formats de données . . . 114
5.2.3 Évaluation des délais des opérateurs . . . 115
5.2.4 Synthèse de pliage . . . 116
5.2.5 Génération du VHDL . . . 117
5.3 Expérimentations . . . 118
5.3.1 Application cible : un filtre numérique . . . 118
5.3.2 Matériel . . . 119
5.3.3 Configuration expérimentale . . . 120
5.3.4 Résultats . . . 123

Conclusion 129

Acronymes 133

Bibliographie 135
5.4 Publications . . . 141

INTRODUCTION

Depuis l'avènement de l'informatique au milieu du vingtième siècle, son usage dans nos sociétés n'a cessé de grandir. Elle occupe aujourd'hui une place importante dans notre quotidien : ordinateur de bureau, ordinateur personnel, smartphone, smart TV, etc. Il devient de plus en plus indispensable pour les individus d'avoir accès à un système informatique et à un réseau de communications, que ce soit pour le travail, pour les déplacements, pour les achats de premières nécessités ou pour les loisirs. L'accès aux systèmes informatiques est alors un facteur d'intégration sociale, entre autres, de par la transition des réseaux sociaux vers le numérique. Cette augmentation massive des systèmes informatiques est, de plus, généralisée dans tous les pays du globe.

L'utilisation grandissante de l'informatique se traduit notamment par de nombreux systèmes embarqués. Bien qu'invisibles pour la plupart des utilisateurs, ils occupent une large partie de cas d'usage de l'informatique au quotidien : ABS ou régulateur pour l'automobile, capteurs optiques pour la maintenance de réseaux ferroviaires, routeurs pour les réseaux de téléphonie, moniteurs de glucose ou de tension pour les suivis médicaux, etc. L'émergence de l'Internet des Objets (IoT), dont la plupart des nœuds du réseau sont des systèmes embarqués, a grandement renforcé leur utilisation.

La conception du système est spécifique à chaque application, elle nécessite une optimisation qui lui est propre. De nombreux travaux s'intéressent à la conception de « petits » systèmes, consommant un minimum de ressources matérielles et énergétiques. On pourrait penser que l'objectif est de réduire la consommation liée aux utilisations existantes (sobriété technologique). Mais il s'agit en fait, d'une volonté des industries de création de nouvelles utilisations de systèmes embarqués. Comme le montre ces travaux [KRP16], la vente de systèmes électroniques n'a cessé d'augmenter depuis les 30 dernières années (dans cette étude l'échelle d'un pays, la Suède). Par exemple, dans le domaine de l'automobile, depuis les années 70 les constructeurs ont peu à peu remplacé des systèmes de contrôle mécaniques par des systèmes électroniques [Lar14]. Ainsi, en 1977 les composants électroniques constituaient 5% du prix du véhicule, dans les années 2010 ils constituent 40% du prix. Récemment, ces « petits » systèmes trouvent toute leur utilité dans le Edge Computing, consistant à traiter les données au plus proche du point d'acquisition, qui

s'inscrit dans la démarche de l'Internet des Objets. En effet, une application typique de Edge Computing n'a pas besoin d'une grande rapidité de traitement, mais nécessite une faible consommation énergétique.

Généralement, ces systèmes embarqués contiennent un microcontrôleur, c'est-à-dire un système qui intègre dans un seul composant tous les éléments essentiels d'un ordinateur. Ce dernier pourra être programmé et associé à des composants périphériques (de la mémoire supplémentaire par exemple) pour répondre au besoin d'utilisation. De plus, il existe une grande variété d'architectures possédant des propriétés différentes (fréquence, jeu d'instructions, nombre de cœurs, périphériques, etc) qui permettent de répondre plus finement aux spécificités de chaque application. Toutefois, dans certains cas, l'adaptabilité des microcontrôleurs n'est pas suffisante. Les concepteurs doivent alors descendre d'un niveau d'abstraction, pour se tourner vers les circuits logiques. Ces derniers permettent, entre autre, de réaliser autant de calcul en parallèle que l'on souhaite, ce qui peut permettre de répondre à des besoins de rapidité de calcul. Les circuits logiques peuvent être implémentés sous forme de Application-Specific Integrated Circuit (ASIC), c'est-à-dire fondu définitivement, ou bien sur un circuit logique « programmable » comme un Field-Programmable Gate Array (FPGA), c'est-à-dire sur un composant qui peut-être reconfiguré a posteriori.

Enfin, pour chacun de ces systèmes, le concepteur doit s'assurer de son bon fonctionnement. Suivant le niveau de criticité de l'application, les méthodes mises en œuvre sont différentes. Par exemple, un piège photo pour recenser la biodiversité animale ne requiert pas les mêmes certifications de fonctionnement qu'un système médical, comme un pacemaker. La méthode la plus fréquemment utilisée est le test. Elle consiste à générer des jeux de données d'entrées pour une grande quantité de cas d'usage, puis à constater le bon fonctionnement du système sur ces derniers. La principale faiblesse de cette approche est sa non-exhaustivité. D'autre part, il existe les méthodes formelles. Elles reposent sur une modélisation rigoureuse des systèmes et permettent de vérifier des propriétés pour un ensemble possiblement infini de cas d'usage. Elles sont cependant peu utilisées (par rapport aux tests) du fait de leur difficulté de mise en œuvre.

Cas d'étude

Notre travail s'inscrit dans le cadre d'une chaire industrielle avec l'entreprise automobile Renault.

Une équipe de recherche travaille sur un nouveau véhicule électrique et en particulier sur la gestion de son énergie. Elle développe un nouveau modèle de chargeur de batterie réversible, embarqué sur le véhicule. Un chargeur réversible permet de brancher son véhicule à une station de recharge, puis au choix de prendre de l'énergie du réseau ou d'envoyer un surplus d'électricité sur le réseau, *Vehicule to Grid* (V2G). Il peut également permettre d'envoyer le surplus d'électricité vers un appareil électrique, *Vehicule to Load* (V2L).

Pour contrôler ce chargeur, l'équipe de Renault a conçu une commande de systèmes dynamiques spécifique. Cette dernière a été développée sur Matlab/Simulink, qui est un outil de conception et simulation de systèmes dynamiques. Cela facilite le développement, car l'équipe possède déjà une grande expertise sur cet outil.

Sur leurs précédents projets, cette équipe implémentait les commandes développées dans Simulink sur un microcontrôleur. Cependant, la commande de ce chargeur nécessite une fréquence d'échantillonnage trop élevée pour un microcontrôleur. Le calcul doit être effectué sur une fenêtre de $3.6\mu s$, ce qui correspond à une fréquence d'échantillonnage d'environ 280kHz. Comme un microcontrôleur opère de manière séquentielle, une implémentation sur ce dernier dépasserait la fenêtre de calcul. Il serait en fait possible d'utiliser un microcontrôleur, mais cela nécessiterait un composant plus puissant et donc plus coûteux, ce qui serait un mauvais compromis avec leur contrainte de grande série.

L'équipe s'est donc tournée vers la conception de circuit logique. Souhaitant garder la possibilité de modifier l'implémentation (pour une mise à jour future), ils ont opté pour une cible FPGA, c'est-à-dire un circuit reconfigurable.

Toutefois, ils ne possédaient pas d'expertise pour la conception de circuit en interne. Ils souhaitaient alors avoir un outil automatisant le passage d'une modélisation Simulink à un circuit logique, décrit à l'aide du langage VHDL. L'objectif de cet outil n'est pas d'effectuer simplement une compilation (Simulink vers VHDL), mais de réaliser des optimisations sur le circuit généré pour qu'il réponde à un besoin spécifique. Les spécifications peuvent se résumer à une cible FPGA qui possède des ressources matérielles limitées, à une fréquence d'échantillonnage et à des précisions de calcul.

Synthèse de circuits logiques

La synthèse automatique de circuits logiques à partir d'une description plus haut niveau est une approche qui s'appelle High Level Synthesis (HLS). Généralement, elle

part d'une description sous forme de langage de programmation itératif comme le C. Dans notre cas, la description est sous forme d'un graphe de flot de données ce qui facilite largement le travail. En effet, chaque bloc Simulink peut être considéré comme un sous-circuit dont il est relativement facile d'écrire une implémentation générique. Ensuite, il suffit de suivre la structure du modèle Simulink pour déterminer les branchements entre les sous-circuits.

Toutefois, si l'on s'en tient à cette approche naïve le circuit généré pourra ne pas tenir les restrictions de ressources disponibles et de temps de calcul. Des optimisations doivent en général être réalisées pour satisfaire les exigences.

La première optimisation concerne le choix des formats de données internes au calcul. En effet, s'il est possible de spécifier des formats de données dans Simulink, par défaut ce dernier utilise des nombres à virgule flottante. Il n'est pas nécessaire, dans la plupart des cas, pour un circuit d'effectuer ses calculs sur des flottants. On préfèrera utiliser une représentation en virgule fixe, ce qui permet de limiter les ressources consommées et de réduire le temps de calcul. Cependant, les formats de données d'entrée et de sorties étant fixés, une précision sur les données de sortie est implicitement définie. Les formats de données internes au calcul doivent donc être choisis avec précaution. Il ne faut pas faire d'erreurs d'arrondi trop grandes qui auraient des répercussions sur la précision des sorties. Il existe alors une configuration optimale des formats de données interne au circuit, qui minimise les ressources consommées tout en garantissant une erreur maximale de sortie.

Une deuxième optimisation concerne l'implémentation générique de chaque bloc de la représentation Simulink. Il existe en général plusieurs circuits permettant de réaliser le calcul d'un bloc Simulink. Par exemple, il existe plusieurs circuits réalisant une multiplication par une constante k. Il peut être implémenté par un multiplieur simple prenant la constante k sur une de ses deux entrées. Mais suivant la valeur de la constante k, il peut être avantageux d'utiliser plutôt des décalages et des additioneurs (*Shift&Add*). Il est également possible de l'implémenter à partir de sa table de vérité (*KCM* [Cha94]), ce qui peut-être particulièrement avantageux pour une application sur FPGA. Généralement le choix d'implémentation est un compromis entre latence (temps de calcul) et consommation de ressources matérielles. Il existe donc un (ou plusieurs) choix optimal(ux) d'implémentation de chaque bloc permettant de ne pas dépasser les limites de ressources et de temps de calcul.

Un troisième optimisation est le choix de l'ordre d'exécution des calculs dans le circuit. Initialement, dans la représentation Simulink toutes les opérations sont exécutées

en parallèle (virtuellement). Une implémentation directe du flot de données produirait un circuit dans lequel toutes les opérations seraient exécutées en parallèle. Mais il peut parfois être avantageux d'exécuter des opérations en séquence. C'est le cas lorsque des opérations sont redondantes dans le modèle, il est alors possible de les *partager*. Plutôt que d'implémenter plusieurs fois ces opérations, à chaque endroit du modèle où elles sont utilisées, elles peuvent être instanciées une seule fois, puis l'accès à l'instance est séquencé. Cette approche s'appelle le *multiplexage temporel*. Il permet de réduire la consommation de ressources au prix d'une latence plus élevée. Là encore, il existe une (ou plusieurs) configuration(s) de multiplexage temporel optimale(s) vis-à-vis des spécifications.

Méthodes Formelles

Les méthodes formelles sont des techniques permettant de raisonner rigoureusement sur des systèmes. Elles reposent sur la construction d'un modèle mathématique du système étudié et sur une formalisation de propriétés attendues. Elles se déclinent en deux approches : la vérification et la synthèse.

La vérification consiste à utiliser un algorithme d'exploration des états du modèle, puis à vérifier que l'ensemble des états valident la propriété attendue.

La synthèse pose le problème inverse. Elle consiste à utiliser un algorithme d'exploration de modèles, puis à sélectionner le modèle dont les états vérifient la propriété attendue.

La principale difficulté des méthodes formelles est de déterminer un modèle suffisamment proche du système réel, pour qu'il soit capable de représenter un grand nombre de caractéristiques du système. Néanmoins, il est également important de choisir un modèle suffisamment simple, en termes de complexité algorithmique pour la vérification, pour pouvoir effectuer cette dernière avec des ressources de calcul raisonnable.

Notre contribution

Les travaux de cette thèse se concentre sur une étape d'optimisation de la synthèse de circuit logique : le multiplexage temporel. Nous proposons une approche formelle permettant d'accomplir cette tâche.

En se basant sur le modèle Réseau de Petri Temporisé (TPN) introduit par Ramchandani [Ram74], nous construisons un nouveau modèle pour les circuits synchrones, appelé Réseau de Petri Temporisé avec reset et transitions retardables (RTPN). Ce der-

nier permet de capturer les propriétés de précédence des opérations du circuit ainsi que leur temps de calcul. Une action appelée reset permet de modéliser le placement de registres et ses conséquences sur le comportement temporel du circuit.

Ce modèle a été créé pour répondre au problème de synthèse de pipeline, c'est-à-dire un placement de registres permettant de paralléliser des portions du circuit. En nous basant sur ce dernier, nous proposons une nouvelle solution au problème de synthèse de pipeline minimisant les bascules (registres de 1 bit) consommées. Puis à partir du même procédé nous sommes capable de faire du multiplexage temporel. Nous montrons en effet, que le problème du multiplexage temporel peut être réduit à la conception d'un pipeline particulier. Nous synthétisons alors le pipeline à partir du modèle et d'une spécification de logique temporelle linéaire (LTL) qui formalise la propriété de multiplexage temporel.

Nous souhaitons également étudier les propriétés de ce modèle RTPN. Nous comparons son expressivité à celle d'autres modèles formels avec du temps explicite (à intervalle ou singleton). Nous étudions également la complexité des problèmes d'accessibilité et de vérification des propriétés TCTL. Notre objectif est de fournir une implémentation fonctionnelle de l'exploration de l'espace d'états du modèle. Nous proposons alors une sémantique symbolique du modèle, ainsi qu'un algorithme d'exploration de l'espace d'états symboliques.

Enfin, pour répondre aux besoins de Renault, nous construisons un outil HLS. Ce dernier est un compilateur de Simulink vers du VHDL. Il implémente entre autres notre solution au problème de multiplexage temporel.

Organisation du manuscrit

Le manuscrit est constitué de 5 chapitres dont un chapitre de définitions préliminaires.

Le Chapitre 1 présente un prérequis des définitions utiles tout au long du manuscrit. L'idée est de s'y référer, si besoin, au cours de la lecture du reste du manuscrit.

Le Chapitre 2 introduit le modèle RTPN. Nous commençons par décrire les choix qui nous ont amené à construire ce modèle. Puis, nous définissons sa syntaxe et sa sémantique. Enfin, nous proposons un modèle plus expressif et plus simple (avec moins d'attribut).

Dans le Chapitre 3, nous étudions les propriétés du modèle. Nous nous intéressons à son expressivité et nous la comparons à celle d'autres modèles avec du temps. Nous proposons une sémantique symbolique ainsi qu'un algorithme d'exploration de l'espace d'états symboliques. Enfin, nous étudions la complexité de plusieurs problèmes classiques

de la littérature sur notre modèle.

Le Chapitre 4 définit les problèmes de synthèse de circuit que nous souhaitons résoudre. Il décrit comment nous modélisons un circuit synchrone avec un RTPN. Puis, il présente notre solution au problème de synthèse de pipeline et la compare à d'autres solutions existantes. Le même procédé est ensuite utilisé pour résoudre le problème de multiplexage temporel.

Enfin, dans le Chapitre 5, nous présentons l'outil qui a été intégralement développé dans le cadre de cette thèse. Nous proposons alors une partie expérimentale dans laquelle notre outil est utilisé pour synthétiser un circuit sur une cible FPGA.

Chapitre 1

Prérequis

Ce chapitre présente des notions qui seront utilisées dans le reste du manuscrit. Il se divise en deux parties.

La première se concentre sur les méthodes formelles. Elle définit les notations mathématiques ainsi que quelques objets mathématiques qui seront utiles par la suite : les Systèmes de Transition, les bisimulations temporelles ainsi que certaines logiques temporelles.

La seconde introduit des notions de conception de circuits. Elle définit les représentations usuelles des nombres réels en arithmétique des ordinateurs, ainsi que des notions sur les circuits logiques programmables.

Il peut être intéressant de se réferer à ce chapitre lors de la lecture du manuscrit pour obtenir des définitions ou du vocabulaire.

1.1 Méthodes formelles

1.1.1 Ensembles

$\mathbb{N}$ et $\mathbb{N}_*$ sont respectivement les ensembles des entiers naturels et des entiers naturels non nuls. $\mathbb{R}_+$ et $\mathbb{R}_{+*}$ sont respectivement les ensembles des réels positifs ou nuls et réels strictement positifs. $\mathbb{B} = \{\bot, \top\}$ est le domaine booléen.

Les opérateurs usuels $+, -, <, \leq, >, \geq$ et $=$ sont appliqués sur les vecteurs de dimension n dans $\mathbb{N}^n$, $\mathbb{R}_+^n$ et $\mathbb{R}_{+*}^n$, et sont les extensions point par point de leurs homologues dans $\mathbb{N}$, $\mathbb{N}_*$, $\mathbb{R}_+$ et $\mathbb{R}_{+*}$. Les opérateurs logiques usuels $\wedge$, $\vee$ et $\neg$ sont appliqués aux éléments de $\mathbb{B}$.

$\bar{\mathbf{0}}$ est le vecteur nul de dimension n.

Pour tout ensemble dénombrable E, le cardinal de cet ensemble est noté $|E|$.

L'union disjointe de deux ensembles E_1 et E_2 tels que $E_1 \cap E_2 = \emptyset$ est noté $E_1 \uplus E_2$.

1.1.2 Système de Transition

Pour définir la sémantique des modèles formels que nous utiliserons par la suite, nous nous appuyons sur les Systèmes de Transition.

Définition 1 (TS)**.** *Un Système de Transition sur un ensemble d'actions $\mathcal{A}$ est un n-uplet $\mathcal{S} = (Q, q_0, \mathcal{A}, \rightarrow)$ où :*

- *Q est l'ensemble des états,*
- *$q_0 \in Q$ est l'état initial,*
- *$\mathcal{A}$ est l'ensemble d'actions,*
- *$\rightarrow \subseteq Q \times \mathcal{A} \times Q$ est l'ensemble des transitions.*

Si $(q, a, q') \in \rightarrow$, on note alors également $q \xrightarrow{a} q'$.

Un Système de Transition Temporisé (TTS) peut-être vu comme un TS dans lequel deux transitions sont possibles : les transitions *d'actions* et les transitions *temporelles*.

Définition 2 (TTS)**.** *Un Système de Transition Temporisé sur un ensemble d'actions $\mathcal{A}$ est un n-uplet $\mathcal{S} = (Q, q_0, \mathcal{A}, \rightarrow)$ où :*

- *Q est l'ensemble des états,*
- *$q_0 \in Q$ est l'état initial,*
- *$\mathcal{A}$ est l'ensemble d'actions disjoint de $\mathbb{R}_{+*}$,*
- *$\rightarrow \subseteq Q \times (\mathcal{A} \cup \mathbb{R}_{+*}) \times Q$ est l'ensemble des transitions composé :*
  - *de transitions* d'actions $\xrightarrow{a \in \mathcal{A}} \subseteq Q \times \mathcal{A} \times Q$ *et*
  - *de transitions* temporelles $\xrightarrow{d \in \mathbb{R}_{+*}} \subseteq Q \times \mathbb{R}_{+*} \times Q$.

Si $(q, e, q') \in \rightarrow$, on note alors $q \xrightarrow{e} q'$.

On parle de sémantique *en temps discret* lorsque les transitions temporelles prennent leurs valeurs dans $\mathbb{N}_*$.

Une exécution d'un TTS (ou d'un TS) $\mathcal{S} = (Q, q_0, \mathcal{A}, \rightarrow)$ est une suite possiblement infinie $\rho = q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \dots$, telle que pour tout i, $q_i \xrightarrow{\alpha_i} q_{i+1}$. Soit une exécution $\rho = q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} q_{n+1}$, on note également $\rho = q_1 \xrightarrow{\alpha_1 \alpha_2 \dots \alpha_n} q_{n+1}$.

Un mot temporisé d'un TTS est une suite possiblement infinie $w = (d_1, a_1)(d_1 + d_2, a_2) \dots (\sum_{i=1\dots n} d_i, a_i)$ tel que $\rho = q_0 \xrightarrow{d_1} q_{d_1} \xrightarrow{a_1} q_{a_1} \dots \xrightarrow{d_n} q_{d_n} \xrightarrow{a_n} q_{a_n}$ est une exécution du TTS. Le langage temporisé reconnu par un TTS est l'ensemble des mots temporisés qu'il reconnaît.

Un graphe d'états d'un TTS (ou d'un TS) $\mathcal{S} = (Q, q_0, \mathcal{A}, \rightarrow)$ est un graphe dirigé $SG(\mathcal{S}) = (V, E)$ dans lequel les nœuds sont $V = Q$ et les arêtes sont $E = \{(q, \alpha, q') \mid q \xrightarrow{\alpha} q'\}$.

Les TTS et TS permettent de décrire les sémantiques de modèles plus abstrait. Par abus de langage, on parlera d'une exécution (resp. mot, langage, graphe d'état) d'un modèle pour désigner une exécution (resp. mot, langage, graphe d'état) de sa sémantique.

1.1.3 Bisimulation

On se réfère à la Bisimulation Temporelle Forte définie dans [B+05] :

Définition 3 (Simulation (temporelle) Forte)**.** *Soit deux TTSs* $\mathcal{S}_1 = (Q_1, q_0^1, \mathcal{A}, \rightarrow_1)$ *et* $\mathcal{S}_2 = (Q_2, q_0^2, \mathcal{A}, \rightarrow_2)$ *et* $\precsim$ *une relation sur* $Q_1 \times Q_2$*. On note* $s \precsim s'$ *lorsque* $(s, s') \in \precsim$*.* $\precsim$ *est une simulation (temporelle) forte de* $\mathcal{S}_1$ *par* $\mathcal{S}_2$ *ssi :*

- $\forall s_1 \in Q_1$, $\exists s_2 \in Q_2$ *tel que* $s_1 \precsim s_2$,
- *Si* $s_1 \xrightarrow{d}_1 s_1'$ *avec* $d \in \mathbb{R}_{+*}$ *alors* $s_2 \xrightarrow{d}_2 s_2'$ *et* $s_1' \precsim s_2'$,
- *Si* $s_1 \xrightarrow{a}_1 s_1'$ *avec* $a \in \mathcal{A}$ *alors* $s_2 \xrightarrow{a}_2 s_2'$ *et* $s_1' \precsim s_2'$,

Un TTS $\mathcal{S}_2$ simule fortement $\mathcal{S}_1$ *s'il existe une simulation (temporelle) forte de* $\mathcal{S}_1$ *par* $\mathcal{S}_2$*. On note* $\mathcal{S}_1 \precsim \mathcal{S}_2$*.*

Définition 4 (Bisimulation (temporelle) Forte)**.** *Soit deux TTSs* $\mathcal{S}_1 = (Q_1, q_0^1, \mathcal{A}, \rightarrow_1)$ *et* $\mathcal{S}_2 = (Q_2, q_0^2, \mathcal{A}, \rightarrow_2)$ *et* $\precsim$ *une relation sur* $Q_1 \times Q_2$*. On note* $\succsim$ *la relation sur* $Q_2 \times Q_1$ *telle que* $s_1 \precsim s_2 \iff s_2 \succsim s_1$*.* $\precsim$ *est une bisimulation (temporelle) forte de* $\mathcal{S}_1$ *par* $\mathcal{S}_2$ *ssi :*

- $\precsim$ *est une simulation (temporelle) forte de* $\mathcal{S}_1$ *par* $\mathcal{S}_2$,
- $\succsim$ *est une simulation (temporelle) forte de* $\mathcal{S}_2$ *par* $\mathcal{S}_1$*.*

On utilisera alors une relation symétrique $\sim$ *plutôt que* $\precsim$ *et* $\succsim$*.* $\mathcal{S}_1$ *et* $\mathcal{S}_2$ *sont* fortement bisimilaires *s'il existe une bisimulation (temporelle) forte entre* $\mathcal{S}_1$ *et* $\mathcal{S}_2$*. On note* $\mathcal{S}_1 \sim \mathcal{S}_2$*.*

Soit $\mathcal{A}$ un ensemble d'action (ou alphabet). On dénote $\mathcal{A}_\varepsilon = \mathcal{A} \cup \{\varepsilon\}$ avec $\varepsilon \notin \mathcal{A}$, où ε est une action *invisible* (mot vide).

Soit un TTS $\mathcal{S} = (Q, q_0, \mathcal{A}_\varepsilon, \rightarrow)$. On définit le TTS ε-abstrait $\mathcal{S}^\varepsilon = (Q, q_0, \mathcal{A}, \rightarrow_\varepsilon)$ par :

- $q \xrightarrow{d}_\varepsilon q'$ avec $d \in \mathbb{R}_{+*}$ ssi il existe une exécution $\rho = q \xrightarrow{*} q'$ de durée d avec uniquement des actions ε,
- $q \xrightarrow{a}_\varepsilon q'$ avec $a \in \mathcal{A}$ ssi il existe une exécution $\rho = q \xrightarrow{*} q'$ de durée nulle avec une unique action visible a.

Définition 5 (Bisimulation (temporelle) Faible)**.** *Soit deux TTSs* $\mathcal{S}_1 = (Q_1, q_0^1, \mathcal{A}_\varepsilon, \rightarrow_1)$ *et* $\mathcal{S}_2 = (Q_2, q_0^2, \mathcal{A}_\varepsilon, \rightarrow_2)$*.* $\mathcal{S}_1$ *et* $\mathcal{S}_2$ *sont* faiblement bisimilaire *ssi il existe une bisimulation (temporelle) forte entre leur* ε*-abstraction respective* $\mathcal{S}_1^\varepsilon$ *et* $\mathcal{S}_2^\varepsilon$*. On note* $\mathcal{S}_1 \sim_\varepsilon \mathcal{S}_2$*.*

Les définitions précédentes s'appliquent également pour des sémantiques en temps discret.

1.1.4 Logiques temporelles

Des logiques temporelles ont été introduites pour décrire des comportements temporels des modèles formels. Les logiques Linear Temporal Logic (LTL) et Computation Tree Logic (CTL) permettent de décrire des propriétés sur le temps logique.

LTL décrit des propriétés sur les exécutions. Elle a été introduite dans [Pnu77], nous rappelons ici uniquement sa syntaxe :

Définition 6 (Syntaxe LTL)**.** *Soit un ensemble de variables propositionnelles AP. L'ensemble des formules LTL sur AP est défini de manière inductive :*

- *si $p \in AP$ alors p est une formule LTL ;*
- *si φ et ψ sont des formules LTL alors $\neg\varphi$, $\varphi \vee \psi$, $\boldsymbol{X}\psi$ et $\varphi \boldsymbol{U} \psi$ sont des formules LTL.*

$\boldsymbol{X}$ et $\boldsymbol{U}$ se lisent respectivement « Next » et « Until ».

Des opérateurs additionels peuvent être construit à partir de ces opérateurs fondamentaux. Sans être exhaustif, nous utiliserons deux opérateurs logiques additionnels $\wedge$ (conjonction) et $\Rightarrow$ (implication), ainsi qu'un opérateur temporel additionnel $\mathbf{G}$ qui se lit « Globally » (toujours).

La logique CTL introduite par [CES86], décrit des propriétés sur les arbres d'exécutions. Enfin, une extension de la logique CTL a été proposé par [AD94], la logique Timed Computation Tree Logic (TCTL), qui permet de décrire des propriétés sur le temps dense. Nous ne rappelons pas leurs définitions ici.

1.2 Conception de circuits

1.2.1 Formats de données

En arithmétique des ordinateurs, il existe deux représentations principales des nombres réels : la représentation en virgule fixe et la représentation en virgule flottante.

La représentation en virgule fixe attribue un nombre de bits à la partie entière et un nombre de bits à la partie décimale.

Définition 7 (Représentation en virgule fixe)**.** *La représentation en virgule fixe est en fait une représentation entière avec un facteur de mise à l'échelle fixe. En base 2, une représentation par le nombre entier k, avec un facteur de mise à l'échelle s, encode le nombre $k * 2^{-s}$.*

On nomme la puissance du bit de poids fort Most-Significant Bit (MSB) et celle du bit de poids faible Least-Significant Bit (LSB). Une représentation en virgule fixe sur n bits, avec un facteur de mise à l'échelle s, a un MSB de $n - s - 1$ et un LSB de $-s$.

Pour encoder un réel négatif, on utilisera une représentation signée comme pour la représentation d'entier relatif (généralement le complément à deux).

La représentation en virgule flottante n'a pas un nombre de bits fixes pour la partie entière et la partie décimale.

Définition 8 (Représentation en virgule flottante)**.** *La représentation en virgule flottante est composée de trois valeurs : le signe s (égal à -1 ou 1), la mantisse m et l'exposant e. En base 2, cette représentation encode le nombre $s * m * 2^e$.*

Deux formats, fixés par la norme IEEE 754, sont généralement utilisés :

— simple précision : sur 32 bits, avec 1 bit pour le signe, 23 bits pour la mantisse et 8 bits pour l'exposant.

— double précision : sur 64 bits, avec 1 bit pour le signe, 52 bits pour la mantisse et 11 bits pour l'exposant.

1.2.2 Circuit logique programmable

Les circuits logiques sont des composants électroniques permettant de réaliser des fonctions logiques. Il existe plusieurs langages permettant de décrire ces circuits, les plus connus sont VHDL et Verilog. Une description dans un de ces langages nécessite ensuite un outil de synthèse pour obtenir un véritable circuit logique (un ensemble de portes logiques, placées et routées). Un tel outil est appelé Electronic Design Automation (EDA).

Une fois la conception d'un circuit logique terminée, ce dernier est généralement fondu, c'est-à-dire qu'il est implémenté dans un circuit définitif. Un tel circuit s'appelle ASIC.

Toutefois, il existe des circuits logiques appelés (par abus) « programmables ». Ce sont des composants permettant d'implémenter des circuits logiques de manière non définitive. Parmi ces composants se trouve la famille des FPGAs.

Pour parvenir à construire un circuit logique reconfigurable, les FPGA utilisent majoritairement deux éléments :

— Look-Up Table (LUT) : des tables de vérités implémentées par de la mémoire (souvent de la SRAM ou de la flash) ;
— Flip-Flop (FF) : des bascules D.

D'autres éléments sont également nécessaires pour effectuer le routage et d'autres éléments sont ajoutés pour améliorer l'efficacité des FPGAs (par exemple les blocs DSP), mais nous ne rentrerons pas dans ces détails.

Pour programmer un FPGA, on utilise également VHDL ou Verilog. L'outil de synthèse (EDA) approprié synthétise alors les valeurs des LUTs et le routage.

Chapitre 2

Un modèle formel pour les circuits pipelinés

Ce chapitre présente les modèles formels sur lesquels nous avons travaillé dans cette thèse.

Une première section explique les choix qui nous ont amenés à utiliser un modèle basé sur les Réseaux de Petri Temporisés (TPNs). La Section 2.2 donne une définition formelle des Réseaux de Petri (PNs) avec deux sémantiques de tir différentes : la sémantique d'entrelacement et la sémantique de pas maximaux. La Section 2.3 définie formellement les TPNs avec la sémantique de tir en trois phases de Ramchandani et Popova, et la compare avec une nouvelle sémantique que nous introduisons. La Section 2.4 présente notre extension des TPNs permettant de modéliser un circuit synchrone avec pipeline. Cette nouvelle classe s'appelle Réseau de Petri Temporisé avec reset et transitions retardables (RTPN). Le R de l'acronyme fait référence à l'action reset qui est ajoutée. Enfin la Section 2.5 décrit une classe de modèle plus expressive que les RTPNs, appelée Réseau de Petri Temporisé avec transitions retardables (DTPN). Le D de l'acronyme fait référence aux transitions *delayables* (*retardables* en français) qui sont ajoutées.

2.1 Modéliser un circuit synchrone

Un circuit synchrone peut être perçu comme un ensemble d'*opérations*, qui sont réalisées *en parallèle* ou bien *en séquence*. Chaque opération possède une certaine *durée* d'exécution dépendant de sa complexité. Il est possible de construire plusieurs configurations de circuits ayant le même *comportement fonctionnel*, c'est-à-dire fournissant les mêmes sorties pour une même séquence d'entrées. Ces configurations sont obtenues en jouant sur l'exécution parallèle ou séquentielle des opérations, nous y reviendrons dans le Chapitre 4. Deux propriétés vont imposer les possibilités de parallélisation et séquentialisation : la précédence des opérations, c'est-à-dire le fait qu'une opération a besoin des

résultats d'une autre opération, et les durées d'exécution des opérations. La modélisation des différentes configurations d'un circuit synchrone nécessite donc un modèle capturant ces deux propriétés. La première propriété peut-être modélisée par ce qu'on appelle le *temps logique*, une abstraction du temps conservant uniquement les successions d'évènements. La deuxième propriété nous force à avoir recours aux modèles *temporisés*, dans lesquels le temps est explicite.

2.1.1 Automates temporisés

Un automate est un modèle mathématique constitué d'un ensemble d'états et de transitions permettant de passer d'un état à autre.

En 1994, Alur et Dill proposent un modèle d'automate temporisé [AD94], qui étend les automates finis avec des horloges explicites. Des contraintes sur les horloges conditionnent la possibilité de prendre les transitions (*gardes*) ainsi que la possibilité de rester dans un état (*invariants*). De plus, les transitions peuvent remettre à zéro un ensemble d'horloges.

2.1.2 Extension des Réseaux de Petri avec du temps

Un Réseau de Petri (PN) est un modèle mathématique constitué d'un ensemble de places et de transitions liées par des arcs. Les places peuvent contenir des jetons qui seront *consommés* ou *produits* par le *tir* d'une transition, en suivant la direction des arcs.

Ce modèle a été étendu avec du temps à deux reprises, dans les travaux de thèse de Ramchandani [Ram74] et de Merlin [Mer74]. Ramchandani, dans son modèle appelé Réseau de Petri Temporisé (TPN), adjoint à chaque transition une *date* déterministe, à laquelle la transition est tirée. Tandis que Merlin, dans son modèle appelé Réseau de Petri Temporel (TPN), associe chaque transition avec un *intervalle*, au cours duquel la transition peut-être tirée.

D'autre part, Sifakis a proposé un modèle dans lequel la date de tir est liée aux places (syntaxe) et l'écoulement du temps aux jetons (sémantique) plutôt qu'aux transitions. Ce dernier s'est avéré être équivalent au TPN en termes d'expressivité [Sif77]. Enfin, on peut également citer les extensions avec du temps stochastique, dans lesquels les dates des évènements sont modélisées par des variables aléatoires [Mol82].

2.1.3 Notre choix

Pour la modélisation de circuits synchrones, nous nous sommes basé sur les TPNs de Ramchandani. Ce dernier a d'ailleurs introduit ce modèle dans l'objectif d'étudier des circuits logiques.

Premièrement, les Réseaux de Petri permettent de représenter aisément les systèmes *concurrents* (les systèmes qui réalisent des opérations en parallèle), ainsi que les flux circulant dans le système (les flux de données dans le cas des circuits synchrones).

Deuxièmement, les études de circuits synchrones dans la littérature s'appuient généralement sur des durées d'exécution fixes pour les opérations. Les modèles à intervalles ou à temps stochastique pourraient permettre d'introduire des incertitudes sur ces durées d'exécution, mais au prix d'une augmentation drastique de la complexité. Nous avons donc considéré dans un premier temps, un modèle plus simple avec des durées déterministes.

2.2 Réseau de Petri

2.2.1 Définition

Un Réseau de Petri est un ensemble de places et de transitions liées par des arcs. Les places contiennent des jetons qui sont consommés ou produits par le tir des transitions, en respectant le poids des arcs entrant et sortant.

Définition 9 (PN)**.** *Un Réseau de Petri est un n-uplet* $(P, T, {}^\bullet(.), (.)^\bullet, M_0)$ *défini par :*

- $P = \{p_1, p_2, \ldots, p_m\}$ *est un ensemble non-vide de places,*
- $T = \{t_1, t_2, \ldots, t_n\}$ *est un ensemble non-vide de transitions,*
- ${}^\bullet(.) \colon T \to \mathbb{N}^P$ *est la fonction d'incidence arrière,*
- $(.)^\bullet \colon T \to \mathbb{N}^P$ *est la fonction d'incidence avant,*
- $M_0 \in \mathbb{N}^P$ *est le* marquage *initial.*

Un marquage M est un élément de $\mathbb{N}^P$ tel que $\forall p \in P$, $M(p)$ est le nombre de jetons dans la place p. Un marquage M *sensibilise* une transition $t \in T$ si $M \geq {}^\bullet t$. L'ensemble des transitions sensibilisées par un marquage M est $\mathsf{enab}(M) = \{t \in T \mid M \geq {}^\bullet t\}$. On dit que deux transitions t_1 et t_2 sont *en conflit* si elles sont sensibilisées une même place, c'est-à-dire qu'il existe p tel que ${}^\bullet t_1(p) \neq 0$ et ${}^\bullet t_2(p) \neq 0$. À partir d'un marquage M, le tir d'une transition $t \in T$ conduit à un marquage $M' = M + t^\bullet - {}^\bullet t$.

Représentation graphique Un PN est représenté par un graphe biparti orienté, dont les sommets sont les places et les transitions, et les arcs sont les fonctions d'incidences. Habituellement, les places sont représentées par des ronds, et les transitions par des rectangles, et les jetons par des points à l'intérieur des places.

Un exemple de Réseau de Petri $\mathcal{N}_1 = (P, T, {}^\bullet(.), (.)^\bullet, M_0)$ est représenté dans la Figure 2.1. Dans ce dernier, on a $P = \{p_0, p_1, p_2, p_3, p_4, p_5, p_6\}$, $T = \{t_0, t_1, t_2, t_3\}$ et $M_0 = (1, 0, 0, 1, 0, 0, 0)$. Pour simplifier la lecture, dans toute la suite du manuscrit, les marquages seront notés par la collection des places marquées (contenant un jeton), avec plusieurs occurrences si la place contient plusieurs jetons. Pour ce qui est du PN $\mathcal{N}_1$, son marquage initial est donc noté $M_0 = \{p_0, p_3\}$.

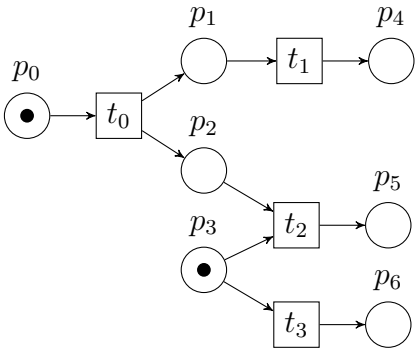


FIGURE 2.1 – Exemple de Réseau de Petri $\mathcal{N}_1$

2.2.2 Sémantique d'entrelacement

La sémantique classique des Réseaux de Petri est la sémantique d'entrelacement, dans laquelle les transitions sont tirées l'une après l'autre. Elle est définie formellement par un Système de Transition :

Définition 10 (Sémantique d'entrelacement d'un PN)**.** *La sémantique d'entrelacement d'un PN* $(P, T, {}^\bullet(.), (.)^\bullet, M_0)$ *est définie par le Système de Transition* $\mathcal{S} = (Q, q_0, T, \rightarrow)$ *tel que* $Q = \mathbb{N}^P$ *est l'ensemble des états,* $q_0 = M_0$ *est l'état initial, et* $\rightarrow \in Q \times T \times Q$ *est la relation de transition défini par :*

$$\forall t \in \mathsf{enab}(M),\ M \xrightarrow{t} M' \text{ ssi } M' = M + t^\bullet - {}^\bullet t$$

Le graphe d'états du PN $\mathcal{N}_1$ obtenu à partir de la sémantique d'entrelacement est présenté dans la Figure 2.2. On observe les entrelacements créés par la sémantique : par

exemple depuis l'état q_0 on peut tirer t_0 puis t_3, ou bien t_3 puis t_0, ce qui nous amène dans le même état q_5.

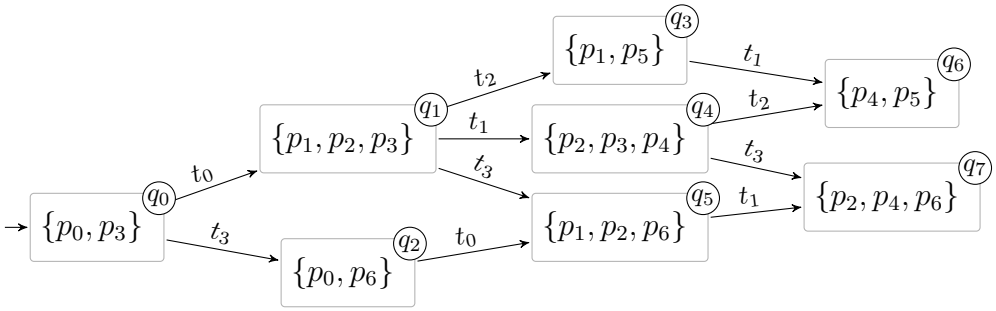


FIGURE 2.2 – Graphe d'états de $\mathcal{N}_1$ avec la sémantique d'entrelacement

2.2.3 Sémantique à pas maximaux

La sémantique à pas maximaux permet de supprimer les entrelacements (les « diamants ») des transitions qui peuvent être tirées en même temps. Dans cette dernière des ensembles maximaux de transitions, appelés *pas maximaux*, sont tirés simultanément.

Définition 11 (Pas maximal d'un PN)**.** *Soit M un marquage du PN $(P, T, {}^\bullet(.), (.)^\bullet, M_0)$, $\tau \subseteq T$ est un* pas maximal *à partir de M ssi :*

1. $\sum_{t\in\tau} {}^\bullet t \leq M$
2. $\forall t' \in T,\ ({}^\bullet t' \leq M \text{ et } t' \notin \tau) \Rightarrow \sum_{t\in\tau} {}^\bullet t + {}^\bullet t' \not\leq M$

L'ensemble des pas maximaux à partir de M est noté $\mathsf{maxstep}(M)$*.*

La première condition garantit que toutes les transitions dans le pas sont tirables, c'est-à-dire qu'elles sont sensibilisées, et qu'elles ne sont pas en conflit. La deuxième condition assure la *maximalité* du pas, elle inhibe l'existence d'un sur-ensemble de τ qui satisferait la condition précédente.

La sémantique à pas maximaux est définie formellement par un Système de Transition :

Définition 12 (Sémantique à pas maximaux d'un PN)**.** *La sémantique à pas maximaux d'un PN $(P, T, {}^\bullet(.), (.)^\bullet, M_0)$ est définie par le Système de Transition $\mathcal{S}_{max} = (Q, q_0, 2^T, \to)$ tel que $Q = \mathbb{N}^P$ est l'ensemble des états, $q_0 = M_0$ est l'état initial, et $\to \in Q \times 2^T \times Q$ est la relation de transition définie par :*

$$\forall \tau \in \mathsf{maxstep}(M),\ M \xrightarrow{\tau} M' \text{ ssi } M' = M + \Sigma_{t\in\tau}(t^\bullet - {}^\bullet t)$$

Le graphe d'états du PN $\mathcal{N}_1$ obtenu à partir de la sémantique à pas maximaux est présenté dans la Figure 2.3. On observe que l'entrelacement entre t_0 et t_3 n'est plus présent et est remplacé par un pas maximal $\{t_0, t_3\}$. On perd donc des états intermédiaires. On remarque également qu'il n'y a plus qu'un état final possible, dans cet exemple.

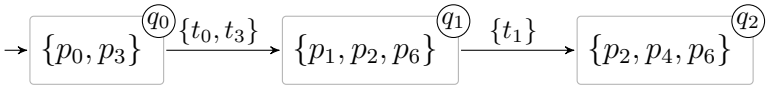


FIGURE 2.3 – Graphe d'états de $\mathcal{N}_1$ avec la sémantique à pas maximaux

2.2.4 Expressivité

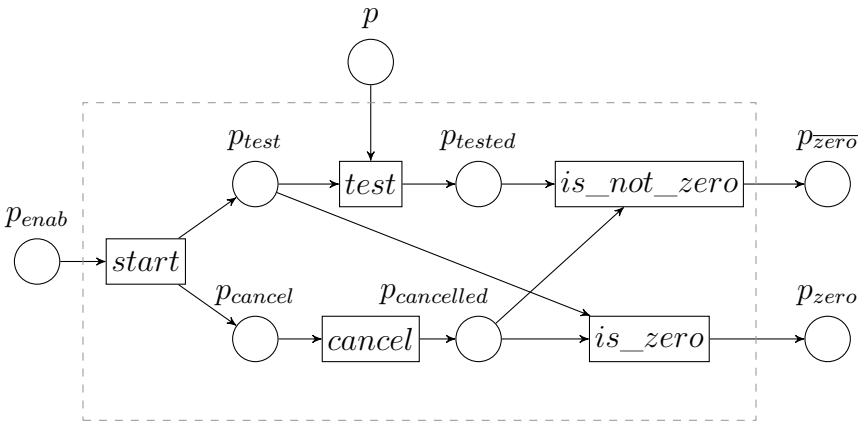
Dans le cas des graphes d'évènements (*marked graph* en anglais), c'est-à-dire les PNs où chaque place à un arc entrant et un arc sortant, les deux sémantiques sont équivalentes (mis à part les états intermédiaires), car il n'y a pas de conflit.

Dans le cas général, la sémantique à pas maximaux augmente l'expressivité des Réseaux de Petri. Popova a démontré que les PNs, avec la sémantique à pas maximaux étaient équivalents à une machine de Turing [Pop13]. Elle montre en fait l'équivalence avec les machines à compteurs.

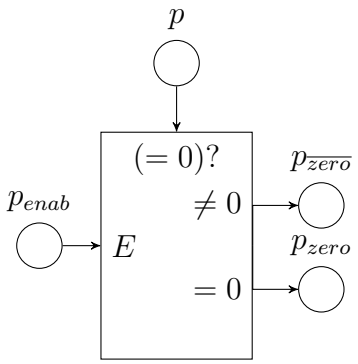
En particulier, elle construit un Réseau de Petri permettant de simuler le fameux *test à zéro*. Ce dernier est dessiné dans la Figure 2.4a. La place testée est p, et pour démarrer l'exécution du test à zéro, il faut mettre un jeton en p_{enab}. Après le tir de la transition *start*, les tirs des transitions *test* et *cancel* sont simultanés si et seulement si la place p est marquée. Le pas suivant inclura la transition is_zero si et seulement si p n'était pas marquée conduisant à un jeton dans p_{zero}. Dans la suite du manuscrit, nous utiliserons le raccourci graphique de la figure 2.4b.

2.3 Réseau de Petri Temporisé

On doit l'introduction de temps déterministe dans les Réseaux de Petri à Ramchandani [Ram74]. Il assigne à chaque transition une étiquette de temps représentant la durée que prend une action avant de s'achever. Le modèle ainsi créé s'appelle Réseau de Petri Temporisé (TPN).



(a) Motif de test à zéro



(b) Notation

FIGURE 2.4 – PN réalisant un test à zéro (avec la sémantique à pas maximaux)

2.3.1 Tir en trois phases

Ramchandani propose une sémantique de tir en trois phases :

1. les jetons d'entrée de la transition sont supprimés (*initialisation*) ;
2. un délai est écoulé jusqu'à la date de tir de la transition (*exécution*) ;
3. les jetons de sortie de la transition sont créés (*terminaison*).

Un tir ne peut être interrompu une fois commencé, ainsi la phase d'initialisation peut être assimilée à une *réservation* (en particulier lorsqu'il y a un conflit).

Dans son modèle, il utilise une sémantique d'entrelacement, c'est-à-dire que les transitions sont initialisées et terminées une par une. Les phases d'exécution des transitions sont synchronisées à une horloge globale (un axe temporel). La cohérence dynamique du réseau est assurée par une *équation d'équilibre des jetons* liant, à un instant τ, dans une place p, les jetons présents dans la place, les jetons initialement présents dans la place,

les jetons qui vont être produits par la terminaison de transitions en amont et les jetons qui vont être consommés par l'initialisation de transitions en aval. Enfin, pour éviter qu'une transition en conflit soit tirée à plusieurs reprises, les durées d'exécution nulles sont interdites.

Plus récemment, Popova a proposé une sémantique basée sur le même tir en trois phases, mais utilisant des pas maximaux [Pop13]. Un pas maximal est sélectionné, puis toutes les transitions contenues sont initialisées en une seule action. Puis les deux phases suivantes se déroulent de la même manière que dans le modèle de Ramchandani. Un des avantages d'utiliser des pas maximaux, est que l'on peut autoriser les transitions avec une durée d'exécution nulle.

2.3.2 Définition

Nous avons proposé une nouvelle sémantique des Réseaux de Petri Temporisés [PBR21c], sans réservation : l'attente est faite en amont en laissant les jetons à leur place, puis lorsque au moins une transition a écoulé son délai, elle devient tirable. Le tir d'une transition est atomique, c'est-à-dire que la consommation et la production des jetons sont réalisées en une seule action. De plus, nous utilisons une sémantique à pas maximaux, sélectionnés après l'attente. Dans notre cas les pas maximaux contiennent des transitions qui ont été sensibilisées pendant un temps égal à leur délai.

Définition 13 (TPN)**.** *Un Réseau de Petri Temporisé est un n-uplet* $(P, T, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ *défini par :*

- $(P, T, {}^\bullet(.), (.)^\bullet, M_0)$ *est un PN appelé* squelette*,*
- $\delta \colon T \to \mathbb{N}$ *est la fonction qui assigne un* délai *aux transitions.*

Une transition sensibilisée doit attendre son délai avant d'être *tirable*. Puis lorsqu'elle est tirée, si elle est encore sensibilisée, elle doit attendre de nouveau son délai avant de pouvoir tirer une fois de plus. On introduit pour cela la notion de transitions *nouvellement sensibilisées*. Une transition t' est dite *nouvellement sensibilisée* depuis un marquage M par le tir d'un ensemble de transitions τ si $M + \Sigma_{t\in\tau}(t^\bullet - {}^\bullet t)$ sensibilise t', mais $M - \Sigma_{t\in\tau}\, {}^\bullet t$ ne sensibilise t'. Si t' reste sensibilisée après son tir, alors t' est nouvellement sensibilisée. L'ensemble des transitions nouvellement sensibilisées par un ensemble de transitions τ depuis un marquage M est noté $\uparrow\mathsf{enab}(M, \tau)$.

Un état d'un Réseau de Petri Temporisé est une paire (M, v) où M est un marquage et $v \in \mathbb{R}_+^T$ est fonction donnant une valeur d'horloge aux transitions, appelée *valuation*.

Dans nos explications, on aura tendance à remplacer *la valeur d'horloge d'une transition* par *l'horloge de cette transition* lorsque la valuation est implicite (métonymie). $v(t)$ est le temps écoulé depuis que $t \in T$ est nouvellement sensibilisée. $\bar{\mathbf{0}}$ est la valuation qui assigne une valeur nulle à toutes les transitions.

Un pas maximal est maintenant défini sur les transitions tirables.

Définition 14 (Pas maximal d'un TPN)**.** *Soit $q = (M, v)$ un état du TPN $(P, T, {}^\bullet(.), (.)^\bullet, \delta, M_0)$, $\tau \subseteq T$ est un* pas maximal *à partir de q ssi :*

1. $\sum_{t\in\tau} {}^\bullet t \leq M$
2. $\forall t' \in T,\ (v(t') = \delta(t')\ et\ {}^\bullet t' \leq M\ et\ t' \notin \tau) \Rightarrow \sum_{t\in\tau} {}^\bullet t\ + {}^\bullet t' \not\leq M$
3. $\forall t \in \tau,\ v(t) = \delta(t)$

L'ensemble des pas maximaux à partir de q est noté $\mathsf{maxstep}(q)$

La nouvelle condition assure que la transition a écoulé un temps égal à son délai depuis qu'elle est nouvellement sensibilisée.

La sémantique d'un TPN est un Système de Transition Temporisé, dans lequel l'attente dans un marquage est une transition temporelle et le tir d'un pas maximal est une transition discrète.

Définition 15 (Sémantique d'un TPN)**.** *La sémantique d'un TPN $(P, T, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ est définie par le Système de Transition Temporisé $\mathcal{S} = (Q, q_0, 2^T, \to)$ tel que $Q = \mathbb{N}^P \times \mathbb{R}_+^T$ est l'ensemble des états, $q_0 = (M_0, \bar{\mathbf{0}})$ est l'état initial, et $\to \in Q \times (\mathbb{R}_{+*} \cup 2^T) \times Q$ est la relation incluant une transition temporelle et une transition discrète :*

— *La transition temporelle est définie $\forall d \in \mathbb{R}_{+*}$ par :*

$$(M, v) \xrightarrow{d} (M, v')\ ssi\ \forall t \in \mathsf{enab}(M), \begin{cases} v'(t) = v(t) + d \\ v'(t) \leq \delta(t) \end{cases}$$

— *La transition discrète est définie $\forall \tau \in \mathsf{maxstep}((M, v))$ par :*

$$(M, v) \xrightarrow{\tau} (M', v')\ ssi \begin{cases} M' = M + \sum_{t\in\tau}(t^\bullet - {}^\bullet t) \\ v'(t) = \begin{cases} 0 & si\ t \in \uparrow\mathsf{enab}(M, \tau) \\ v(t) & sinon \end{cases} \end{cases}$$

Un exemple de Réseau de Petri Temporisé dont le squelette est le PN $\mathcal{N}_1$ (Figure 2.1) est présenté dans la Figure 2.5. Les délais des transitions sont notés en dessous des transitions (en rouge).

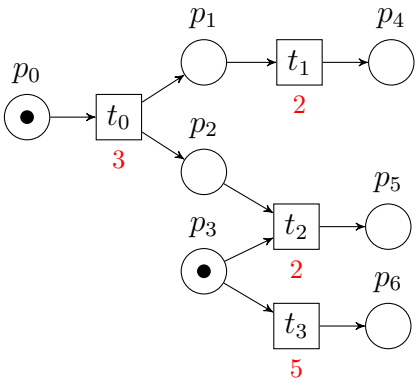


FIGURE 2.5 – Exemple de Réseau de Petri Temporisé $\mathcal{N}_2$

Une partie du graphe d'état de $\mathcal{N}_2$ est représentée dans la Figure 2.6. Il s'agit seulement d'une partie, car les transitions temporelles peuvent prendre leur délai dans $\mathbb{R}_{+*}$, il y en a donc une infinité. Pour plus de lisibilité, seulement les valeurs d'horloge des transitions sensibilisées sont représentées dans chaque état.

On remarque que t_3 conserve sa valeur dans l'état q_2, car elle n'est pas désensibilisée par le tir de t_0. Également, on peut voir que dans l'état q_3 les trois transitions t_1, t_2 et t_3 sont tirables. Les conflits sont gérés par la construction des pas maximaux.

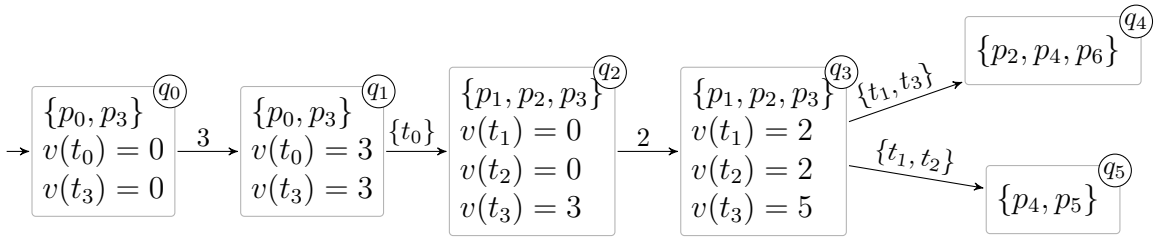


FIGURE 2.6 – Partie du graphe d'états de $\mathcal{N}_2$

2.3.3 Comparaison tir atomique / tir en trois phases

En l'absence de conflit notre sémantique atomique est identique à la sémantique de tir en trois phases avec pas maximaux : il n'y a pas d'indéterminisme.

Dans le cas général, il est possible de simuler le tir à trois phases dans notre sémantique : il suffit d'ajouter une transition avec un délai nul avant chaque transition, qui va simuler l'action de réservation.

Un exemple de Réseau de Petri Temporisé $\mathcal{N}_3$ est dessiné dans la Figure 2.7a. Une exécution obtenue avec la sémantique à trois phases est dessinée dans la Figure 2.7b, et une obtenue avec la sémantique atomique est dessinée dans la Figure 2.7c. Dans le cas de la sémantique à trois phases une transition discrète correspond à la phase d'initialisation des transitions (consommation des jetons), et les jetons sont produits lorsque l'horloge de la transition atteint son délai, après une transition temporelle. Dans le cas de notre sémantique, une transition discrète correspond à la fois à la consommation et à la production des jetons, et une transition temporelle fait seulement évoluer les valeurs d'horloge.

Dans cet exemple la sémantique à trois phases en pas maximaux ne permet pas de tirer la transition t_2, car elle ne sera sensibilisée qu'après la terminaison de t_0. Le seul pas maximal possible pour la phase de réservation est donc $\{t_0, t_1\}$.

À l'opposé, notre sémantique ne permet pas de tirer la transition t_1, car elle n'atteindra pas son délai avant que t_0 et t_2 soient tirées.

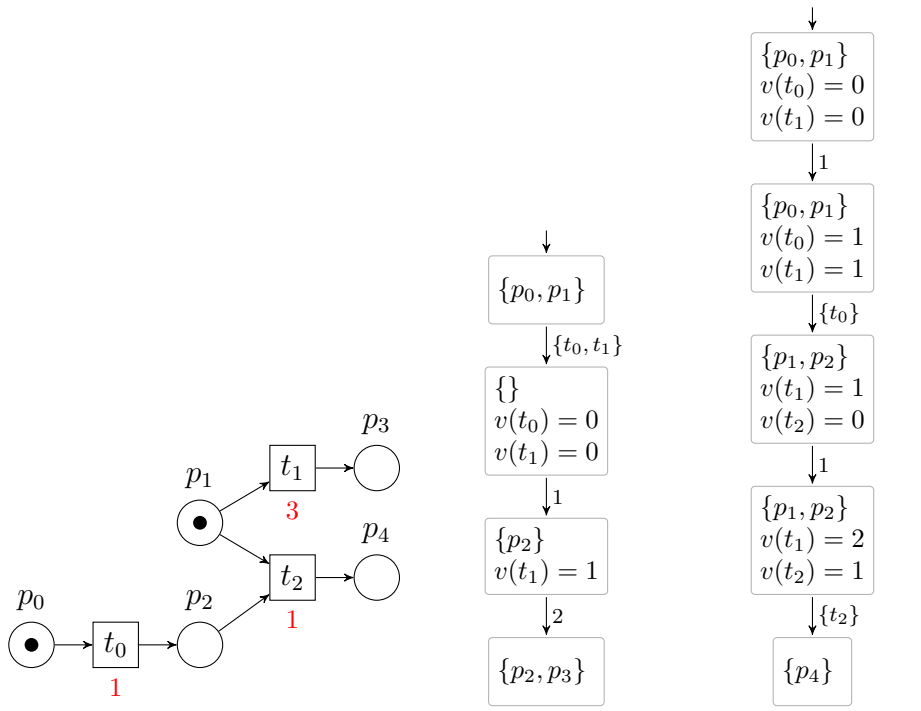
Le TPN $\mathcal{N}_4$ dessiné dans la Figure 2.8, permet de simuler avec notre sémantique atomique, le comportement du TPN $\mathcal{N}_3$ avec la sémantique à trois phases.

Ainsi notre sémantique est au moins aussi expressive que la sémantique à trois phases.

2.4 Réseau de Petri Temporisé avec reset et transitions retardables

Comme expliqué dans l'introduction notre objectif est d'explorer les différentes configurations d'un circuit synchrone, en jouant sur la parallélisation et la séquentialisation des opérations. Pour construire ces différentes configurations, on construit ce qu'on appelle un *pipeline*. Ce dernier consiste en un découpage du circuit en *étages* qui vont pouvoir être exécutés en parallèle. Pour construire un étage, on place dans le circuit des *registres* qui vont sauvegarder les valeurs des signaux pendant un cycle d'horloge. Nous souhaitons donc modéliser le placement dans le circuit de ces registres.

Nous avons étendu les TPNs avec une action particulière reset qui modélise le placement d'un étage de pipeline et des transitions *retardables* qui permettent de relaxer les contraintes sur le pipeline généré [PBR21c].



(a) Exemple de TPN $\mathcal{N}_3$

(b) Exécution de $\mathcal{N}_3$ avec le tir à 3 phases

(c) Exécution de $\mathcal{N}_3$ avec le tir atomique

FIGURE 2.7 – Comparaison des sémantiques des TPN

2.4.1 Définition

Notre extension s'appelle Réseau de Petri Temporisé avec reset et transitions retardables (RTPN). Les transitions peuvent maintenant être de deux types : soit elles sont tirées dès que possible (comme classiquement dans les TPNs), soit elles sont *retardables*, c'est-à-dire qu'elles peuvent tirer après que leur délai soit dépassé. Pour limiter les comportements infinis, on autorise les transitions retardables à tirer après leur délai, seulement si elles sont associées à une transition qui tire à son délai. De plus, les horloges peuvent être réinitialisées (l'action correspondante est appelée reset), et le délai entre deux réinitilisations est donné par un intervalle I_{reset}.

Définition 16 (RTPN)**.** *Un Réseau de Petri Temporisé avec reset et transitions retardables $\mathcal{N}$ est un n-uplet $(P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\mathsf{reset}}, M_0)$ défini par :*

— *$(P, T, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ est un TPN,*

— *$T_D \subseteq T$ est l'ensemble des transitions retardables,*

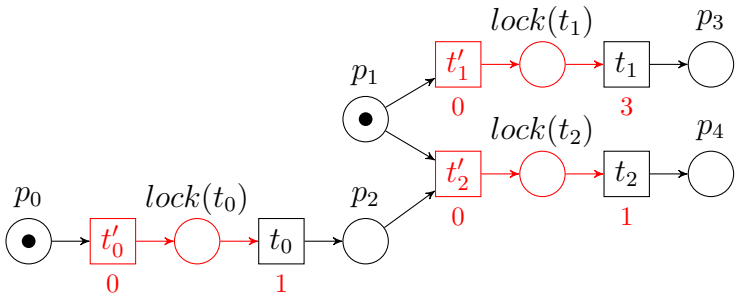


FIGURE 2.8 – TPN $\mathcal{N}_4$ simule $\mathcal{N}_3$ avec le tir à trois phases.

— I_{reset} *est l'intervalle de* reset, *dont les bornes inférieure* ($\underline{I_{\mathsf{reset}}}$) *et supérieure* ($\overline{I_{\mathsf{reset}}}$) *sont dans* $\mathbb{N}$.

Depuis un marquage M et une valuation v, une transition est tirable si elle est sensibilisée et son horloge est supérieure ou égale à son délai.

Comme pour les TPN, l'horloge des transitions non-retardables $t \notin T_D$ ne peut dépasser leur délai $\delta(t)$. Donc $v(t) \leq \delta(t)$, et t doit tirer lorsque $v(t) = \delta(t)$.

Une transition retardable $t \in T_D$ peut tirer soit lorsque $v(t) = \delta(t)$ (non retardée dans ce cas), soit lorsque $v(t) > \delta(t)$, mais dans ce cas, elle doit être associée à (au moins) une transition t' telle que $v(t') = \delta(t')$.

Un état est maintenant un n-uplet (M, v, χ) telle que $v \in \mathbb{R}_+^{T \cup \{\mathsf{reset}\}}$ est étendue avec une valeur d'horloge pour reset, mesurant le temps écoulé depuis le dernier reset, et $\chi \colon T_D \to \mathbb{B}$ est la *fonction d'activation* qui attribue une valeur booléenne à toutes les transitions retardables.

L'action reset remet à zéro toutes les horloges du réseau. Elle est possible lorsque son horloge est dans l'intervalle de reset, $v(\mathsf{reset}) \in I_{\mathsf{reset}}$.

Les pas maximaux sont dorénavant maximaux du point de vue des transitions non-retardables uniquement.

Définition 17 (Pas maximal d'un RTPN)**.** *Soit* $q = (M, v, \chi)$ *un état du RTPN* $(P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\mathsf{reset}}, M_0)$, $\tau \subseteq T$ *est un* pas maximal *à partir de* q *ssi :*

1. $\sum_{t \in \tau} {}^\bullet t \leq M$
2. $\forall t' \in T \setminus T_D,\ (v(t') = \delta(t')$ *et* ${}^\bullet t' \leq M$ *et* $t' \notin \tau) \Rightarrow \sum_{t \in \tau} {}^\bullet t + {}^\bullet t' \not\leq M$
3. $\forall t \in \tau,\ v(t) \geq \delta(t)$
4. $\exists t \in \tau$ *t.q.* $v(t) = \delta(t)$
5. $\forall t \in \tau \cap T_D,\ \chi(t) = \top$

L'ensemble des pas maximaux à partir de q est noté $\mathsf{maxstep}(q)$

La quatrième condition s'assure qu'il y a au moins une transition dans le pas qui tire à son délai. La dernière condition s'assure que toutes les transitions retardables dans le pas sont activées.

Initialement, toutes les transitions retardables sont activées. On note χ_0 la fonction d'activation initiale qui attribue à toutes les transitions dans T_D la valeur $\top$. Le tir d'un pas maximal *désactive* une transition retardable, si cette dernière ne fait pas partie du pas, alors qu'elle était tirable. Elle ne sera alors réactivée qu'à partir du prochain écoulement de temps (non nul), du prochain reset ou après une nouvelle sensibilisation. Ceci permet d'éviter certains entrelacements : lorsqu'une transition retardable est tirée directement après un pas maximal dans lequel elle aurait pu être tirée. Soit un état (M, v, χ), et soit τ un pas maximal depuis cet état. On note $\mathsf{disab}((M, v, \chi), \tau)$ l'ensemble des transitions désactivées par le tir de τ, défini par : $\mathsf{disab}((M, v, \chi), \tau) = \{t \in T_D \mid \tau \cup \{t\} \in \mathsf{maxstep}((M, v, \chi)),\ t \notin \tau \text{ et } t \notin \uparrow\mathsf{enab}(M, \tau)\}$.

La sémantique d'un RTPN est un Système de Transition Temporisé.

Définition 18 (Sémantique d'un RTPN)**.** *La sémantique d'un RTPN* $(P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\mathsf{reset}}, M_0)$ *est définie par le Système de Transition Temporisé* $\mathcal{S} = (Q, q_0, 2^T \cup \{\mathsf{reset}\}, \to)$ *tel que* $Q = \mathbb{N}^P \times \mathbb{R}_+^T \times \mathbb{B}^{T_D}$ *est l'ensemble des états,* $q_0 = (M_0, \bar{\mathbf{0}}, \chi_0)$ *est l'état initial, et* $\to \in Q \times (\mathbb{R}_{+*} \cup 2^T \cup \{\mathsf{reset}\}) \times Q$ *est la relation incluant une transition temporelle et une transition discrète :*

— *La transition temporelle est définie* $\forall d \in \mathbb{R}_{+*}$ *par :*

$$(M, v, \chi) \xrightarrow{d} (M, v', \chi') \text{ ssi } \begin{cases} \forall x \in \mathsf{enab}(M) \cup \{\mathsf{reset}\},\ v'(x) = v(x) + d \\ \forall t \in \mathsf{enab}(M) \setminus T_D,\ v'(t) \leq \delta(t) \\ v'(\mathsf{reset}) \leq \overline{I_{\mathsf{reset}}} \\ \chi' = \chi_0 \end{cases}$$

— *La transition discrète est définie par :*
  — $\forall \tau \in \mathsf{maxstep}((M, v, \chi))$ *par :*

$$(M, v, \chi) \xrightarrow{\tau} (M', v', \chi') \text{ ssi } \begin{cases} M' = M + \sum_{t \in \tau} (t^\bullet - {}^\bullet t) \\ v'(t) = \begin{cases} 0 & si\ t \in \uparrow \mathsf{enab}(M, \tau) \\ v(t) & sinon \end{cases} \\ \forall t \in T_D,\ \chi'(t) = \begin{cases} \top & si\ t \in \uparrow \mathsf{enab}(M, \tau) \\ \bot & si\ t \in \mathsf{disab}((M, v, \chi), \tau) \\ \chi(t) & sinon \end{cases} \end{cases}$$

— $(M, v, \chi) \xrightarrow{\mathsf{reset}} (M, v', \chi')$ *ssi* $\begin{cases} v(\mathsf{reset}) \in I_{\mathsf{reset}} \\ v' = \bar{\mathbf{0}} \\ \chi' = \chi_0 \end{cases}$

2.4.2 Exemple

Un exemple de RTPN est dessiné dans la Figure 2.9 et une partie de son graphe d'états est dessinée dans la Figure 2.10, limité aux occurrences des premiers resets.

Les états après un reset sont encadrés en cyan. Le reset est tirable dès lors que sa valuation est dans l'intervalle de reset, $v(\mathsf{reset}) \in I_{\mathsf{reset}}$

La transition t_0 étant retardable, elle peut être tirée à son délai (transition entre q_1 et q_2), ou bien en même temps que t_3 (transition entre q_8 et q_9). Remarquons que dans cet exemple, si t_0 n'est pas tirée est que son délai est dépassé (dans l'état q_{12}) elle ne sera plus tirable, jusqu'au prochain reset et après avoir atteint de nouveau son délai (depuis l'état q_{13}).

La fonction d'activation χ est représentée une valeur de valuation barrée lorsque la transition est désactivée. L'état q_{17} illustre l'exemple de la désactivation de la transition t_4, car elle n'a pas été tirée depuis q_5 alors qu'elle était tirable. Dans cet exemple, la fonction χ permet d'éviter l'ajout d'un arc supplémentaire menant à l'état q_6 (tir du pas maximal $\{t_4\}$ depuis q_{17}), qui est déjà existant.

2.5 Réseau de Petri Temporisé avec transitions retardables

Dans cette section, nous allons voir qu'il est possible de définir une sur-classe des RTPNs sans l'action reset. Cette dernière s'appelle Réseau de Petri Temporisé avec tran-

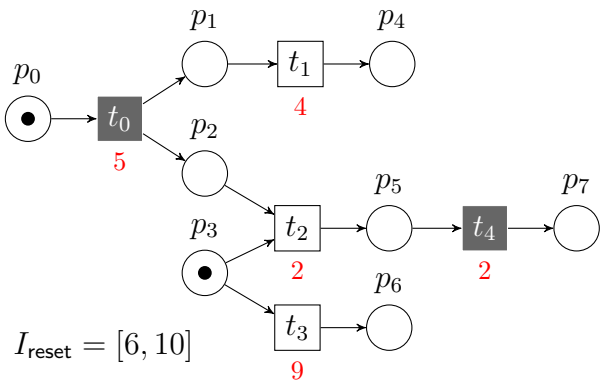


FIGURE 2.9 – Exemple de RTPN $\mathcal{N}_3$

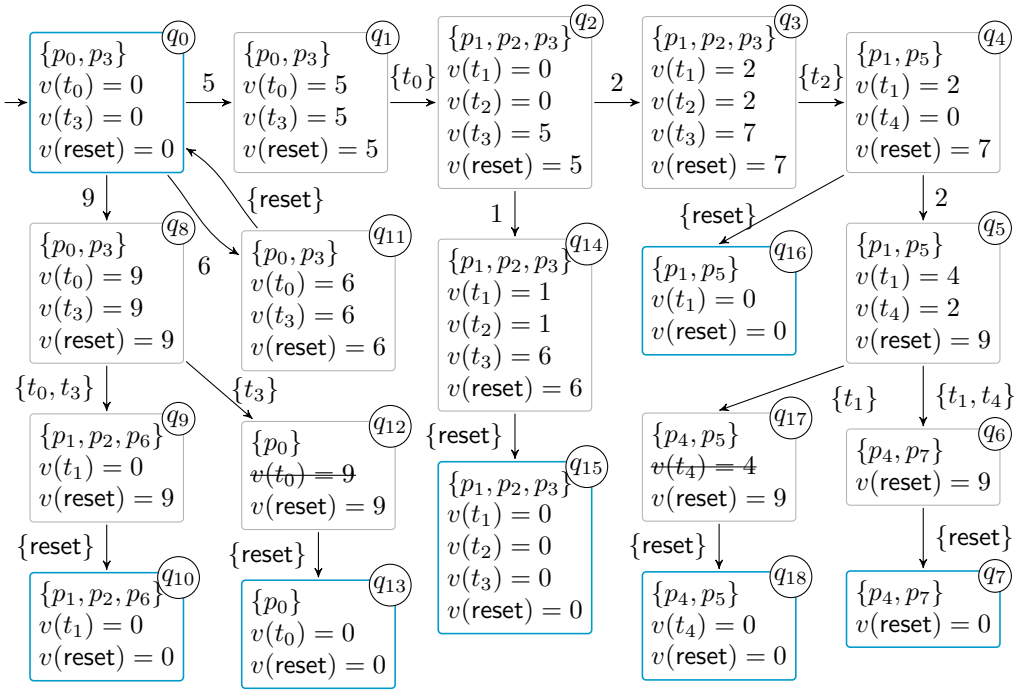


FIGURE 2.10 – Partie du graphe d'états de $\mathcal{N}_3$ (jusqu'aux premiers resets)

sitions retardables (DTPN).

2.5.1 Reset explicite

En raison de la densité du temps le reset peut être tiré depuis une infinité d'états ayant le même marquage, mais l'état successeur sera toujours le même. Ainsi, les seuls temps pertinents de tir du reset sont soit sur les bornes de son intervalle ($\underline{I_{\text{reset}}}$ et $\overline{I_{\text{reset}}}$), soit en même temps qu'un tir de transitions (dans la sémantique juste après le tir de

transitions). C'est pourquoi le reset peut-être considéré comme une transition retardable possédant une borne supérieure temporelle, et cela préserve les exécutions discrètes. Ainsi pour tout RTPN nous pouvons construire un Réseau de Petri Temporisé avec transitions retardables (sans reset) qui le simule.

En effet, il est possible d'exprimer explicitement une transition reset avec le motif présenté dans la Figure 2.11. La place p_{reset} contient un jeton depuis le dernier reset tiré. L'intervalle de reset est assuré par la transition retardable avec pour délai $\underline{I_{\mathsf{reset}}}$ et la transition non-retardable avec pour délai $\overline{I_{\mathsf{reset}}}$. Pour chaque place p_i du réseau, le motif encadré en pointillé est ajouté et lié aux transitions *reset* et *end_reset*. Ce motif réalise le reset des transitions sensibilisées par la place p_i. Il fonctionne en deux étapes : tout d'abord la place est vidée de ses jetons, ces derniers sont temporairement placés en p_i^{stock}, puis lorsque la vidange est terminée tous les jetons sont remis dans la place. Les deux étapes sont basées sur un test à zéro bouclé qui simule une *boucle tant que*. Autrement dit, les jetons sont retirés de p_i (resp. p_i^{stock}) tant qu'il y en reste. Notons qu'avec un réseau sauf (au plus un jeton par place), le motif peut être largement simplifié : il suffit en fait d'un seul test à zéro qui enlève puis remet le jeton en p_i.

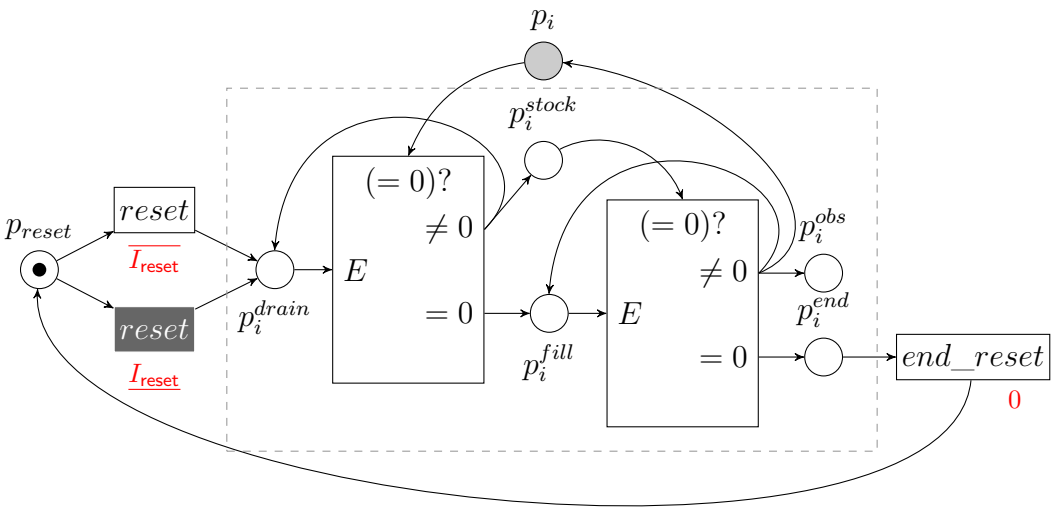


FIGURE 2.11 – Motif permettant d'exprimer le reset

Sur le motif présenté dans la Figure 2.11, la place p_i^{obs} compte le nombre total de jetons ayant subi un reset en p_i depuis l'état initial. Autrement dit, il est possible de construire un observateur qui compte le nombre de jetons ayant subi un reset par place. Ce résultat

nous sera utile par la suite.

2.5.2 Définition

Un Réseau de Petri Temporisé avec transitions retardables est donc un RTPN sans tous les objets permettant de définir le reset.

Définition 19 (DTPN)**.** *Un Réseau de Petri Temporisé avec transitions retardables $\mathcal{N}$ est un n-uplet $(P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ défini par :*

— *$(P, T, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ est un TPN,*

— *$T_D \subseteq T$ est l'ensemble des transitions retardables.*

Les états d'un DTPN sont également des n-uplets (M, v, χ). Les pas maximaux d'un DTPN sont définis de la même manière que ceux d'un RTPN (voir Définition 17).

La sémantique d'un DTPN est un Système de Transition Temporisé.

Définition 20 (Sémantique d'un DTPN)**.** *La sémantique d'un DTPN $(P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ est définie par le Système de Transition Temporisé $\mathcal{S} = (Q, q_0, 2^T, \to)$ tel que $Q = \mathbb{N}^P \times \mathbb{R}_+^T \times \mathbb{B}^{T_D}$ est l'ensemble des états, $q_0 = (M_0, \bar{\mathbf{0}}, \chi_0)$ est l'état initial, et $\to \in Q \times (\mathbb{R}_{+*} \cup 2^T) \times Q$ est la relation incluant une transition temporelle et une transition discrète :*

— *La transition temporelle est définie $\forall d \in \mathbb{R}_{+*}$ par :*

$$(M, v, \chi) \xrightarrow{d} (M, v', \chi') \text{ ssi } \begin{cases} \forall t \in \mathsf{enab}(M),\ v'(t) = v(t) + d \\ \forall t \in \mathsf{enab}(M) \setminus T_D,\ v'(t) \leq \delta(t) \\ \chi' = \chi_0 \end{cases}$$

— *La transition discrète est définie $\forall \tau \in \mathsf{maxstep}((M, v, \chi))$ par :*

$$(M, v, \chi) \xrightarrow{\tau} (M', v', \chi') \text{ ssi } \begin{cases} M' = M + \sum_{t \in \tau} (t^\bullet - {}^\bullet t) \\ v'(t) = \begin{cases} 0 & si\ t \in \uparrow \mathsf{enab}(M, \tau) \\ v(t) & sinon \end{cases} \\ \forall t \in T_D,\ \chi'(t) = \begin{cases} \top & si\ t \in \uparrow \mathsf{enab}(M, \tau) \\ \bot & si\ t \in \mathsf{disab}((M, v, \chi), \tau) \\ \chi(t) & sinon \end{cases} \end{cases}$$

Chapitre 3

Propriétés des DTPNs

Ce chapitre se concentre sur l'études des propriétés des modèles RTPN et DTPN.

Pour commencer, nous souhaitons comparer l'expressivité de notre modèle DTPN avec celle des Réseaux de Petri Temporels (TPNs) de Merlin. C'est ce que nous faisons dans la Section 3.1. Puis, dans l'objectif de produire un algorithme d'exploration des états des DTPNs, nous proposons une sémantique symbolique dans la Section 3.2. En appliquant cette sémantique symbolique à la sous-classe RTPN, nous montrons une traduction en automate temporisé à une seule horloge dans la Section 3.3. Dans la Section 3.4, nous étudions la complexité théorique de problèmes classiques pour les DTPNs (problèmes d'accessibilité et de vérification TCTL) ainsi que la complexité pratique du calcul du successeur avec notre sémantique symbolique. Enfin, la Section 3.5 rassemble les algorithmes d'exploration de l'espace d'états symboliques que nous avons implémenté dans un outil dont nous montrons l'application dans la suite du manuscrit.

3.1 Expressivité des DTPNs par rapport aux classes à intervalles

Dans cette section nous allons comparer la classe DTPN avec différentes classes de Réseau de Petri avec du temps :

- La classe Réseau de Petri Temporel (TPN) de Merlin dans laquelle un intervalle de temps dense est associé à chaque transition. Une transition sensibilisée peut tirer dès lors qu'elle a attendu une durée incluse dans son intervalle associé et elle doit tirer avant d'avoir atteint la borne supérieure. La sémantique formelle est donnée dans [BD91].
- La classe TPN avec une sémantique à pas maximaux (TPN_{ms}) dans laquelle en une étape toutes les transitions tirables, pas en conflit, sont tirées simultanément.
- La classe TPN avec une sémantique en temps discret (TPN^{dt}), comme étudié

dans [Pop91]. Une transition est associée à un intervalle comme dans le modèle TPN, mais le temps s'écoule par pas discrets (entiers).

— La classe TPN avec une sémantique en temps discret et à pas maximaux (TPN^{dt}_{ms}).

On nomme de plus la classe DTPN^{dt} la classe DTPN avec une sémantique en temps discret.

3.1.1 Définitions des différentes classes

Soit $\mathbb{I}_{\mathbb{R}_+}$ et $\mathbb{I}_{\mathbb{N}}$ les ensembles des intervalles sur $\mathbb{R}_+$ dont les bornes sont respectivement dans $\mathbb{R}_+$ et $\mathbb{N}$ ($\mathbb{I}_{\mathbb{N}} \subseteq \mathbb{I}_{\mathbb{R}_+}$). Soit $I \in \mathbb{I}_{\mathbb{R}_+}$, $\underline{I}$ dénote sa borne inférieure et $\overline{I}$ dénote sa borne supérieure si elle existe ou $+\infty$ si I n'est pas borné. Soit $d \in \mathbb{R}_+$, on note $I - d$ l'intervalle défini par $I - d = \{x - d \mid x \in I \text{ et } x \geq d\}$.

Définition 21 (TPN)**.** *Un Réseau de Petri Temporel est un n-uplet* $(P, T, {}^\bullet(.), (.)^\bullet, I_S, m_0)$ *défini par :*

— $(P, T, {}^\bullet(.), (.)^\bullet, m_0)$ *est un PN appelé* squelette*,*

— $I_S \colon T \to \mathbb{I}_{\mathbb{N}}$ *est la fonction qui assigne un* intervalle statique *aux transitions.*

Pour la sémantique des TPNs, on s'appuie sur la définition donnée par [BV03] avec un intervalle dynamique décroissant.

Un état d'un TPN est une paire (m, I) où m est un marquage et $I \colon T \mapsto \mathbb{I}_{\mathbb{R}_+}$ est une fonction d'attribution d'*intervalle dynamique*. Notons que pour les classes en temps discret, I prend ses valeurs dans $\mathbb{I}_{\mathbb{N}}$.

Une transition sensibilisée $t \in \mathsf{enab}(m)$ devient tirable lorsque la borne inférieure de son intervalle est nulle $\underline{I(t)} = 0$. Une transition doit tirer lorsque la borne supérieure de son intervalle (si elle existe) est nulle $\overline{I(t)} = 0$.

Les sémantiques d'entrelacements en temps dense et en temps discret sont des TTSs.

Définition 22 (Sémantique des classes TPN et TPN^{dt})**.** *La sémantique d'un* TPN *(resp.* TPN^{dt}*)* $(P, T, {}^\bullet(.), (.)^\bullet, I_S, m_0)$ *est définie par le Système de Transition Temporisé* $\mathcal{S} = (Q, q_0, T, \to)$ *tel que* $Q = \mathbb{N}^P \times \mathbb{I}^T_{\mathbb{R}_+}$ *est l'ensemble des états,* $q_0 = (m_0, I_S)$ *est l'état initial, et* $\to \in Q \times (\mathbb{R}_+ \cup T) \times Q$ *est la relation incluant une transition temporelle et une transition discrète :*

— *La transition temporelle est définie* $\forall d \in \mathbb{R}_+$ *pour la classe* TPN *(resp.* $\forall d \in \mathbb{N}$

pour la classe $\mathbb{T}\text{PN}^{dt}$*) par :*

$$(m, I) \xrightarrow{d} (m, I') \text{ ssi } \forall t \in \mathsf{enab}(m), \begin{cases} I'(t) = I(t) - d \\ I'(t) \neq \emptyset \end{cases}$$

— *La transition discrète est définie* $\forall t \in \mathsf{enab}(m)$ *tel que* $\underline{I(t)} = 0$ *par :*

$$(m, I) \xrightarrow{t} (m', I') \text{ ssi } \begin{cases} m' = m + t^\bullet - {}^\bullet t \\ I'(t) = \begin{cases} I_S(t) & \text{si } t \in \uparrow \mathsf{enab}(m, \{t\}) \\ I(t) & \text{sinon} \end{cases} \end{cases}$$

Le pas maximal pour un $\mathbb{T}$PN est similaire à celui pour un TPN, il suffit d'adapter la définition de transitions tirables.

Définition 23 (Pas maximal d'un $\mathbb{T}$PN)**.** *Soit* $q = (m, I)$ *un état du* $\mathbb{T}$PN $(P, T, {}^\bullet(.), (.)^\bullet, I_S, m_0)$*,* $\tau \subseteq T$ *est un* pas maximal *à partir de* q *ssi :*

1. $\sum_{t \in \tau} {}^\bullet t \leq M$
2. $\forall t' \in T,\ (\underline{I(t')} = 0 \text{ et } {}^\bullet t' \leq m \text{ et } t' \notin \tau) \Rightarrow \sum_{t \in \tau} {}^\bullet t + {}^\bullet t' \not\leq m$
3. $\forall t \in \tau,\ \underline{I(t)} = 0$

L'ensemble des pas maximaux à partir de q *est noté* $\mathsf{maxstep}(q)$

Les sémantiques à pas maximaux en temps dense et en temps discret sont également des TTSs.

Définition 24 (Sémantique des classes $\mathbb{T}\text{PN}_{ms}$ et $\mathbb{T}\text{PN}^{dt}_{ms}$)**.** *La sémantique d'un* $\mathbb{T}\text{PN}_{ms}$ *(resp.* $\mathbb{T}\text{PN}^{dt}_{ms}$*)* $(P, T, {}^\bullet(.), (.)^\bullet, I_S, m_0)$ *est définie par le Système de Transition Temporisé* $\mathcal{S} = (Q, q_0, 2^T, \rightarrow)$ *tel que* $Q = \mathbb{N}^P \times \mathbb{I}_{\mathbb{R}_+}^T$ *est l'ensemble des états,* $q_0 = (m_0, I_S)$ *est l'état initial, et* $\rightarrow \in Q \times (\mathbb{R}_+ \cup 2^T) \times Q$ *est la relation incluant une transition temporelle et une transition discrète :*

— *La transition temporelle est définie* $\forall d \in \mathbb{R}_+$ *pour la classe* $\mathbb{T}\text{PN}_{ms}$ *(resp.* $\forall d \in \mathbb{N}$ *pour la classe* $\mathbb{T}\text{PN}^{dt}_{ms}$*) par :*

$$(m, I) \xrightarrow{d} (m, I') \text{ ssi } \forall t \in \mathsf{enab}(m), \begin{cases} I'(t) = I(t) - d \\ I'(t) \neq \emptyset \end{cases}$$

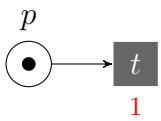


FIGURE 3.1 – Un DTPN (ou DTPN^{dt}) $\mathcal{N}_1$

— *La transition discrète est définie* $\forall \tau \in \mathsf{maxstep}((m, I))$ *par :*

$$(m, I) \xrightarrow{\tau} (m', I') \text{ ssi } \begin{cases} m' = m + \sum_{t \in \tau}(t^\bullet - {}^\bullet t) \\ I'(t) = \begin{cases} 0 & si\ t \in \uparrow \mathsf{enab}(m, \tau) \\ I(t) & sinon \end{cases} \end{cases}$$

3.1.2 Expressivité des classes DTPN et DTPN^{dt}

Pour commencer nous montrons que les classes TPN, TPN^{dt}, TPN$_{ms}$ et TPN$^{dt}_{ms}$ bornées ne sont pas plus expressives que la classe DTPN borné, en regard de la simulation temporelle faible.

Nous rappelons le lemme énoncé par [B$\acute{+}$05] qui stipule que *attendre ne désensibilise pas de transition* dans les TPNs, et ce résultat s'applique également aux classes TPN^{dt}, TPN$_{ms}$ et TPN$^{dt}_{ms}$.

Lemme 1. *Soit* (m, I) *un état d'un* TPN. *S'il existe* (m', I') *tel que* $(m, I) \xrightarrow{t_1 t_2 \dots t_k} (m', I')$ *avec* $t_1 t_2 \dots t_k$ *une séquence de tirs instantanée (sans attente entre les tirs) et s'il existe* (m_d, I_d) *tel que* $(m, I) \xrightarrow{d} (m_d, I_d)$ *avec* $d \in \mathbb{R}_+$, *alors il existe* (m'_d, I'_d) *tel que* $(m_d, I_d) \xrightarrow{t_1 t_2 \dots t_k} (m'_d, I'_d)$.

Théorème 3.1.1. *Il n'existe pas de* TPN *et* TPN$_{ms}$ *(resp.* TPN^{dt} *et* TPN$^{dt}_{ms}$*) faiblement bisimilaire avec le DTPN (resp. DTPN^{dt})* $\mathcal{N}_1$ *de la Figure 3.1.*

Démonstration. Supposons qu'il existe un TPN $\mathcal{N}$ faiblement bisimilaire avec $\mathcal{N}_1$ et soit $\sim$ la bisimulation entre leurs sémantiques respectives $\mathcal{S}_\mathcal{N}$ et $\mathcal{S}_{\mathcal{N}_1}$. Soit (m_0, I_0) l'état initial de $\mathcal{S}_\mathcal{N}$ et $(M_0, 0)$ l'état initial de $\mathcal{S}_{\mathcal{N}_1}$. On a $(M_0, 0) \xrightarrow{1} (M_0, 1)$ et $(m_0, I_0) \xrightarrow{1}_\varepsilon (m_1, I_1)$ avec $(M_0, 0) \sim (m_0, I_0)$ et $(M_0, 1) \sim (m_1, I_1)$. Comme t peut tirer depuis $(M_0, 1)$ alors tous les états (m'_1, I'_1), accessible depuis (m_1, I_1) en un temps nul (transitions ε), peuvent tirer une séquence instantanée ne contenant que l'action t. t étant une transition retardable il doit exister un état (m'_1, I'_1) tel qu'on peut attendre $d \in \mathbb{R}_{+*}$ depuis (m'_1, I'_1) ce qui nous

amène en (m'', I'') faiblement bisimilaire avec l'état $(M_0, 1+d)$. Dans cet état $(M_0, 1+d)$ il n'est plus possible de tirer t. Or d'après le Lemme 1 il existe une séquence instantanée contenant l'action t depuis (m'', I'') ce qui est une contradiction. Le raisonnement est similaire pour les classes $\mathbb{TPN}^{dt}$, $\mathbb{TPN}_{ms}$ et $\mathbb{TPN}^{dt}_{ms}$. □

On étudie ensuite la relation dans l'autre sens.

3.1.3 Les classes en temps dense

Dans la sémantique d'un DTPN, une transition discrète ne peut avoir lieu que lorsqu'une transition a été sensibilisée pendant un temps $\delta \in \mathbb{N}$. Les transitions discrètes n'ont donc lieu qu'à des temps entiers, et nous avons le lemme suivant :

Lemme 2. *Soit un DTPN* $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ *et* $\rho = q_1 \xrightarrow{\tau_1} q_2 \xrightarrow{d} q_3 \xrightarrow{\tau_2} q_4$ *une exécution de sa sémantique* $\mathcal{S}_\mathcal{N}$*, avec* $\tau_1 \subseteq T$*,* $\tau_2 \subseteq T$ *et* $d \in \mathbb{R}_{+*}$*. Alors* $d \in \mathbb{N}_*$.

On peut ainsi établir le théorème suivant :

Théorème 3.1.2. *Les classes DTPN et* $\mathbb{TPN}$ *sont incomparables au regard de la simulation faible, et de même pour les classes DTPN et* $\mathbb{TPN}_{ms}$.

Démonstration. Dans les sémantiques des $\mathbb{TPN}$ et $\mathbb{TPN}_{ms}$ une transition peut être tirée à n'importe quelle date de l'intervalle de temps dense. Or d'après le Lemme 2 ceci ne peut pas arriver dans un DTPN. Le Théorème 3.1.1 termine la preuve. □

3.1.4 Les classes en temps discret

On s'intéresse maintenant aux classes $\mathbb{TPN}^{dt}$ et $\mathbb{TPN}^{dt}_{ms}$ qui ont une sémantique discrète.

Le graphe temporisé d'états d'un $\mathbb{TPN}^{dt}$ ou d'un $\mathbb{TPN}^{dt}_{ms}$ peut s'écrire sous la forme d'une Structure de Kripke Temporisée (DKS) stricte (en anglais *tight* DKS) [AD94 ; LMS02] dans lequel l'ensemble des actions est T pour un $\mathbb{TPN}^{dt}$ ou une partie de 2^T pour un $\mathbb{TPN}^{dt}_{ms}$. Cette DKS peut être traduit en DTPN^{dt} (avec un jeton unique) faiblement bisimilaire au réseau de départ. Nous faisons la démonstration de cette construction pour la classe $\mathbb{TPN}^{dt}$ (mais une démonstration similaire s'applique à la classe $\mathbb{TPN}^{dt}_{ms}$).

Soit un $\mathbb{TPN}^{dt}$ borné $\mathfrak{N} = (\mathfrak{P}, \mathfrak{T}, pre, post, I_S, m_0)$. Le graphe temporisé d'états de $\mathfrak{N}$ est défini comme le graphe dont les nœuds $V_\mathfrak{N}$ sont des états et dont les arêtes $E_\mathfrak{N}$ correspondent aux exécutions $q \xrightarrow{d,\mathfrak{t}} q'$ avec $d \in \mathbb{N}$ une transition temporelle (possiblement nulle)

et $\mathfrak{t} \in \mathfrak{T}$ une transition d'action de la sémantique. Autrement dit, les états atteignables par simple délai, sans tir de transition, sont ignorés. Le graphe temporisé d'états de $\mathfrak{N}$ est noté $SGT(\mathfrak{N}) = (V_{\mathfrak{N}}, E_{\mathfrak{N}})$. Ce graphe converge par égalité des états (quotient de l'arbre d'états par la relation égalité).

Notons qu'au vu de la sémantique avec intervalles dynamiques décroissants définie précédemment, l'ensemble des états de $V_{\mathfrak{N}}$ est fini. Cependant, s'il existe des intervalles ouverts dans I_S, alors il peut exister une infinité d'arêtes dans $E_{\mathfrak{N}}$. Dans un premier temps, nous supposons que tous les intervalles de I_S sont bornés, ainsi $SGT(\mathfrak{N})$ est fini.

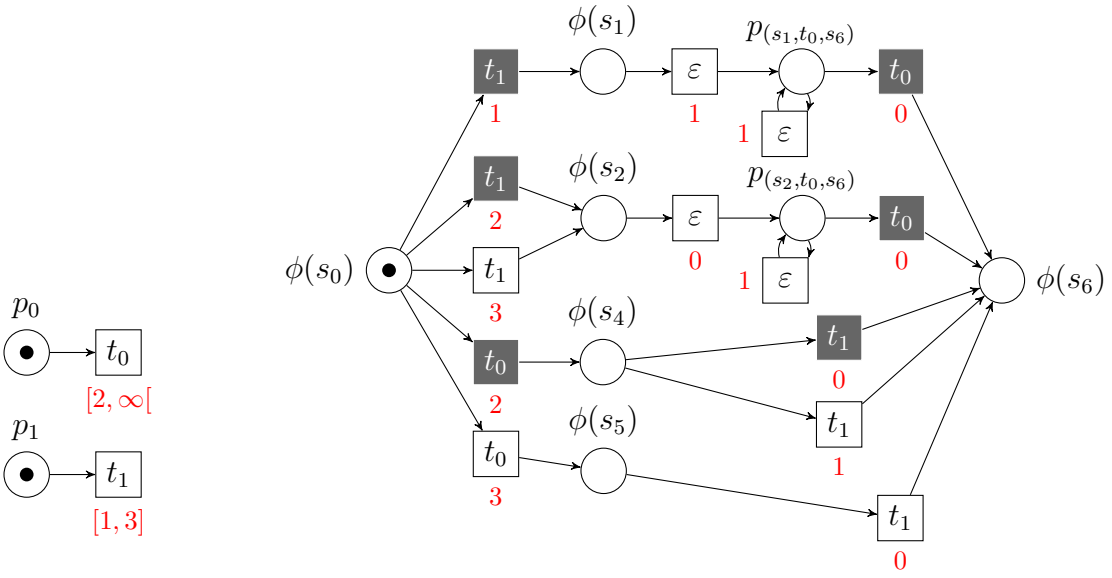
Nous construisons le DTPN à jeton unique $\mathcal{N}_{\mathfrak{N}} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ à partir de $SGT(\mathfrak{N}) = (V_{\mathfrak{N}}, E_{\mathfrak{N}})$. On lui ajoute une fonction de label λ qui associe à chaque transition t un label $\lambda(t)$ (dans l'objectif d'avoir plusieurs transitions avec le même label dans le réseau).

- $\phi \colon V_{\mathfrak{N}} \mapsto P$ est une application injective
- $P \supseteq \{\phi(s) \mid s \in V_{\mathfrak{N}}\}$, on note $p_0 = \phi((m_0, I_S))$
- $\forall p \in P$, $M_0(p) = 0$ si $p \neq p_0$, et $M_0(p_0) = 1$
- $\lambda \colon T \mapsto \mathfrak{T}$ est la fonction de label
- Pour chaque arête $(s_1, (d_1, \mathfrak{t}_1), s_2)$ de $E_{\mathfrak{N}}$, il existe une transition t dans T, telle que :
  - ${}^\bullet t(\phi(s_1)) = t^\bullet(\phi(s_2)) = 1$
  - $\delta(t) = d_1$
  - Si $d_1 < \max\limits_{(s_1,(d,\mathfrak{t}),s_2') \in E_{\mathfrak{N}}} (d)$ alors $t \in T_D$
  - $\lambda(t) = \mathfrak{t}_1$

Dans le cas où les intervalles de I_S ne sont pas bornés, il existe des paires d'états $(s, s') \in V_{\mathfrak{N}} \times V_{\mathfrak{N}}$ qui ont une infinité d'arêtes les liant (toutes les dates de tirs possibles des transitions qui ont un intervalle ouvert). On note T_∞ l'ensemble des transitions qui ont un intervalle ouvert.

Soit une paire $(s, s') \in V_{\mathfrak{N}} \times V_{\mathfrak{N}}$ telle qu'il existe une infinité d'arêtes $(s, (d, \mathfrak{t}), s') \in E_{\mathfrak{N}}$. On note $s = (m, I)$. Remarquons qu'une telle paire n'existe que si $\mathsf{enab}(m) \subseteq T_\infty$: toutes les transitions sensibilisées ont un intervalle ouvert. Pour chaque $\mathfrak{t}_i \in \mathsf{enab}(m) \cap T_\infty$, on note $(s, \mathfrak{t}_i, s') = \{(s, (d, \mathfrak{t}), s') \in E_{\mathfrak{N}} \mid \mathfrak{t} = \mathfrak{t}_i\}$ et $c_i = \min\limits_{(s,(d_{ij},\mathfrak{t}_i),s') \in (s,\mathfrak{t}_i,s')} (d_{ij})$. Pour chaque transition $\mathfrak{t}_i$, il existe une place $p_{(s,\mathfrak{t}_i,s')}$ dans P et trois transitions t_d, t et t_1 dans T, telles que :

- ${}^\bullet t_d(\phi(s)) = t_d^\bullet(p_{(s,\mathfrak{t}_i,s')}) = 1$, $\delta(t_d) = c_i$ et $\lambda(t_d) = \varepsilon$
- Si $c_i < \max\limits_{\mathfrak{t}_k \in \mathsf{enab}(m) \cap T_\infty} (c_k)$ alors $t_d \in T_D$



(a) Exemple de $\mathbb{TPN}^{dt}$ $\mathcal{N}_2$ (b) Traduction en un DTPN^{dt} $\mathcal{N}_3$

FIGURE 3.2 – Exemple de traduction faiblement bisimilaire de $\mathbb{TPN}^{dt}$ en un DTPN^{dt}

— $t \in T_D$, ${}^\bullet t(p_{(s,\mathfrak{t}_i,s')}) = t^\bullet(\phi(s')) = 1$, $\delta(t) = 0$ et $\lambda(t) = \mathfrak{t}_i$
— ${}^\bullet t_1(p_{(s,\mathfrak{t}_i,s')}) = t_1^\bullet(p_{(s,\mathfrak{t}_i,s')}) = 1$, $\delta(t_1) = 1$ et $\lambda(t_1) = \varepsilon$

Autrement dit, on ajoute une place $p_{(s,\mathfrak{t}_i,s')}$ depuis laquelle on pourra attendre indéfiniment, et on pourra tirer à tout moment la transition $\mathfrak{t}_i$.

Une illustration de cette traduction est donnée sur l'exemple du $\mathbb{TPN}^{dt}$ $\mathcal{N}_2$ traduit en DTPN^{dt} $\mathcal{N}_3$, dans la Figure 3.2. Les transitions t sont directement représentées avec leur label associé $\lambda(t)$. Dans cet exemple, deux cas de figure même à un état d'attente. Soit t_1 est tirée après une unité de temps, auquel cas il faut encore attendre au moins une unité de temps pour pouvoir tirer t_0, ce qui nous amène dans l'état d'attente représenté par la place $p_{(s_1,t_0,s_6)}$. Soit t_1 est tirée après (au moins) 2 unités de temps, auquel cas t_0 peut être tirée directement, ce qui correspond à l'état d'attente modélisé par la place $p_{(s_2,t_0,s_6)}$.

Théorème 3.1.3. *La classe $DTPN^{dt}$ est strictement plus expressive que les classes $\mathbb{TPN}^{dt}$ et $\mathbb{TPN}^{dt}_{ms}$ au regard de la simulation faible.*

Démonstration. La traduction précédente montre que tout $\mathbb{TPN}^{dt}$ et tout $\mathbb{TPN}^{dt}_{ms}$ peut être traduit par un DTPN^{dt} faiblement bisimilaire. Le Théorème 3.1.1 termine la preuve. □

3.2 Une sémantique symbolique pour les DTPNs

Dans cette section, on cherche à regrouper les états d'un DTPN en classe d'états, appelés *état symbolique*. En effet, la sémantique étant en temps dense, il existe (la plupart du temps) une infinité d'états. Une abstraction symbolique permettant de se ramener à un ensemble fini d'états, tout en conservant suffisamment d'informations pour notre application, est proposée.

Soit l'opération $\ominus$ définie sur les éléments de $\mathbb{R}_+$ et prenant ses valeurs dans $\mathbb{R}_+ \cup \{-\infty\}$, telle que :

$$\forall a, b \in \mathbb{R}_+,\ a \ominus b = \begin{cases} a - b & \text{si } a \geq b \\ -\infty & \text{sinon} \end{cases}$$

3.2.1 Le délai dynamique

Dans cette sous-section, on définit une nouvelle sémantique équivalente pour les DTPNs basée sur des valeurs d'horloge décroissantes. Cette dernière nous permettra de simplifier l'écriture de la sémantique symbolique, et de réduire la complexité de l'exploration des états.

Soit un DTPN $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ et soit (M, v, χ) un état de $\mathcal{N}$. La fonction de valuation des transitions v est remplacée par une fonction de *délai dynamique* Δ. Le nouvel état correspondant à l'état (M, v, χ) est donc un n-uplet (M, Δ, χ) tel que pour toutes transitions sensibilisées $t \in \mathsf{enab}(M)$, $\Delta(t) = \delta(t) \ominus v(t)$. La valeur $\Delta(t)$ correspond au délai restant avant que la transition t soit tirable.

L'état initial est (M_0, Δ_0, χ_0) tel que $\forall t \in T,\ \Delta_0(t) = \delta(t)$.

Une attente $d \in \mathbb{R}_{+*}$ est possible depuis un état (M, Δ, χ) si $\forall t \in \mathsf{enab}(M) \setminus T_D$, $\Delta(t) \geq d$. La transition temporelle $(M, \Delta, \chi) \xrightarrow{d} (M, \Delta', \chi')$ produit la fonction de délai dynamique :

$$\forall t \in T,\ \Delta'(t) = \begin{cases} \Delta(t) \ominus d & \text{si } t \in \mathsf{enab}(M) \\ \Delta(t) & \text{sinon} \end{cases}$$

Une transition sensibilisée non-retardable (resp. retardable) t est tirable lorsque $\Delta(t) = 0$ (resp. $\Delta(t) = 0$ ou $-\infty$). La transition discrète $(M, \Delta, \chi) \xrightarrow{\tau} (M', \Delta', \chi')$ produit la

fonction de délai dynamique :

$$\forall t \in T,\ \Delta'(t) = \begin{cases} \delta(t) & \text{si } t \in \uparrow \mathsf{enab}(M, \tau) \\ \Delta(t) & \text{sinon} \end{cases}$$

Un exemple de DTPN et une partie de son graphe d'états, obtenu avec les délais dynamiques, sont donnés dans la Figure 3.3. Comme lorsqu'on représente les états avec des valuations, on ne représente que les délais dynamiques des transitions sensibilisées et ils sont barrés lorsque la transition est désactivée ($\chi(t) = \bot$).

En utilisant les délais dynamiques, seule l'information du délai restant ou du délai terminé est retenue. On ne retient pas la valeur d'horloge des transitions lorsqu'elles ont déjà dépassé leur délai. Ainsi, dès lors que t_1 a atteint son délai en q_1, tous les états qui suivront seront tels que $\Delta(t_1) = -\infty$, jusqu'à ce que t_1 soit tirée. C'est le cas, par exemple de tous les états « entre » q_1 et q_2.

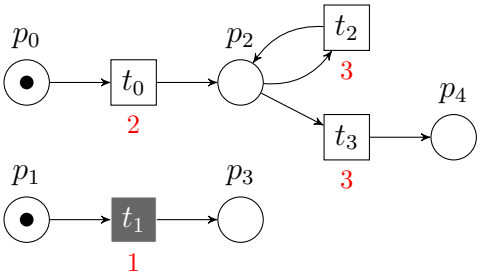
Ceci permet de fusionner des états qui ont un comportement similaire. Par exemple, depuis l'état q_{11} peu importe l'attente qu'on effectue, on restera dans le même état.

3.2.2 Les états symboliques

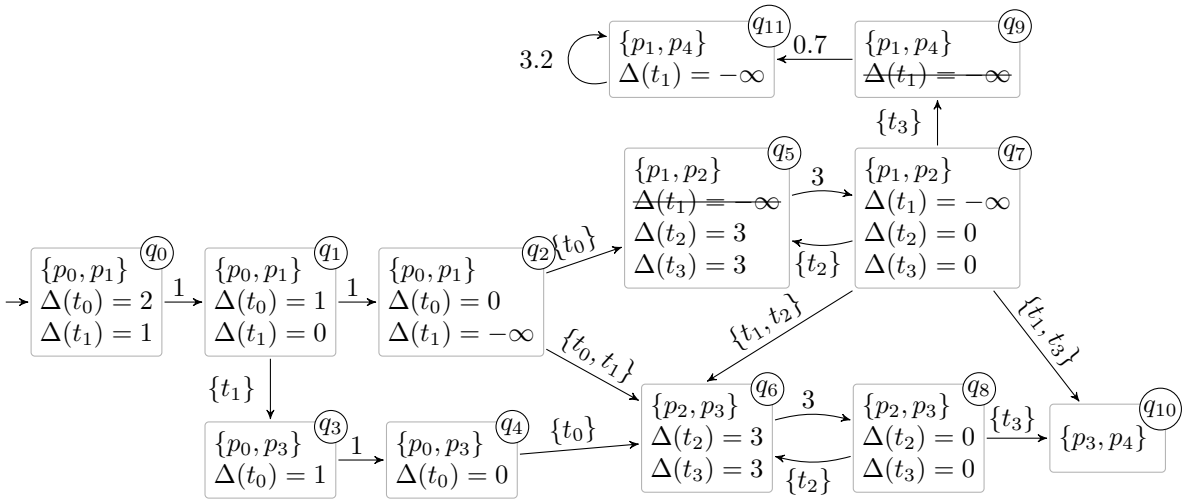
Dans cette sous-section nous proposons une abstraction symbolique dans l'objectif de regrouper des états. Tous les états accessibles depuis un même état par progression du temps sont regroupés ensemble.

Soit $q = (M, \Delta, \chi)$ et $q' = (M', \Delta', \chi')$ deux états tels que q' est accessible depuis q en écoulant d unités de temps. Il y a une relation de bisimulation (non-temporelle) entre q et q', c'est-à-dire que toute exécution discrète faisable depuis q est également faisable depuis q' (et inversement). On peut donc représenter un état symbolique par l'état « originel » contenu, c'est-à-dire l'état depuis lequel tous les états contenus sont accessibles par progression du temps. En d'autres termes, on peut représenter un état symbolique par l'état qu'il contient qui a les plus grandes valeurs de délai dynamique.

Définition 25 (État symbolique)**.** *Un état symbolique est un n-uplet* $(M, \widehat{\Delta}, \chi)$ *dans lequel* M *est un marquage,* χ *une fonction d'activation et* $\widehat{\Delta}$ *est un ensemble de délais*



(a) Exemple de DTPN $\mathcal{N}_4$



(b) Partie du graphe d'états de $\mathcal{N}_4$

FIGURE 3.3 – États d'un DTPN avec le délai dynamique

dynamiques défini par :

$$(M, \widehat{\Delta}, \chi) = \{(M, \Delta, \chi)\} \cup \{(M, \Delta', \chi_0) \mid \forall t \in \mathsf{enab}(M),\ \Delta'(t) = \Delta(t) \ominus d,$$
$$\text{avec } d \in \mathbb{R}_{+*} \text{ et } d \leq \min_{t \in \mathsf{enab}(M) \setminus T_D} (\Delta(t))\}$$

avec pour convention si $\mathsf{enab}(M) \setminus T_D = \emptyset$ *alors* d *n'est pas borné.*

L'état symbolique $(M, \widehat{\Delta}, \chi)$ *contient tous les états accessibles depuis l'état* (M, Δ, χ) *par progression du temps.*

Remarque. *Dans le cas où* d *n'est pas borné (s'il n'y a que des transitions retardables sensibilisées), les valeurs de* Δ' *restent bornées. En effet, par définition l'opération* $\ominus$ *sature à zéro, et lorsqu'elle dépasse zéro (en négatif) elle assigne le symbole* $-\infty$*.*

3.2.3 La sémantique symbolique

Le successeur d'un état symbolique se calcule naturellement : on sélectionne un état dans lequel un pas maximal est tirable, on calcule le successeur par ce tir, puis on en déduit le nouvel état symbolique (par écoulement du temps).

Définition 26 (Successeur symbolique)**.** *Soit $(M, \widehat{\Delta}, \chi)$ un état symbolique et τ un pas maximal de $(M, \widehat{\Delta}, \chi)$. Le successeur symbolique de $(M, \widehat{\Delta}, \chi)$ par τ est l'état symbolique $(M', \widehat{\Delta'}, \chi')$ tel qu'il existe $d \in \mathbb{R}_{+*}$ et (M, Δ_d, χ_d), et $(M, \Delta, \chi) \xrightarrow{d} (M, \Delta_d, \chi_d) \xrightarrow{\tau} (M', \Delta', \chi')$.*

Ce successeur symbolique est noté $\mathsf{Succ}((M, \widehat{\Delta}, \chi), \tau)$.

Lemme 3. *Dans la Définition 26, si le successeur de $(M, \widehat{\Delta}, \chi)$ par τ existe alors la transition temporelle $(M, \Delta, \chi) \xrightarrow{d} (M, \Delta_d, \chi_d)$ est définie par $d = \max_{t \in \tau}(\Delta(t))$.*

Démonstration. Le pas maximal τ est tirable si toutes les transitions qu'il contient ont atteint leur délai, c'est-à-dire $\forall t \in \tau,\ \Delta_d(t) \leq 0$. □

La sémantique symbolique d'un DTPN $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ est donc définie par un Système de Transition $\mathfrak{S}_\mathcal{N} = (S, s_0, 2^T, \hookrightarrow)$, avec S l'ensemble des états symboliques, $s_0 = (M_0, \widehat{\Delta_0}, \chi_0)$ et $(M, \widehat{\Delta}, \chi) \overset{\tau}{\hookrightarrow} (M', \widehat{\Delta'}, \chi')$ si et seulement si $(M', \widehat{\Delta'}, \chi') = \mathsf{Succ}((M, \widehat{\Delta}, \chi), \tau)$. On note $\mathsf{Reach}(\mathcal{N})$ l'ensemble des états symboliques accessibles à partir de cette sémantique.

Théorème 3.2.1. *Soit $\mathcal{N}$ un DTPN borné.* $\mathsf{Reach}(\mathcal{N})$ *est un ensemble fini.*

Démonstration. Premièrement, le nombre de marquages accessibles est borné. Deuxièmement, à partir du Lemme 2, on déduit qu'il existe, dans un état symbolique donné, un nombre fini d'états depuis lesquels un pas maximal est tirable. Enfin, il existe un nombre fini de transitions, et donc un nombre fini de pas maximal tirables depuis un état. □

Reprenons l'exemple du DTPN $\mathcal{N}_4$ de la Figure 3.3a. La construction de son graphe d'états symboliques produit le résultat dessiné dans la Figure 3.4. Un état symbolique $(M, \widehat{\Delta}, \chi)$ est représenté par son état « originel » contenu (M, Δ, χ).

3.3 Expressivité des RTPNs

Dans cette section nous nous intéressons à l'expressivité de la classe Réseau de Petri Temporisé avec reset et transitions retardables, et nous montrons qu'il existe un Automate Temporisé à une seule horloge fortement bisimilaire.

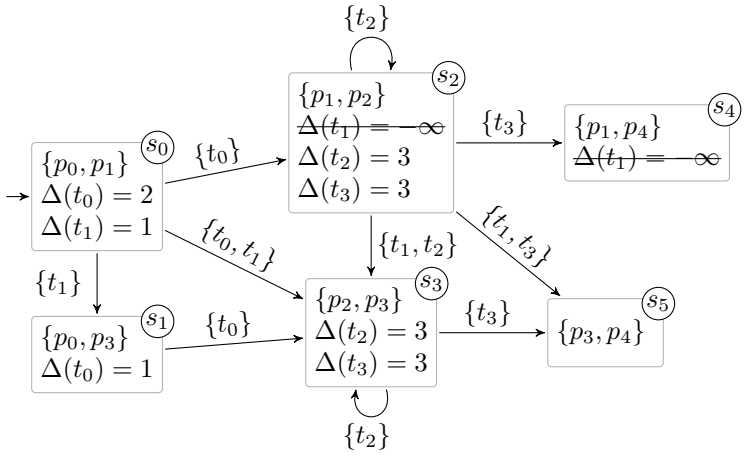


FIGURE 3.4 – Graphe d'états symboliques complets de $\mathcal{N}_4$

3.3.1 Sémantique symbolique pour les RTPNs

La sémantique symbolique présentée dans la Section 3.2 s'applique également aux RTPNs (qui sont une sous-classe des DTPNs).

Le délai dynamique Δ est étendue avec une valeur pour l'action reset : $\Delta \in (\mathbb{R}_+ \cup \{-\infty\})^{T \cup \{\mathsf{reset}\}}$. Il est initialisé à la borne supérieure de l'intervalle de reset : $\Delta_0(\mathsf{reset}) = \overline{I_{\mathsf{reset}}}$. Une transition temporelle $d \in \mathbb{R}_{+*}$ fait décroître le délai dynamique du reset de même que celui de toutes les transitions sensibilisées. L'action reset est tirable lorsque $\Delta(\mathsf{reset}) \in [0, \overline{I_{\mathsf{reset}}} - \underline{I_{\mathsf{reset}}}]$. Son tir $(M, \Delta, \chi) \xrightarrow{\mathsf{reset}} (M, \Delta', \chi)$ réinitialise le délai dynamique : $\Delta' = \Delta_0$.

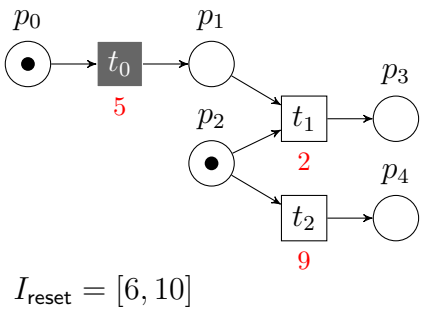
Un état symbolique $(M, \widehat{\Delta}, \chi)$ est maintenant défini par :

$$(M, \widehat{\Delta}, \chi) = \{(M, \Delta, \chi)\} \cup \{(M, \Delta', \chi_0) \mid \forall x \in \mathsf{enab}(M) \cup \{\mathsf{reset}\},\ \Delta'(x) = \Delta(x) \ominus d,$$
$$\text{avec } d \in \mathbb{R}_{+*} \text{ et } d \leq \min_{x \in (\mathsf{enab}(M) \setminus T_D) \cup \{\mathsf{reset}\}} (\Delta(x))\}$$

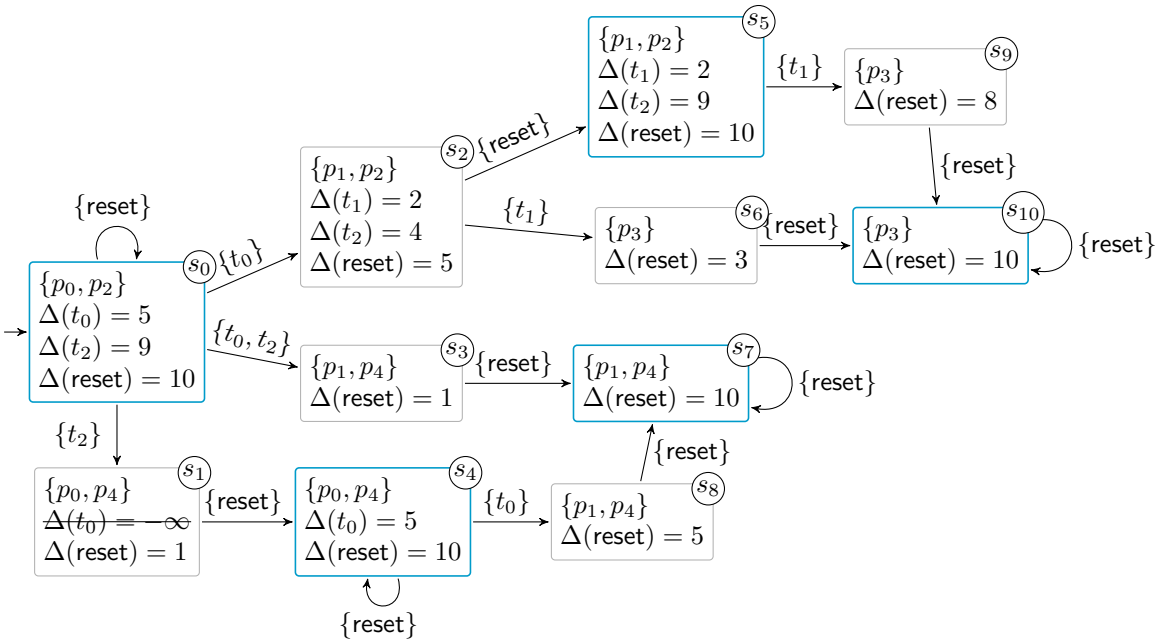
Le successeur d'un état symbolique est étendu naturellement avec l'action reset : $(M, \widehat{\Delta'}, \chi') = \mathsf{Succ}((M, \widehat{\Delta}, \chi), \mathsf{reset})$ ssi il existe $d \in \mathbb{R}_{+*}$ et (M, Δ_d, χ_d) tels que $(M, \Delta, \chi) \xrightarrow{d} (M, \Delta_d, \chi_d) \xrightarrow{\mathsf{reset}} (M, \Delta', \chi')$.

Un exemple de RTPN $\mathcal{N}_5$ et son graphe d'états symboliques sont donnés Figure 3.5. Comme pour les DTPNs, un état symbolique $(M, \widehat{\Delta}, \chi)$ est représenté par l'état (M, Δ, χ). De plus, les états symboliques obtenus après un reset sont encadrés en cyan.

Remarquons que dans un état symbolique $(M, \widehat{\Delta}, \chi)$ de RTPN, les transitions sensibilisées t vérifient la contrainte diagonale $(\Delta(\mathsf{reset}) \ominus \Delta(t) = c)$ avec $c \in \mathbb{N} \cup \{-\infty\}$, et les transitions non-sensibilisées t' vérifient $\Delta(t') = \delta(t')$. En effet, dès lors qu'une transi-



(a) Exemple de RTPN $\mathcal{N}_5$



(b) Graphe d'états symboliques de $\mathcal{N}_5$

FIGURE 3.5 – Sémantique symbolique pour un RTPN

tion est sensibilisée, la différence (au regard de l'opérateur $\ominus$) entre son délai dynamique et le délai dynamique du reset est fixée, et ce jusqu'au prochain tir de reset. Si on note $\Delta(\text{reset}) \ominus \Delta(t) = (\overline{I_{\text{reset}}} - c') \ominus \delta(t)$, cela signifie que la transition t a été sensibilisée c' unité de temps après le dernier tir de reset. Autrement dit, il est possible de décrire l'ensemble des délais dynamiques en connaissant uniquement le délai dynamique du reset.

3.3.2 Définition d'un Automate Temporisé

Une *contrainte atomique* est une formule de la forme $x \bowtie c$ avec $x \in X$, $c \in \mathbb{N}$ et $\bowtie \in \{<, \leq, \geq, >, =\}$. L'ensemble des *contraintes* sur un ensemble X de variables est noté

$\xi(X)$ et contient les conjonctions de contraintes atomiques.

Les automates temporisés, introduit par Alur et Dill [AD94] étendent les automates finis, avec un ensemble fini d'horloges :

Définition 27. *Automate Temporisé Un Automate Temporisé (TA)* $\mathcal{A}$ *est un n-uplet* $(L, l_0, X, \Sigma, E, \mathsf{Inv})$ *dans lequel :*

- L *est un ensemble de* localités,
- $l_0 \in L$ *est la* localité initiale,
- X *est un ensemble fini d'*horloges,
- Σ *est un ensemble fini d'*actions,
- $E \subseteq L \times \xi(X) \times \Sigma \times 2^X \times L$ *un ensemble fini d'*arcs *où* $e = (l, \gamma, a, R, l') \in E$ *représente un arc allant de* l *vers* l' *avec la* garde $\gamma \in \xi(X)$*, l'action* $a \in \Sigma$ *et l'ensemble de réinitialisation* $R \subseteq X$,
- $\mathsf{Inv} \in \xi(X)^L$ *associe un* invariant *à chaque localité.*

Les invariants sont restreints aux conjonctions de termes de la forme $x \leq k$ *pour* $x \in X$ *et* $k \in \mathbb{N}$.

Une valuation d'horloge est une fonction $\nu : X \to \mathbb{R}_+$. Soit $R \subseteq X$, alors $v[R \mapsto 0]$ dénote la valuation telle que $\forall x \in X \setminus R$, $\nu[R \mapsto 0](x) = \nu(x)$ et $\forall x \in R$, $\nu[R \mapsto 0](x) = 0$. La relation de satisfaction $\nu \models c$ pour $c \in \xi(X)$ est définie de manière naturelle.

Définition 28 (Sémantique d'un TA)**.** *La sémantique d'un TA* $(L, l_0, X, \Sigma, E, \mathsf{Inv})$ *est définie par le Système de Transition Temporisé* $\mathcal{S} = (Q, q_0, \to)$ *tel que* $Q = \{(l, \nu) \in L \times (\mathbb{R}_+)^X \mid \nu \models \mathsf{Inv}(l)\}$*,* $q_0 = (l_0, \bar{\mathbf{0}})$ *est l'état initial, et* $\to \in Q \times \Sigma \cup \mathbb{R}_{+*} \times Q$ *est la relation de transition défini par :*

- *une transition d'action :* $(l, \nu) \xrightarrow{a} (l', \nu')$ *ssi* $\exists (l, \gamma, a, R, l') \in E$ *t.q.* $\nu \models \gamma$*,* $\nu' = \nu[R \mapsto 0]$ *et* $\nu' \models \mathsf{Inv}(l')$ *;*
- *une transition temporelle :* $(l, \nu) \xrightarrow{d} (l', \nu')$ *ssi* $l = l'$*,* $\nu' = \nu + d$ *et* $\nu' \models \mathsf{Inv}(l)$.

3.3.3 Traduction vers un automate temporisé à une horloge

Comme vu précédemment, pour décrire l'état exact d'un RTPN dans un état symbolique (un point dans la zone), il suffit de connaître la valeur de $\Delta(\mathsf{reset})$. En se basant sur ce principe, on peut construire un automate temporisé bisimilaire, dont les localités correspondent aux états symboliques et dont l'unique horloge permet de mesurer l'évolution de $\Delta(\mathsf{reset})$.

Soit un RTPN $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\mathsf{reset}}, M_0)$ et soit
$\mathfrak{S}_\mathcal{N} = (\mathsf{Reach}(\mathcal{N}), (M_0, \widehat{\Delta_0}, \chi_0), 2^T \cup \{\mathsf{reset}\}, \hookrightarrow)$ sa sémantique symbolique.

Nous définissons un Automate Temporisé associé $\mathcal{A}_\mathcal{N}$ dont l'unique horloge x prendra les valeurs de $(\overline{I_{\mathsf{reset}}} - \Delta(\mathsf{reset}))$. Le TA $\mathcal{A}_\mathcal{N} = (L, l_0, X, \Sigma, E, \mathsf{Inv})$ est construit à partir du graphe d'états symbolique de $\mathcal{N}$, $SG(\mathfrak{S}_\mathcal{N})$:

- ϕ: $\mathsf{Reach}(\mathcal{N}) \mapsto L$ est une bijection
- $L = \{\phi(s) \mid s \in \mathsf{Reach}(\mathcal{N})\}$
- $X = \{x\}$
- La localité initiale est $l_0 = \phi((M_0, \widehat{\Delta_0}, \chi_0))$
- Chaque localité $l \in L$ possède l'invariant $(x \leq \overline{I_{\mathsf{reset}}})$
- Pour chaque arête $s \xhookrightarrow{\mathsf{reset}} s'$ de $SG(\mathfrak{S}_\mathcal{N})$, il existe une arête $(\phi(s), x \geq \underline{I_{\mathsf{reset}}}, \{\mathsf{reset}\}, \{x\}, \phi(s')))$ dans E
- Pour chaque arête $(M, \widehat{\Delta}, \chi) \overset{\tau}{\hookrightarrow} (M', \widehat{\Delta'}, \chi')$ de $SG(\mathfrak{S}_\mathcal{N})$ on a :
  - il existe une arête $(\phi((M, \widehat{\Delta}, \chi)), x = (\overline{I_{\mathsf{reset}}} - \Delta'(\mathsf{reset})), \tau, \emptyset, \phi((M', \widehat{\Delta'}, \chi')))$ dans E,
  - pour toute transition $t \in \tau$ non retardable $t \notin T_D$ la contrainte suivante est ajoutée (par conjonction) $(x \leq \overline{I_{\mathsf{reset}}} - \Delta'(\mathsf{reset}))$ à l'invariant $\mathsf{Inv}(\phi((M, \widehat{\Delta}, \chi)))$

L'Automate Temporisé $\mathcal{A}_{\mathcal{N}_5}$ construit à partir du graphe d'états symboliques du RTPN $\mathcal{N}_5$ de la Figure 3.5 est dessiné dans la Figure 3.6. Les invariants sont représentés en orange, les gardes et les réinitialisations $(x \leftarrow 0)$ en magenta, et les actions en noir. Les localités accessibles par une action reset sont représentées en cyan. Les conjonctions de contraintes sur les invariants sont simplifiées (contrainte minimale) pour éviter de surcharger la figure.

Remarquons que si $\mathcal{N}$ est borné, alors l'automate $\mathcal{A}_\mathcal{N}$ est nécessairement fini, car il possède autant de localités qu'il y a d'états symboliques dans $\mathsf{Reach}(\mathcal{N})$, et ce dernier est un ensemble fini d'après le Théorème 3.2.1.

Théorème 3.3.1. *Le RTPN $\mathcal{N}$ et l'Automate Temporisé à une horloge $\mathcal{A}_\mathcal{N}$ sont fortement bisimilaire.*

Démonstration. Soit (M, Δ, χ) un état de $\mathcal{N}$, et $\phi((M', \widehat{\Delta'}, \chi'), \nu)$ un état de $\mathcal{A}_\mathcal{N}$. On définit la relation $\sim$ par $(M, \Delta, \chi) \sim \phi((M', \widehat{\Delta'}, \chi'), \nu)$ ssi $(M, \Delta, \chi) \in (M', \widehat{\Delta'}, \chi')$ et $\nu(x) = \overline{I_{\mathsf{reset}}} - \Delta(\mathsf{reset})$. Cette relation est, par construction, une bisimulation. □

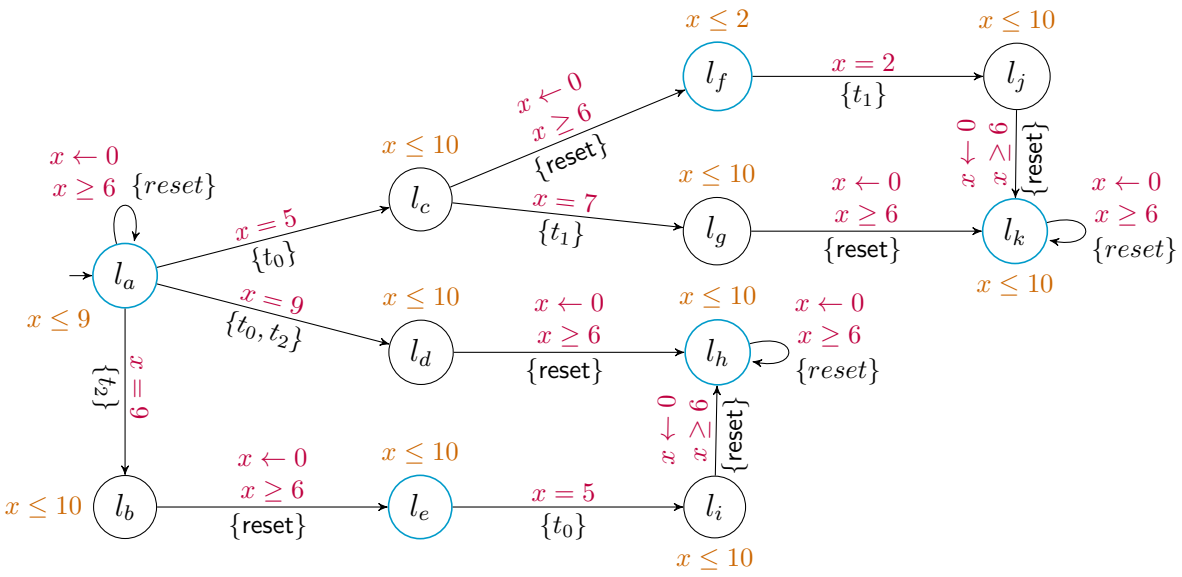


FIGURE 3.6 – Automate Temporisé correspondant au graphe d'états symboliques de $\mathcal{N}_5$

3.3.4 Corollaires sur le langage

Grâce à la traduction d'un RTPN et à la bisimulation démontrée précédemment, on hérite de résultats de décidabilité établis sur les TAs à une horloge.

Étant donné deux modèles temporisés A et B, le problème d'inclusion des langages consiste à se demander si tous les mots temporisés reconnus par B le sont aussi par A. Ce problème est démontré indécidable pour la classe Automate Temporisé. Cependant, il devient décidable pour la sous-classe des Automates Temporisés à une seule horloge, sur des mots finis [OW04].

Corollaire. *Le problème d'inclusion des langages sur des mots finis est décidable pour la classe RTPN.*

Étant donné un modèle temporisé A, le problème d'universalité consiste à se demander si A reconnaît tous les mots temporisés. Alur et Dill ont démontré que ce problème était indécidable pour un Automate Temporisé à deux horloges. Cependant pour un Automate Temporisé à une horloge, le problème d'universalité sur des mots finis devient décidable [Abd+08].

Corollaire. *Le problème d'universalité sur des mots finis est décidable pour la classe RTPN.*

3.4 Complexité théorique et pratique

Dans cette section, nous nous intéressons à la complexité théorique du problème d'accessibilité pour la classe DTPN. La complexité en pratique du calcul du successeur symbolique défini dans la Section 3.2 est également établie.

3.4.1 Problèmes d'accessibilité

Le problème d'accessibilité consiste à se demander pour un état donné, s'il existe une exécution du modèle qui se termine dans cet état.

Théorème 3.4.1. *Le problème d'accessibilité pour les classes RTPN et DTPN est indécidable.*

Démonstration. La classe Réseau de Petri (sans temps) avec la sémantique à pas maximaux est une sous-classe de RTPN (qui est elle-même une sous-classe de DTPN). En effet, un PN (sans temps) avec la sémantique à pas maximaux $\mathcal{N}_1$ peut être simulé par un RTPN $\mathcal{N}_2 = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\mathsf{reset}}, M_0)$ avec le même squelette $(P, T, {}^\bullet(.), (.)^\bullet, M_0) = \mathcal{N}_1$ et tel que $T_D = \emptyset$, $\forall t \in T$, $\delta(t) = 0$ et $\underline{I_{\mathsf{reset}}} > 0$. De plus, la classe Réseau de Petri (sans temps) avec la sémantique à pas maximaux est Turing-complète [Pop13]. □

Pour la suite, nous considérons uniquement des réseaux bornés, c'est-à-dire que le nombre de jetons pour chaque place du réseau est borné. Tout d'abord, nous nous intéressons aux réseaux saufs, c'est-à-dire que chaque place contient au plus 1 jeton.

Lemme 4. *Le problème d'accessibilité pour un PN sauf (sans temps) avec sémantique à pas maximaux, un TPN sauf, un RTPN sauf et un DTPN sauf est PSPACE-difficile.*

Démonstration. Considérons tout d'abord un Réseau de Petri sauf. On se ramène à un problème d'accessibilité pour un Réseau de Petri avec sémantique d'entrelacement. Soit $\mathcal{N} = (P, T, {}^\bullet(.), (.)^\bullet, M_0)$ un PN sauf avec sémantique d'entrelacement. Soit $\mathcal{N}' = (P, T', pre, post, M_0)$ un PN avec sémantique à pas maximaux tel que $T \subseteq T'$, $\forall t \in T$, $pre(t) = {}^\bullet t$ et $post(t) = t^\bullet$. De plus, $T' = T \cup T_p$ avec T_p un ensemble de transitions défini par $T_p \cap T = \emptyset$ et $\forall p \in P$ il existe une transition $t_p \in T_p$ telle que $pre(t_p)(p) = post(t_p)(p) = 1$. De façon plus informelle, on ajoute une transition qui boucle sur chaque place de P.

Ainsi, on crée un conflit entre toutes les transitions de $\mathcal{N}'$ avec une transition de T_p. Étant donné que le tir d'une transition t_p de T_p préserve le marquage de p, ce nouveau PN $\mathcal{N}'$ simule la sémantique d'entrelacement de $\mathcal{N}$.

L'accessibilité pour un PN sauf avec sémantique d'entrelacement est un problème PSPACE-complet [CEP95]. Il suit donc que l'accessibilité pour un PN sauf avec sémantique à pas maximaux est PSPACE-difficile.

Ce résultat s'étend ensuite aux classes TPN, RTPN et DTPN qui sont des sur-classes de PN à pas maximaux. □

On s'intéresse au problème de vérification de propriétés TCTL, dont le problème d'accessibilité est un cas particulier.

Pour utiliser les méthodes de vérification discrètes usuelles, il faut construire une structure finie. Nous nous basons sur la relation d'équivalence de région $\simeq$ sur les valuations d'horloges, initialement définie sur les TAs [ACD93 ; AD94]. La relation d'équivalence de région $\simeq$ peut-être facilement appliquée aux DTPNs comme cela a été fait pour les $\mathbb{T}$PNs dans [BGR09]. Pour construire le graphe de région, on adapte simplement le calcul des transitions d'action (en utilisant la sémantique à pas maximaux). Le graphe des régions est exponentiel par rapport à la taille du réseau $|T| + |P|$. Pour vérifier une propriété TCTL, nous pourrions construire le graphe des régions complet, et étiqueter ses nœuds avec les sous-formules de la propriété TCTL. L'algorithme nécessiterait alors un espace exponentiel par rapport à la taille du réseau.

Cependant en procédant comme dans [ACD93 ; BGR09] on peut construire un algorithme polynomial en espace pour vérifier une propriété TCTL.

Théorème 3.4.2. *L'accessibilité et la vérification TCTL pour un DTPN borné sont PSPACE-complets.*

Démonstration. Tout d'abord, les marquages d'un DTPN k-borné sont dans $([0..k])^P$. Avec un encodage binaire, un marquage peut être représenté en $|P| \cdot \log_2(k+1)$ bits, c'est-à-dire en $O(|P|)$. De même, une région est définie par $|T|$ valeurs bornées dans $\mathbb{N}$, et peut donc être représentée en $O(|T|)$.

Comme dans [ACD93 ; BGR09], on vérifie une propriété TCTL par une procédure récursive $label(v, \varphi)$ appelée pour chaque sous-formule de la propriété. Cette dernière renvoie $vrai$ si le nœud v doit être étiqueté par φ, et $faux$ sinon. Elle explore des chemins du graphe des régions de manière non-déterministe, et n'a besoin de garder en mémoire que le nœud actuel et le nœud précédent. Ainsi, comme chaque nœud peut-être représenté en

espace polynomial, cet algorithme est non-déterministe et en espace polynomial. D'après le théorème de Savitch, il existe un algorithme déterministe en espace polynomial qui résout le même problème.

Enfin, le Lemme 4 permet de conclure. □

3.4.2 Calcul du successeur symbolique

Bien que la complexité théorique du problème d'accessibilité pour les TPNs, RTPNs et DTPNs est polynomiale (en mémoire), en pratique il n'existe pas d'algorithme permettant d'atteindre cette complexité. En effet, leur espace d'états (même symboliques) est exponentiel par rapport à la taille du réseau (nombre de transitions ou nombre de places). Cependant, il reste intéressant de chercher à réduire la complexité du calcul du successeur symbolique.

Habituellement, la construction de l'espace d'états symboliques d'un modèle temporisé se base sur la structure de Difference Bound Matrix (DBM). Les cœfficients de cette matrice D sont les bornes supérieures des différences de valeur d'horloge de chaque couple de transitions du réseau : $v(t_i) - v(t_j) \leq D_{ij}$ (la relation peut être stricte). Elle a donc n^2 cœfficients, où n est le nombre de transition du réseau (en fait $(n+1)^2$ car il y a une horloge supplémentaire qui vaut toujours 0).

Pour calculer le successeur symbolique, il faut trouver une forme canonique de cette matrice. Pour ce faire on a recours à un algorithme du plus court chemin, par exemple l'algorithme de Floyd-Warshall, qui est en complexité $O(n^3)$ par rapport au nombre de transitions n du réseau. Cette complexité a été cependant réduite à $O(n^2)$ par les auteurs de [BR07].

Dans le cas des DTPNs, les DBMs ne sont pas nécessaires car comme on l'a vu dans la Section 3.2 il n'y a besoin que d'une valeur (ou symbole) par transition pour décrire un état symbolique. Ainsi la structure de données du délai dynamique Δ simplifie grandement le calcul du successeur symbolique.

Théorème 3.4.3 (Complexité successeur symbolique)**.** *Soit $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ un DTPN. Soit $(M, \widehat{\Delta}, \chi)$ un état symbolique de $\mathcal{N}$ et soit τ un pas maximal depuis $(M, \widehat{\Delta}, \chi)$. Le successeur symbolique de $(M, \widehat{\Delta}, \chi)$ par τ, $\mathsf{Succ}((M, \widehat{\Delta}, \chi), \tau)$, peut-être calculé en $O(n)$ avec $n = |T|$ le nombre de transition de $\mathcal{N}$.*

Démonstration. Dans la Définition 26 du successeur symbolique, le successeur $(M', \widehat{\Delta'}, \tau')$ est défini à partir des transitions $(M, \Delta, \chi) \xrightarrow{d} (M, \Delta_d, \chi_d) \xrightarrow{\tau} (M', \Delta', \chi')$.

— le délai d peut-être calculé en $|\tau|$ étapes, car d'après le Lemme 3 il s'agit du calcul d'un maximum,
— le successeur de la transition temporelle (M, Δ_d, χ_d) peut-être calculé en $|\mathsf{enab}(M)|+|T_D|$ étapes, car il suffit de calculer les nouvelles valeurs de Δ pour les transitions sensibilisées, et de χ pour les transitions retardables,
— le successeur de la transition d'action (M', Δ', χ') peut-être calculé en au plus $|P| \times |\tau|$ étapes, pour calculer le nouveau marquage.

□

Le calcul des pas maximaux tirables depuis un état symbolique $(M, \widehat{\Delta}, \chi)$ est en fait plus compliqué qu'il n'y parait. Dans la pratique, nous calculons d'abord les délais faisables depuis (M, Δ, χ) qui nous emmènent dans des états (M, Δ_d, τ_d) où des pas maximaux sont tirables. Puis nous calculons l'ensemble des pas maximaux tirables depuis (M, Δ_d, τ_d). La complexité reste cependant en $O(n)$ avec n le nombre de transitions du réseau.

Les algorithmes d'exploration de l'espace d'états symboliques sont présentés dans la Section 3.5.

3.5 Algorithmes d'exploration de l'espace d'états symboliques

Dans cette section, nous présentons les algorithmes permettant de construire le graphe d'états symboliques d'un DTPN. Tous ces algorithmes sont implémentés dans un outil dont nous montrons une application dans un autre chapitre du manuscrit.

Nous considérons le DTPN $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, M_0)$ et son état symbolique initial $(M_0, \widehat{\Delta_0}, \chi_0)$.

3.5.1 Successeur d'un état

Ensemble des transitions sensibilisées Pour commencer, nous nous intéressons au calcul de l'ensemble des transitions sensibilisées par un marquage M. Nous avons en fait simplement besoin de savoir pour une transition t donnée, si elle est sensibilisée par M. C'est ce que fait l'Algorithme 1.

Algorithme 1. *Vérifie si une transition est sensibilisée*

1: ***function*** `IS_ENABLED`*(M, t)*

```
2:     return M ≥• t
3: end function
```

L'Algorithme 1 est de complexité $O(|P|)$.

Transition de la sémantique Nous souhaitons maintenant depuis un état (M, Δ, χ) calculer le successeur d'une transition temporelle, et d'une transition d'action.

L'Algorithme 2 calcule le successeur d'une transition temporelle de durée d depuis un état (M, Δ, χ). Il calcule les nouvelles valeurs de Δ pour les transitions sensibilisées et réinitialise les valeurs de χ.

Algorithme 2. *Transition temporelle de délai d*

```
1: function DELAY((M, Δ, χ), d)
2:     (M_r, Δ_r, χ_r) ← (M, Δ, χ_0)
3:     for all t ∈ T do
4:         if IS_ENABLED(M_r, t) then
5:             if Δ_r(t) ≥ d then
6:                 Δ_r(t) ← Δ_r(t) − d
7:             else
8:                 Δ_r(t) ← −∞
9:             end if
10:        end if
11:    end for
12:    return (M_r, Δ_r, χ_r)
13: end function
```

L'Algorithme 2 est de complexité $O(|T| \cdot |P|)$. Cette complexité peut-être réduite à $O(|T|)$ en sauvegardant dans l'état courant l'ensemble des transitions sensibilisées, ce qui évite un appel à `IS_ENABLED`.

L'Algorithme 3 calcule le successeur d'une transition d'action de pas maximal τ depuis une état (M, Δ, χ). Il calcule le nouveau marquage, les nouvelles valeurs de délais dynamiques pour les transitions nouvellement sensibilisées, et les transitions désactivées ou réactivées. Nous supposons qu'il reçoit en paramètre l'ensemble des pas maximaux tirables ms depuis (M, Δ, χ).

Algorithme 3. *Transition discrète de pas maximal τ*

1: ***function*** `FIRE`$((M, \Delta, \chi), \tau, ms)$
2: $(M_r, \Delta_r, \chi_r) \leftarrow (M, \Delta, \chi)$
3: $M'_r \leftarrow M_r$
4: ***for all*** $t \in \tau$ ***do***
5: $M_r \leftarrow M_r - {}^\bullet t + t^\bullet$
6: $M'_r \leftarrow M'_r - {}^\bullet t$
7: ***end for***
8: ***for all*** $t \in T$ ***do***
9: ***if*** `IS_ENABLED`$(M_r, t) \wedge \neg$`IS_ENABLED`(M'_r, t) ***then***
10: $\Delta(t) \leftarrow \delta(t)$
11: ***end if***
12: ***end for***
13: ***for all*** $t \in T_D$ ***do***
14: ***if*** `IS_ENABLED`$(M_r, t) \wedge \neg$`IS_ENABLED`(M'_r, t) ***then***
15: $\chi_r(t) \leftarrow \top$
16: ***else if*** $t \notin \tau \wedge \tau \cup \{t\} \in ms$ ***then***
17: $\chi_r(t) \leftarrow \bot$
18: ***end if***
19: ***end for***
20: ***return*** (M_r, Δ_r, χ_r)
21: ***end function***

L'Algorithme 3 est de complexité $O(|T| \cdot |P|)$.

3.5.2 Successeurs d'un état symbolique

Pour calculer les successeurs d'un état symbolique $(M, \widehat{\Delta}, \chi)$, nous calculons tout d'abord les délais qui depuis l'état (M, Δ, χ) nous mène à un état (M_d, Δ_d, χ_d) depuis lequel des transitions sont tirables. Puis depuis (M_d, Δ_d, χ_d) nous calculons les pas maximaux tirables.

Délais minimaux L'Algorithme 4 calcule l'ensemble des délais qui depuis un état (M, Δ, χ) nous mène à un état (M_d, Δ_d, χ_d) depuis lequel des transitions sont tirables. Pour ce faire, il calcule le minimum des valeurs de Δ parmi les transitions non-retardables sensibilisées, noté d_{min}. Le délai des transitions non-retardables ne peut ainsi jamais être

dépassé. Puis, il ajoute toutes les valeurs de Δ inférieures à d_{min} parmi les transitions retardables sensibilisées.

Algorithme 4. *Délais minimaux menant à un état où des tirs sont possibles*

1: ***function*** `MIN_DELAYS`$((M, \Delta, \chi))$
2: $\quad d_{min} \leftarrow +\infty$
3: $\quad$ ***for all*** $t \in T \setminus T_D$ ***do***
4: $\quad\quad$ ***if*** `IS_ENABLED`$(M, t) \wedge (\Delta(t) < d_{min})$ ***then***
5: $\quad\quad\quad d_{min} \leftarrow \Delta(t)$
6: $\quad\quad$ ***end if***
7: $\quad$ ***end for***
8: $\quad$ ***if*** $d_{min} \neq +\infty$ ***then***
9: $\quad\quad result \leftarrow \{d_{min}\}$
10: $\quad$ ***end if***
11: $\quad$ ***for all*** $t \in T_D$ ***do***
12: $\quad\quad$ ***if*** `IS_ENABLED`$(M, t) \wedge (\Delta(t) < d_{min}) \wedge (\Delta(t) \neq -\infty)$ ***then***
13: $\quad\quad\quad result \leftarrow result \cup \{\Delta(t)\}$
14: $\quad\quad$ ***end if***
15: $\quad$ ***end for***
16: $\quad$ ***return*** $result$
17: ***end function***

L'Algorithme 4 est de complexité $O(|T| \cdot |P|)$. Comme que pour l'Algorithme 2, cette complexité peut-être réduite à $O(|T|)$ en sauvegardant dans l'état courant l'ensemble des transitions sensibilisées.

Pas maximaux Pour construire les pas maximaux tirables depuis un état (M, Δ, χ), l'idée est de se ramener à une exploration d'un Réseau de Petri sans temps avec une sémantique d'entrelacement. L'ensemble des transitions tirables depuis l'état (M, Δ, χ) est sélectionné, il est noté T_f. Puis le PN $\mathcal{N}_f$ est construit à partir du squelette du DTPN $\mathcal{N}$ avec pour marquage initial M, en ne conservant que les transitions dans T_f et en supprimant les arcs sortants de toutes les transitions : $\mathcal{N}_f = (P, T_f, pre_f, post_f, M)$, avec $pre_f = {}^\bullet(.)$ et $post_f = \bar{\mathbf{0}}$.

Il suffit maintenant d'explorer le graphe d'état de $\mathcal{N}_f$. À tout état du graphe dans lequel on ne peut plus tirer de transition non-retardable, correspond un pas maximal de

$\mathcal{N}$ depuis l'état (M, Δ, χ). Le pas maximal contient toutes les transitions qui ont été tirées dans l'exécution menant à cet état. Notons que les exécutions dans lesquelles une même transition est tirée plusieurs fois sont ignorées.

Cette approche est utilisée dans l'Algorithme 5. L'exploration des états de $\mathcal{N}_f$ est réalisée par l'algorithme classique avec une liste d'attente et une liste d'états passés. Les états sont composés d'un marquage M', d'un ensemble de transitions tirées T_{step} (historique de l'exécution menant à l'état) et d'un ensemble de transitions restantes à tirer T_{remain}. Les ensembles T_{step} et T_{remain} empêchent le tir d'une même transition à plusieurs reprises. Enfin avant d'ajouter un pas T_{step} au résultat final, on s'assure qu'il contient au moins une transition t qui tire à sa date pour laquelle $\Delta(t) = 0$.

Algorithme 5. *Pas maximaux tirables depuis* (M, Δ, χ)

1: ***function*** MAXSTEP*((M, Δ, χ))*
2: $T_f \leftarrow \emptyset$
3: ***for all*** $t \in T$ ***do***
4: ***if*** IS_ENABLED$(M, t) \wedge \chi(t) \wedge ((\Delta(t) = 0) \vee (\Delta(t) = -\infty))$ ***then***
5: $T_f \leftarrow T_f \cup \{t\}$
6: ***end if***
7: ***end for***
8: $waiting \leftarrow \{(M, \emptyset, T_f)\}$, $passed \leftarrow \emptyset$, $result \leftarrow \emptyset$
9: ***while*** $waiting \neq \emptyset$ ***do***
10: *select and remove* $(M', T_{step}, T_{remain})$ *from waiting*
11: ***if*** $(M', T_{step}, T_{remain}) \notin passed$ ***then***
12: $passed \leftarrow passed \cup \{(M', T_{step}, T_{remain})\}$
13: $f_{max} \leftarrow \top$
14: ***for all*** $t' \in T_{remain}$ ***do***
15: ***if*** $M' \geq^\bullet t'$ ***then***
16: $waiting \leftarrow waiting \cup \{(M' -^\bullet t', T_{step} \cup \{t'\}, T_{remain} \setminus \{t'\})\}$
17: ***if*** $t' \notin T_D$ ***then***
18: $f_{max} \leftarrow \bot$
19: ***end if***
20: ***end if***
21: ***end for***
22: ***if*** $f_{max} \wedge (\exists t \in T_{step} \mid \Delta(t) = 0)$ ***then***
23: $result \leftarrow result \cup \{T_{step}\}$

24: ***end if***
25: ***end if***
26: ***end while***
27: ***return*** *result*
28: ***end function***

Théorème 3.5.1. *L'Algorithme 5 a une complexité au pire cas de* $O(\frac{|T|\cdot|T|!\cdot|P|}{2^{|T|}})$.

Démonstration. Pour calculer la complexité de l'Algorithme 5, on suppose qu'il explore les états en largeur (*Breadth-first search*). On note $n = |T_f|$. On se place à l'étape où il sélectionne un état $(M', T_{step}, T_{remain})$ tel que $|T_{step}| = k$. Autrement dit, il a déjà fallu tirer k transitions pour atteindre cet état, et il reste encore $(n-k) = |T_{remain}|$ transitions tirables. Dans le pire des cas, toutes les transitions de T_{remain} peuvent être tirées (pas de conflit) et ne permettent pas de construire un pas maximal. L'état $(M', T_{step}, T_{remain})$ a donc $(n-k)$ successeurs, et c'est également le cas pour tous les états au même *niveau* d'exploration que lui (tous les états obtenus après avoir tiré k transitions). Cependant, les successeurs de tous les états du *niveau* k se rejoignent deux à deux, car l'ordre de tir des transitions n'a pas d'importance. On ajoute donc $\frac{n-k}{2}$ successeurs pour chaque état du niveau k. Au niveau 0, on a 1 état qui a n successeurs. On ajoute donc $\frac{\prod_{0\leq i\leq k}(n-i)}{2^k} = \frac{n!}{2^k\cdot(n-k-1)!}$ successeurs au total au niveau k, avec $0 \leq k \leq n-2$. Enfin au dernier niveau $(n-1)$, il n'y a qu'un seul successeur. L'algorithme explore donc au plus $1 + \sum_{0\leq k\leq n-2}(\frac{n!}{2^k\cdot(n-k-1)!}) + 1$ états.

À chaque étape de la boucle, il faut parcourir les transitions restantes à tirer, et vérifier la condition $M' \geq^\bullet t'$. Pour un état du niveau k, il reste $n-k$ transitions dans T_{remain}, ce qui fait $(n-k)\cdot|P|$ étapes. La boucle d'exploration contient donc $(n+\sum_{0\leq k\leq n-2}(\frac{(n-k)\cdot n!}{2^k\cdot(n-k-1)!}) + 1)\cdot|P|$ étapes.

Pour $n \geq 3$ et pour tout k tel que $0 \leq k \leq n-2$, on a : $(n-k-1)! = \prod_{1\leq i\leq n-k-1}(i) = (n-k-1)\cdot\prod_{1\leq i\leq n-k-2}(i) \geq (n-k-1)\cdot 2^{n-k-3}$. On en déduit $\frac{(n-k)\cdot n!}{2^k\cdot(n-k-1)!} \leq \frac{n-k}{n-k-1}\cdot\frac{n!}{2^{n-3}} \leq 2\cdot\frac{n!}{2^{n-3}}$. Ainsi $\sum_{0\leq k\leq n-2}(\frac{(n-k)\cdot n!}{2^k\cdot(n-k-1)!}) \leq 2(n-1)\frac{n!}{2^{n-3}}$.

Enfin dans le pire des cas, on a $T_f = T$ et donc $n = |T|$. □

Successeurs symboliques L'Algorithme 6 calcule tous les successeurs d'un état symbolique $(M, \widehat{\Delta}, \chi)$. Comme décris plus haut, il calcule tout d'abord les délais qui depuis l'état (M, Δ, χ) mènent à un état (M_d, Δ_d, χ_d) depuis lequel des transitions sont tirables. Puis depuis (M_d, Δ_d, χ_d) il calcule les pas maximaux tirables.

Algorithme 6. *Calcule tous les successeurs symboliques de* $(M, \widehat{\Delta}, \chi)$

1: ***function*** SUCC$((M,\widehat{\Delta},\chi))$
2: ***for all*** $d \in$ MIN_DELAYS$((M, \Delta, \chi))$ ***do***
3: $(M_d, \Delta_d, \chi_d) \leftarrow$ DELAY$((M, \Delta, \chi), d)$
4: $ms \leftarrow$ MAXSTEP$((M_d, \Delta_d, \chi_d))$
5: ***for all*** $\tau \in ms$ ***do***
6: $(M', \Delta', \chi') \leftarrow$ FIRE$((M_d, \Delta_d, \chi_d), \tau, ms)$
7: $result \leftarrow result \cup \{(M', \widehat{\Delta'}, \chi')\}$
8: ***end for***
9: ***end for***
10: ***return*** $result$
11: ***end function***

3.5.3 Graphe d'états symboliques

L'Algorithme 7 explore les états symboliques de $\mathcal{N}$ par une approche classique avec une liste d'attente et une liste d'états explorés.

Algorithme 7. *Construction du graphe d'états symboliques*

1: ***function*** REACH
2: $\Delta_0 \leftarrow (\delta(t_0) \dots \delta(t_n))$
3: $\chi_0 \leftarrow (\top \dots \top)$
4: $waiting \leftarrow \{(M_0, \widehat{\Delta_0}, \chi_0)\}$
5: $passed \leftarrow \emptyset$
6: ***while*** $waiting \neq \emptyset$ ***do***
7: *select and remove* $(M, \widehat{\Delta}, \chi)$ *from* $waiting$
8: ***if*** $(M, \widehat{\Delta}, \chi) \notin passed$ ***then***
9: $passed \leftarrow passed \cup \{(M, \widehat{\Delta}, \chi)\}$
10: ***for all*** $(M', \widehat{\Delta'}, \chi') \in$ SUCC$((M, \widehat{\Delta}, \chi))$ ***do***
11: $waiting \leftarrow waiting \cup \{(M', \widehat{\Delta'}, \chi')\}$
12: ***end for***
13: ***end if***
14: ***end while***
15: ***end function***

Chapitre 4

Synthèse de pipeline

Ce chapitre s'intéresse à la synthèse de pipeline pour des circuits synchrones.

La Section 4.1 définit les problèmes que nous tentons de résoudre dans ce travail, à savoir la synthèse de pipeline optimal et la synthèse de pipeline pliant. Le premier s'intéresse à la synthèse d'un pipeline qui minimise les bascules consommées tout en assurant une fréquence minimale de fonctionnement. Le deuxième consiste à « plier » le circuit, c'est-à-dire à partager dans le temps des portions de circuits qui sont utiles à plusieurs reprises. La Section 4.2 présente des solutions existantes aux problèmes de synthèse de pipeline et d'optimisation du nombres de bascules consommées par ce dernier. Puis, notre solution au problème de synthèse de pipeline minimisant les ressources consommées est décrit dans la Section 4.3. Elle se base sur une modélisation du circuit par un Réseau de Petri Temporisé avec reset et transitions retardables (RTPN). Enfin, en appliquant la même approche et en raffinant les pipelines produits, nous montrons dans la Section 4.4 que nous sommes capables de résoudre le problème de pipeline pliant.

4.1 Définition du problème

Dans cette section, nous définissons les problèmes de conception de circuits sur lesquels portent notre travail, à savoir la synthèse de pipeline de circuit synchrone et le pliage de circuit synchrone.

4.1.1 Qu'est-ce qu'un circuit synchrone ?

La conception de circuit logique se décline sous deux paradigmes : les *circuits synchrones* et les *circuits asynchrones*.

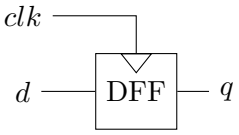
Pour comprendre ces deux approches, prenons les définitions fournies par Wikipédia :

— Un circuit synchrone est un circuit logique dans lequel les changements d'états sont synchronisés par un *signal d'horloge* [Wik21]. Ce dernier est un signal carré

périodique, dont les fronts montants, appelés *tick*, déclenchent la progression des données dans le circuit.

— Par opposition, un circuit asynchrone est un circuit logique qui n'utilise pas de signal d'horloge pour synchroniser ces composants [Wik22].

La progression des données dans un circuit synchrone à chaque tick d'horloge est obtenue à l'aide de *bascules* (en anglais *flip-flop*). La plus commune est la bascule D. Une version simplifiée (avec une seule sortie) et sa table de vérité sont représentées dans la Figure 4.1.



clk	d	q_{next}	
⌐↓	0	0	Reset
⌐↓	1	1	Set
↓⌐ , 0 ou 1	x	q	Mémoire

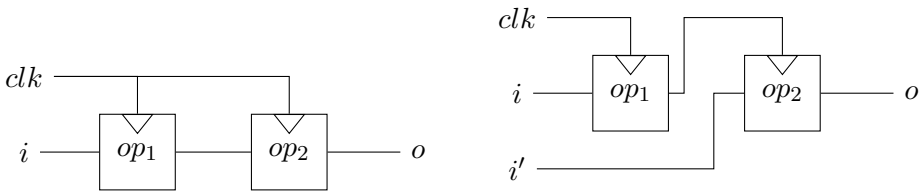
FIGURE 4.1 – Bascule D

Elle a une entrée de donnée d, une sortie de donnée q et reçoit en entrée la valeur du signal d'horloge clk. Le symbole de triangle est une notation usuelle pour représenter une entrée sensible à un front. À un front montant, c'est-à-dire lorsque clk passe de 0 à 1, q prend la valeur courante de d. Tant que le signal d'horloge clk n'a pas de front montant, la sortie q conserve sa valeur, quelle que soit la valeur x de l'entrée d, c'est un accès mémoire.

Ces bascules D sont ainsi placées entre les opérateurs logiques pour assurer une progression périodique des données dans le circuit.

Il faut en fait nuancer les définitions de circuit synchrone et asynchrone. Prenons les exemples de circuits synchrone et asynchrone dessinés dans la Figure 4.2. Pour simplifier l'exemple tous les signaux sont sur un bit. Le rythme d'exécution des opérateurs op_1 et op_2, c'est-à-dire la vitesse de progression des données dans l'opérateur, est dicté par le signal envoyé sur leur entrée triangulaire. Dans la Figure 4.2a, la synchronisation des opérateurs est contrôlée par le signal d'horloge global. Alors que dans la Figure 4.2b, l'exécution de l'opérateur op_2 progresse lorsque l'opérateur op_1 lui fournit un signal qui passe à 1 (front montant). La synchronisation de op_2 dépend du résultat de op_1, c'est donc un circuit asynchrone.

Considérons maintenant le circuit représenté dans la Figure 4.3. Supposons que la *latence* du circuit, c'est-à-dire le temps de propagation des données depuis l'entrée jusqu'à



(a) Circuit synchrone (b) Circuit asynchrone

FIGURE 4.2 – Circuit synchrone vs asynchrone

la sortie, est inférieure à la période du signal d'horloge *clk*. Ce dernier est alors un circuit synchrone, contenant un circuit asynchrone. Autrement dit, les définitions de circuit synchrone et asynchrone dépendent de la granularité avec laquelle on décrit le circuit.

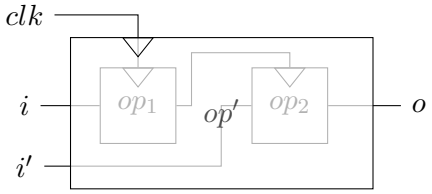


FIGURE 4.3 – Circuit synchrone contenant un circuit asynchrone

Hypothèses de travail Dans ce travail, nous ne nous intéressons qu'aux circuits synchrones. Nous considérons que tous les opérateurs sont synchronisés avec une horloge globale et donc « s'exécutent tous au même rythme ».

De plus, nous supposons que la *latence* de chaque opérateur est fixe et connue. Cette dernière sera également appelée temps de propagation, d'exécution ou de calcul, ou délai de l'opérateur. Elle est exprimée sous la forme d'un entier correspondant au nombre d'unité de temps nécessaires pour propager les données depuis les entrées jusqu'aux sorties de l'opérateur. Une unité de temps correspond à la période du signal d'horloge, autrement dit le temps écoulé entre deux ticks d'horloge.

Représentation Un exemple de circuit dans notre cas d'étude est dessiné dans la Figure 4.4. Il est constitué d'opérateur op_i et de signaux s_j. Les opérateurs ont un certain délai noté en rouge en dessous (par exemple op_1 a un délai de 6). Les signaux ont une certaines tailles (nombre de bits), notée en vert (par exemple le signal s_3 est sur 8 bits). Nous supposerons par la suite que tous les signaux d'entrées sont synchronisés (ici il n'y en a qu'un : s_0).

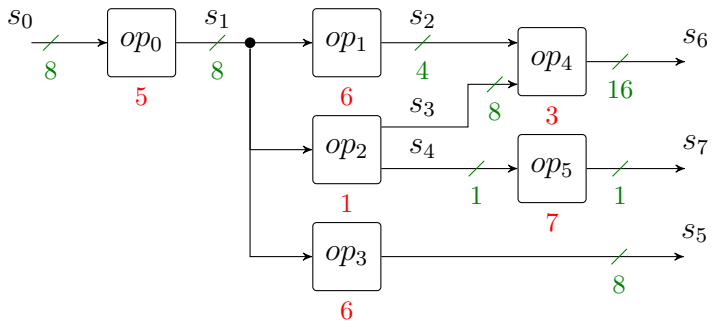


FIGURE 4.4 – Exemple de circuit C_1

La propagation des données dans le circuit peut être représentée à l'aide d'un diagramme de Gantt. Celui du circuit C_1 est dessiné dans la Figure 4.5. Les opérateurs op_1,

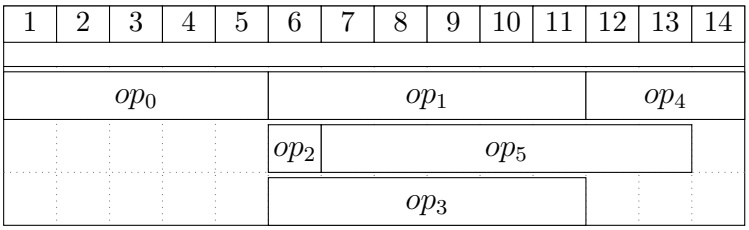


FIGURE 4.5 – Diagramme de Gantt du circuit C_1

op_2 et op_3 doivent attendre la fin de l'exécution de op_0 avant de pouvoir s'exécuter. C'est-à-dire qu'avant que les données se soient propagées dans l'opérateur op_0, ces opérateurs ne traitaient pas de données *valides*. On observe que la latence du circuit C_1 est de 14 unités de temps.

Dans cet exemple, si l'on veut s'assurer que le circuit produit des données valides, une première approche consisterait à ne fournir de nouvelles entrées que toutes les 14 unités de temps. Le circuit pourrait alors produire un nouveau résultat valide, au mieux toutes les 14 unités de temps. On dit qu'il a une *fréquence de fonctionnement* de $\frac{1}{14}$.

4.1.2 Le pipeline

Le *pipeline* est une construction permettant d'améliorer la fréquence de fonctionnement d'un circuit synchrone et donc le débit, au prix d'une latence plus élevée. Si l'on s'intéresse à une seule entrée, le temps d'exécution total du circuit pipeliné sera plus élevé. Cependant, pour une succession d'entrées le circuit fournira plus fréquemment des résultats.

Le pipeline consiste à découper le circuit en plusieurs étapes de calcul appelées *étages*. Puis ces étages peuvent être « exécutés » en parallèle avec des jeux d'entrées différents.

Pour réaliser ce découpage, il faut pouvoir mémoriser les valeurs entre chaque étage. Ceci est matérialisé par des *registres* qui sont tout simplement des collections de bascules : un registre de n bits contient n bascules.

Un exemple de pipeline du circuit C_1 est dessiné dans la Figure 4.6. Les registres sont représentés par des rectangles bleus que traversent les signaux. Dans cet exemple, le premier étage est l'ensemble des opérateurs $\{op_0, op_2\}$, le deuxième est $\{op_1, op_5, op_3\}$ et le troisième $\{op_4\}$.

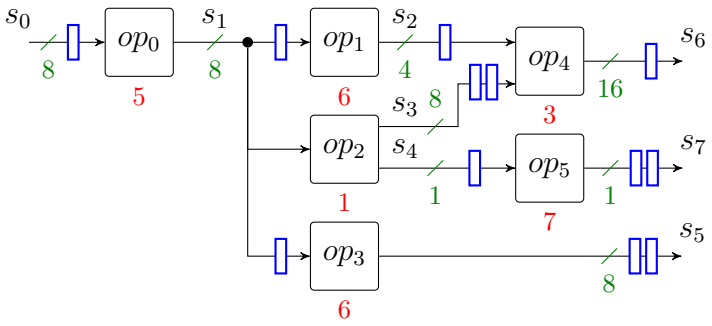


FIGURE 4.6 – Circuit C_1 avec pipeline P_1

Considérons que le pipeline P_1 a une fréquence de $\frac{1}{8}$, c'est-à-dire que les registres du pipeline sont synchronisés à un signal carré de fréquence $\frac{1}{8}$. Le diagramme de Gantt du circuit devient donc celui dessiné dans la Figure 4.7. Le registre en entrée impose une nouvelle entrée toutes les 8 unités de temps, ce qui correspond aux nouvelles lignes du diagramme de Gantt (à partir du temps 9 et à partir du temps 17). Entre les temps 9 et 16, le deuxième étage s'exécute sur la première entrée, en même temps que le premier étage s'exécute sur la deuxième entrée, etc.

L'intérêt du pipeline commence dès lors que tous les étages contiennent des données valides à traiter. Dans cet exemple, à partir de la date 17 le système est initialisé et peut fonctionner à son débit nominal : un nouveau résultat est produit toutes les 8 unités de temps. Cependant, la latence du circuit est dégradée, il faut maintenant 24 unités de temps avant qu'une entrée soit traitée complètement par le circuit, contre 14 unités de temps dans le circuit sans pipeline.

Optimisation du pipeline Dans l'exemple du pipeline P_1 de la Figure 4.6, on dénombre $(8) + (8+8+8+1) + (4+8+1+8) + (16+1+8) = 79$ bascules. Il est en fait

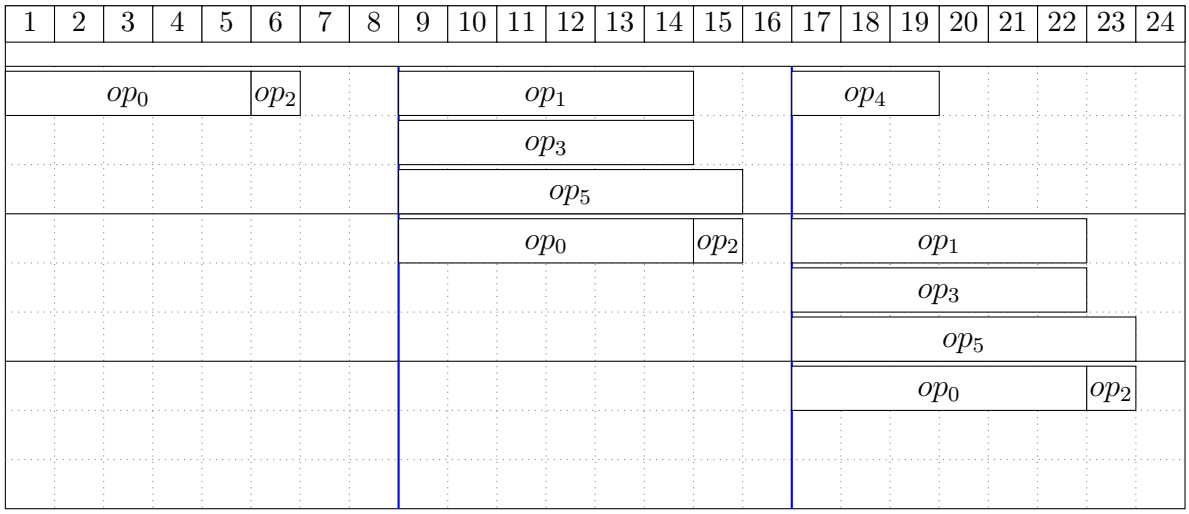


FIGURE 4.7 – Diagramme de Gantt du circuit C_1 avec pipeline P_1

possible de construire d'autres pipelines assurant la même fréquence de fonctionnement, mais nécessitant moins de bascules. Un tel pipeline est donné dans la Figure 4.8. En effet, le pipeline P_2 contient pour sa part $(8) + (8) + (4 + 8) + (16 + 1 + 8) = 53$ bascules. De plus, comme le montre son diagramme de Gantt dessiné dans la Figure 4.9, il possède la même fréquence de fonctionnement de $\frac{1}{8}$.

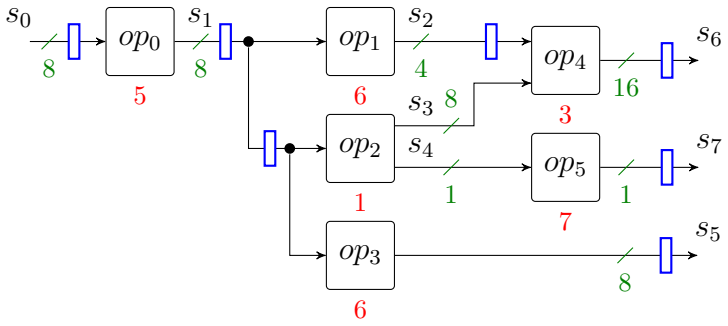


FIGURE 4.8 – Circuit C_1 avec pipeline P_2

Il existe deux manières de réduire le nombre de bascules consommées par un pipeline :

— La première s'appelle le *retiming* et consiste à déplacer un opérateur d'un étage de pipeline au précédent ou au suivant. En faisant cela, les registres séparant les étages sont déplacés et peuvent ainsi être placés sur des signaux nécessitant moins de bits. Par exemple, un registre en amont de l'opérateur op_2 consommera 8 bascules, alors que des registres en aval consommeront $8 + 1 = 9$ bascules.

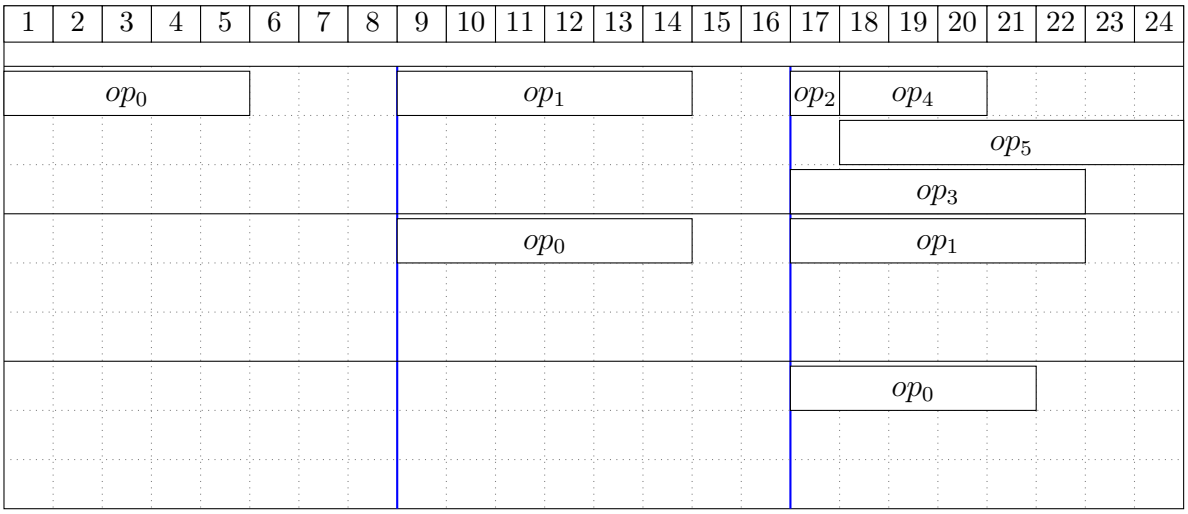


FIGURE 4.9 – Diagramme de Gantt du circuit C_1 avec pipeline P_2

— La deuxième est le *partage de registres*. Par exemple, plutôt que de placer un registre en amont de l'opérateur op_2 et un registre en amont de l'opérateur op_3, il vaut mieux partager un seul registre entre les deux opérateurs.

Le problème de la synthèse de pipeline optimal s'énonce de la manière suivante :

Problème 1 (Synthèse de pipeline optimal)**.** *Étant donné un circuit synchrone, construire un pipeline assurant une fréquence de fonctionnement minimale f et minimisant les bascules consommées.*

4.1.3 Le pliage

Le *pliage* ou *multiplexage temporel* est une méthode permettant de réduire le nombre d'opérateurs implémentés dans un circuit. Cette méthode est particulièrement intéressante pour les applications qui nécessitent un faible débit par rapport à la fréquence de l'horloge. C'est notamment le cas des applications de traitement du signal qui ont vocation à être implémentées sur un FPGA, et qui nécessitent un taux d'échantillonnage très faible par rapport à la fréquence du FPGA. Dans ce contexte, il peut être intéressant de fusionner des portions de circuits similaires, c'est-à-dire de n'implémenter qu'une seule fois des portions du circuit qui sont utilisées plusieurs fois et de séquencer leur accès en utilisant un pipeline.

La Figure 4.10 présente un circuit dont une portion est « pliable ». Les opérateurs op_i et op'_i sont deux instances d'un même opérateur. Pour alléger la figure, les tailles des

signaux ne sont pas représentées ici.

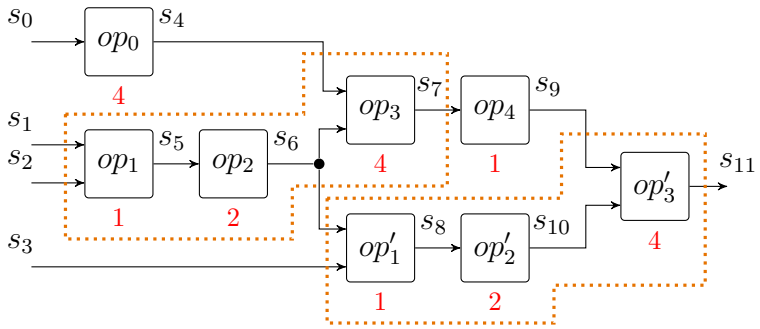


FIGURE 4.10 – Circuit C_2 pliable

Les portions de circuit encadrées en pointillés orange sont donc strictement similaires. Il est alors possible de les fusionner, c'est-à-dire d'instancier une seule fois la portion dans le circuit final.

Pour ce faire, il faut s'assurer que les portions de circuit fusionnées ne sont jamais utilisées au même moment dans les deux parties du circuit. Cette opération est réalisée grâce à un pipeline particulier. Un exemple de pipeline permettant le pliage du circuit est présenté dans la Figure 4.11.

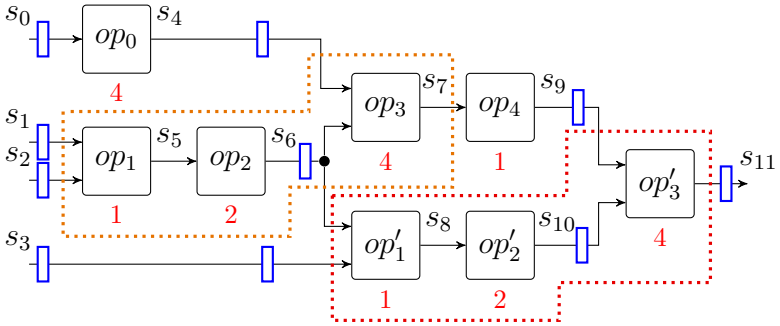


FIGURE 4.11 – Circuit C_2 avec pipeline

Pour vérifier que ce pipeline permet effectivement le pliage du circuit, il suffit de regarder son diagramme de Gantt. Ce dernier est dessiné dans la Figure 4.12.

Le pipeline a une fréquence de fonctionnement de $\frac{1}{5}$. On observe dans ce diagramme que les opérateurs deux à deux similaires (op_1 et op'_1, etc) ne sont jamais exécutés en même temps. Le pipeline permet donc de séquencer leur accès et le pliage est possible.

Il faut noter que dans le cas du pliage, la fréquence de fonctionnement du circuit est l'inverse de la latence. Autrement dit, on attend que toutes les données soient traitées

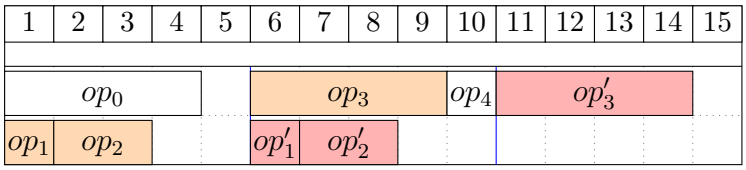


FIGURE 4.12 – Diagramme de Gantt du circuit C_2 avec pipeline

avant d'envoyer de nouvelles entrées. Dans notre cas d'étude, cette hypothèse de travail découle du fait que la fréquence d'échantillonnage est très faible par rapport à la fréquence d'horloge d'un FPGA. La majorité des travaux sur le pliage que l'on trouve dans la littérature ne considère pas cette hypothèse de travail, car ils cherchent à optimiser le débit du circuit (quantité de résultats produits par unité de temps). Dans notre cas, nous cherchons à ne pas dépasser une latence maximale, mais le débit n'est pas important. Il existe tout de même d'autres travaux qui s'intéressent à ce cas particulier de pliage [Mö+15].

Enfin, à partir de ce pipeline il est possible de construire un pliage, à l'aide de multiplexeurs, comme le montre la Figure 4.13. Chaque (dé)multiplexeur prend en entrée un signal de sélection *sel* qui évolue en fonction du signal d'horloge du pipeline. Dans notre exemple, il vaut 0 dans les intervalles de temps $[1, 5]$ et $[11, 15]$, et il vaut 1 dans l'intervalle $[6, 10]$. Lorsque le signal *sel* vaut 0 le multiplexeur sélectionne la première entrée. Par exemple, MUX1 sélectionne le signal s_1 sur les intervalles $[1, 5]$ et $[11, 15]$, et le signal s_6 sur l'intervalle $[6, 10]$.

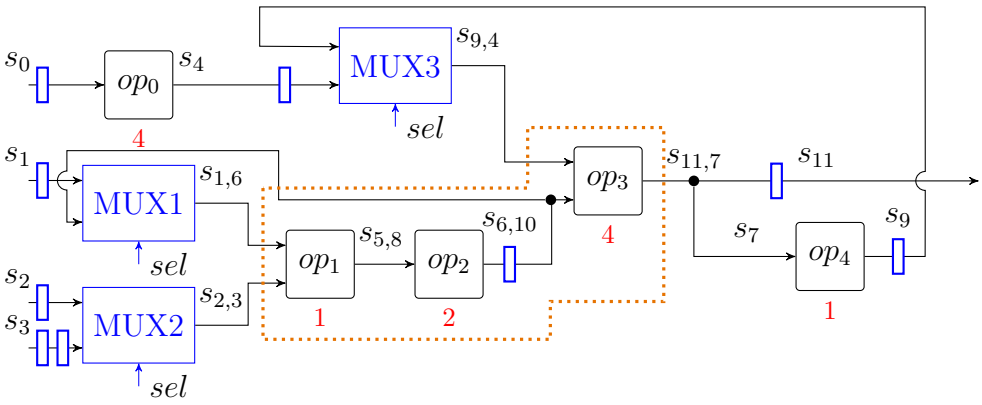


FIGURE 4.13 – Circuit C_2 plié

Comme précédemment, pour un certain motif de pliage, il est possible de construire plusieurs pipelines permettant le pliage. Nous souhaitons obtenir celui qui minimise le nombre de bascules consommées. Le problème de la synthèse de pipeline pliant optimal

s'énonce donc de la manière suivante :

Problème 2 (Synthèse de pipeline pliant optimal)**.** *Étant donné un circuit synchrone et un motif de pliage, construire un pipeline permettant le pliage et minimisant les bascules consommées.*

4.2 Synthèse de pipeline

Dans cette section, nous présentons des approches classiques de la littérature pour résoudre le problème de pipeline optimal, c'est-à-dire le problème de la synthèse de pipeline minimisant les bascules consommées et assurant une fréquence de fonctionnement minimale.

4.2.1 Bibliographie

Le problème de pipeline optimal a été initialement formalisé par Leiserson et Saxe dans [LS91]. Leur approche part d'un circuit préalablement pipeliné (avec l'algorithme glouton présenté dans la Section 4.2.2 par exemple) et le modélise par un graphe (doublement) pondéré. Ils introduisent l'opération de *retiming*, qui permet de déplacer des registres dans le circuit sans altérer son comportement fonctionnel. En se basant sur cette opération, ils reformulent le problème de pipeline optimal en un problème de flot à coût minimum et propose donc une solution. Ce résultat est amélioré plus tard par les auteurs de [HMB07] qui reformulent en un problème de flot maximum itératif. Cette solution s'avère bien plus efficace pour des circuits de grandes tailles. Elle est implémentée au niveau de la synthèse logique ou synthèse Register Transfer Level (RTL), dans l'outil ABC [Ber], qui est à notre connaissance l'état de l'art actuel.

Ces algorithmes basés sur le retiming sont principalement mis en œuvre au niveau de la synthèse logique dans les outils des fournisseurs de FPGA (par exemple *Xilinx Vivado*). L'inconvénient de cette approche est qu'elle ne modélise pas la propagation des données dans le circuit. Le problème de pliage, c'est-à-dire le problème de synthèse d'un pipeline permettant le pliage, devient alors complexe à résoudre. De plus, cette approche nécessite la construction d'un pipeline au préalable. Ceci n'est pas une forte contrainte, comme nous allons le voir par la suite.

4.2.2 Algorithme glouton

Dans cette section, nous nous intéressons à la construction d'un pipeline assurant une fréquence de fonctionnement minimale $f = \frac{1}{T}$ pour un circuit synchrone. L'algorithme présenté est un algorithme glouton « dès que possible » qui ne vise pas l'optimal en terme de minimisation des bascules consommées par le pipeline. On trouve cependant des cas concrets d'utilisation dans des outils, comme FloPoCo [ID17], du fait de sa simplicité d'implémentation et de sa faible complexité algorithmique.

FloPoCo [DP11] est un outil de synthèse d'opérateurs arithmétiques (en particulier) sur des flottants avec une précision arbitraire. Il génère des circuits pipelinés (décrits en VHDL) à partir d'informations sur la cible FPGA et d'une demande de fréquence minimale de fonctionnement.

Pour expliquer le fonctionnement de l'algorithme glouton, les auteurs de [ID17] s'appuient sur un graphe des dépendances des signaux du circuit. Ici, nous nous baserons sur le modèle de graphe pondéré tel qu'introduit par Leiserson et Saxe. L'algorithme présenté ici est donc légèrement différent, mais le principe général reste le même.

Le circuit synchrone est modélisé par un graphe dirigé doublement pondéré $G = (V, E, d, w)$. Dans lequel, les nœuds V et les arêtes E représentent respectivement les opérateurs et les signaux du circuit. À chaque nœud $v \in V$ est attribué un poids fixe $d(v)$ qui représente le temps de propagation dans l'opérateur et à chaque arête $e \in E$ est attribué un poids $w(e)$ qui représente le nombre de registres que traverse le signal. Initialement, toutes les arêtes $e \in E$ ont un poids nul $w(e) = 0$, l'objectif est de déterminer les valeurs de ces poids. De plus, est ajouté un nœud supplémentaire $v_h \in V$ depuis lequel partent tous les signaux d'entrée et arrivent tous les signaux de sortie du circuit.

Reprenons l'exemple de circuit C_1 dessiné dans la Figure 4.14. Ce dernier est modélisé par le graphe $G_1 = (V, E, d, w)$ dessiné dans la Figure 4.15, dans lequel les nœuds v_i et les arêtes e_j représentent respectivement les opérateurs op_i et les signaux s_j. À noter que le signal s_1 est représenté par trois arêtes e_{11}, e_{12} et e_{13} qui correspondent aux trois branchements dans le circuit. Enfin, les poids d et w sont représentés respectivement en rouge et en bleu.

L'idée de l'algorithme glouton est d'évoluer dans le graphe depuis les entrées jusqu'aux sorties, de cumuler les délais des opérateurs, et dès que le délai cumulé dépasse la période de fonctionnement objectif T de placer un registre. Pour ce faire, un *timing* (c, τ) est associé à chaque arête $e \in E$ du graphe. Le nombre de cycles c correspond au nombre de registres sur le chemin le plus long depuis un signal d'entrée jusqu'au signal représenté par

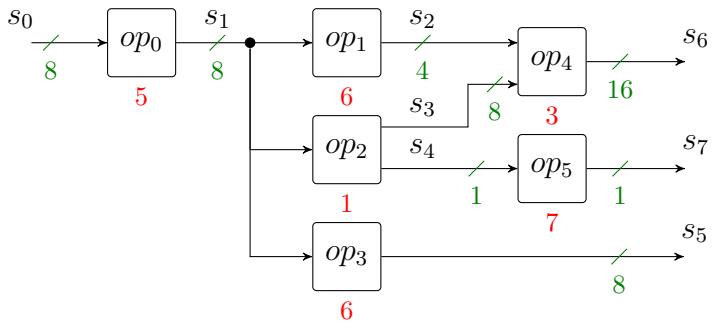


FIGURE 4.14 – Exemple de circuit C_1

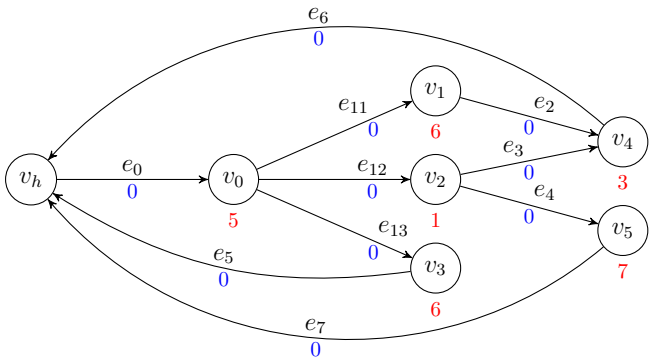


FIGURE 4.15 – Graphe $G_1 = (V, E, d, w)$ modélisant le circuit C_1 avant construction du pipeline

e. τ est le *chemin critique*, c'est-à-dire les délais des opérateurs cumulés sur le chemin, depuis le dernier registre (ou signal d'entrée) jusqu'à e. Lorsqu'un nouveau délai δ est ajouté, la formule suivante est appliquée au timing :

$$(c, \tau) + \delta = \left(c + \left\lfloor \frac{\tau + \delta}{T} \right\rfloor, \tau \cdot \left(1 - \left\lfloor \frac{\tau + \delta}{T} \right\rfloor \right) + \delta \right)$$

Notons que cette formule ne fonctionne que si $\left\lfloor \frac{\tau+\delta}{T} \right\rfloor \in \{0, 1\}$. Sinon, dans le cas où $\left\lfloor \frac{\tau+\delta}{T} \right\rfloor > 1$, cela signifie que le délai ajouté δ est plus grand que la période objectif T et le pipeline n'est pas faisable.

L'Algorithme 8 présente une version de l'algorithme glouton. Les timings de toutes les arêtes sont initialisés à $(+\infty, +\infty)$, sauf pour les arêtes succédant v_h qui sont initialisées à $(0, 0)$. Une liste d'attente *waiting* contient toutes les arêtes en attente de traitement. À chaque tour de boucle, une arête (v, v') est sélectionnée et le maximum des timings (ordre

lexicographique) de ses prédécesseurs (v_p, v) est calculé. S'il est différent de $(+\infty, +\infty)$, c'est-à-dire que tous les prédécesseurs ont un timing attribué, alors le timing de (v, v') est calculé et ses successeurs (v', v_s) sont ajoutés à la liste d'attente. Sinon l'arête retourne dans la liste d'attente. Les successeurs du nœud v_h ne sont jamais ajoutés dans la liste d'attente, pour éviter une boucle infinie.

Algorithme 8. *Algorithme de construction de pipeline glouton*

1: $\forall e \in E$, $timing[e] \leftarrow (+\infty, +\infty)$
2: $waiting \leftarrow \{\}$
3: ***for all*** $(v_h, v) \in E$ ***do***
4: $\quad timing[(v_h, v)] \leftarrow (0, 0)$
5: $\quad$ ***for all*** $(v, v') \in E$ ***do***
6: $\quad\quad waiting \leftarrow waiting \cup \{(v, v')\}$
7: $\quad$ ***end for***
8: ***end for***
9: ***while*** $waiting \neq \emptyset$ ***do***
10: $\quad$ *select and remove* (v, v') *from waiting*
11: $\quad (c, \tau) \leftarrow \max_{(v_p, v) \in E} (timing[(v_p, v)])$
12: $\quad$ ***if*** $(c, \tau) \neq (+\infty, +\infty)$ ***then***
13: $\quad\quad (c', \tau') \leftarrow (c, \tau) + d(v)$
14: $\quad\quad$ ***if*** $v' \neq v_h$ ***then***
15: $\quad\quad\quad$ ***for all*** $(v', v_s) \in E$ ***do***
16: $\quad\quad\quad\quad waiting \leftarrow waiting \cup \{(v', v_s)\}$
17: $\quad\quad\quad$ ***end for***
18: $\quad\quad$ ***end if***
19: $\quad$ ***else***
20: $\quad\quad waiting \leftarrow waiting \cup \{(v, v')\}$
21: $\quad$ ***end if***
22: ***end while***

La Figure 4.16 représente l'algorithme glouton en cours d'exécution sur le graphe G_1. La période objectif pour le pipeline est 8. Les valeurs de la liste des timings sont représentées en violet sous les arêtes. Un timing a déjà été attribué à toutes les arêtes, exceptées e_6 et e_7. Pour le calcul du timing de e_7, il suffit d'ajouter le délai de v_5 au timing de e_4 : $(0, 6) + 7$. Or, on dépasse la période objectif de 8, donc on passe au cycle

suivant, on obtient le timing $(1, 7)$. Enfin, pour calculer le timing de e_6, il faut prendre le maximum (lexicographique) entre les timings de e_2 et e_3. Puis, on ajoute le délai de v_4, ce qui nous donne pour e_6 le timing $(1, 6) + 3 = (2, 3)$.

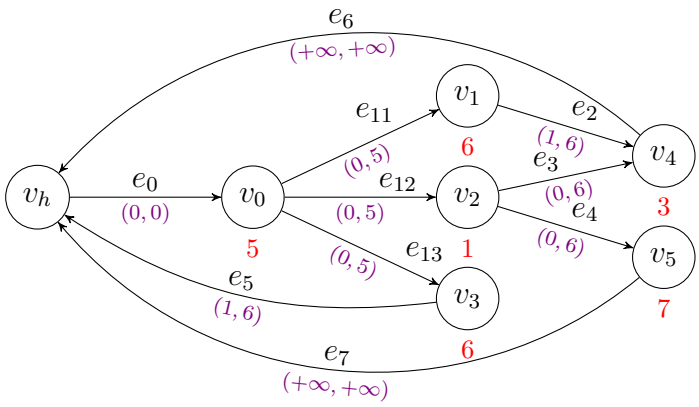


FIGURE 4.16 – Construction du pipeline de période 8 sur le graphe G_1 avec l'algorithme glouton

Notons que si la sélection de l'arête dans la liste d'attente n'est pas déterministe, l'algorithme peut boucler indéfiniment sur une même arête. Pour résoudre ce problème, les arêtes peuvent être triées et sélectionnées par ordre lexicographique de leur timing, comme le propose les auteurs de [ID17].

Enfin, notons que si le circuit possède une boucle, l'algorithme ne fonctionne pas. En fait, il est impossible de construire un pipeline sur un circuit possédant une boucle si cette dernière a un chemin critique supérieure à la période de fonctionnement objectif T. Il est impossible d'ajouter un registre dans une boucle sans altérer le comportement fonctionnel du circuit. Par contre, si la boucle a un chemin critique inférieur à la période de fonctionnement objectif, elle peut être considérée comme un seul opérateur dont le délai est le chemin critique de la boucle.

Dans la pratique, on peut souvent considérer les circuits synchrones comme sans boucle. Lorsqu'ils en ont, elles contiennent des *registres fonctionnels*, c'est-à-dire des registres cadencés à une fréquence différente de celle du pipeline, typiquement des registres à la fréquence d'échantillonnage pour une application de traitement de signal. Or, dans nos cas d'étude la fréquence d'échantillonnage est largement inférieure à celle du pipeline. Ces registres fonctionnels peuvent alors être considérés comme des entrées/sorties du système. Ceci sera expliqué plus en détail dans les expériences du Chapitre 5.

4.2.3 Leiserson & Saxe

Leiserson et Saxe proposent une solution au problème de pipeline optimal dans [LS91]. Ils se basent sur un pipeline déjà existant qui assure la fréquence de fonctionnement minimale. Ce dernier peut par exemple être synthétisé à partir d'un algorithme glouton comme vu précédemment.

Ils modélisent le circuit par un graphe doublement pondéré $G = (V, E, d, w)$, de la même manière que nous l'avons fait pour présenter l'algorithme glouton. Cependant, cette fois ci, les valeurs de w sont initialisées en fonction du pipeline initial.

Reprenons l'exemple de circuit C_1 avec un pipeline P_1 généré à partir d'un algorithme glouton dessiné dans la Figure 4.17. Ce dernier est modélisé par le graphe $G'_1 = (V, E, d, w')$ dessiné dans la Figure 4.18.

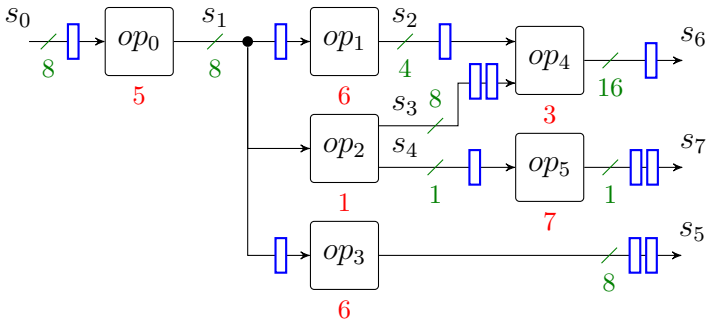


FIGURE 4.17 – Exemple de circuit C_1 avec pipeline P_1

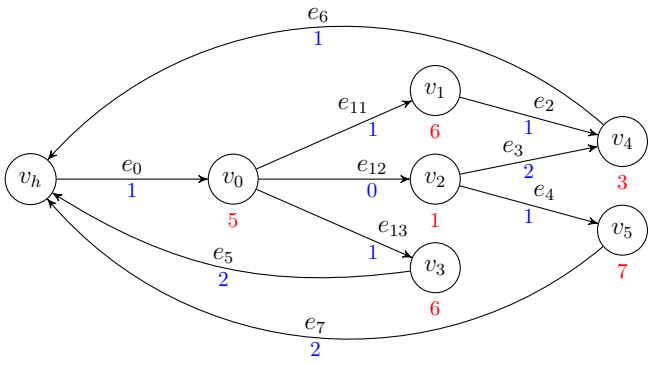


FIGURE 4.18 – Graphe $G'_1 = (V, E, d, w')$ modélisant le circuit C_1 avec pipeline P_1

Remarquons que, dans un premier temps, les tailles des signaux ne sont pas modélisées.

Ils introduisent ensuite la notion de *retiming*, qui consiste à déplacer les registres dans le circuit sans altérer son comportement fonctionnel. Un *retiming* r associe à chaque nœud du graphe un retard $r(v) \in \mathbb{Z}$, qui correspond au nombre de cycles ajoutés (ou supprimés si $r(v) < 0$) avant l'exécution de l'opérateur. Le graphe obtenu à partir de G auquel est appliqué le retiming r est le graphe $G_r = (V, E, d, w_r)$, tel que $\forall e = (u, v) \in E$, $w_r(e) = w(e) + r(v) - r(u)$. Cette équation assure le maintien de la synchronisation dans le circuit. Un retiming est légal s'il vérifie $\forall e = (u, v) \in E$, $w(e) \geq r(u) - r(v)$, c'est-à-dire s'il n'engendre pas un nombre négatif de registres ($w_r(e) \geq 0$).

Considérons dans un premier temps que l'ensemble des signaux du circuit sont sur 1 bit. Le nombre total de bascules du pipeline est donc $S(G) = \sum_{e \in E} w(e)$. Après application du retiming, le nombre total de bascules devient :

$$\begin{aligned} S(G_r) &= \sum_{(u,v) \in E} w_r((u,v)) \\ &= \sum_{(u,v) \in E} (w((u,v)) + r(v) - r(u)) \\ &= S(G) + \sum_{(.,v) \in E} r(v) - \sum_{(u,.) \in E} r(u) \\ &= S(G) + \sum_{v \in V} r(v)(FI(v) - FO(v)) \end{aligned}$$

où $FI(v)$ (resp. $FO(v)$) est le nombre d'arêtes entrantes (resp. sortantes) de v (*Fan-In* et *Fan-Out*).

Minimiser $S(G_r)$ revient alors à minimiser la quantité $\sum_{v \in V} r(v)(FI(v) - FO(v))$ qui est une combinaison linéaire des $r(v)$, car $FI(v) - FO(v)$ est constant pour chaque nœud v. S'ajoute à cela l'ensemble des contraintes $\forall e = (u, v) \in E$, $w(e) \geq r(u) - r(v)$. On obtient alors un problème d'optimisation linéaire.

Dans l'objectif d'ajouter la contrainte de fréquence minimale de fonctionnement du pipeline, Leiserson et Saxe introduisent deux objets (définis pour deux nœuds quelconques du graphe u et v) :

- $W(u, v)$ le nombre de registres minimal sur tous les chemins du graphe allant de u à v, et
- $D(u, v)$ la latence maximale parmi tous les chemins du graphe allant de u à v et possédant un nombre de registres égal à $W(u, v)$.

Ils démontrent que pour assurer une fréquence minimale de fonctionnement $f = \frac{1}{T}$ dans

le circuit retimé G_r, il suffit de vérifier que pour tous nœuds $u, v \in V$, si $D(u,v) > T$ alors $r(u) - r(v) \leq W(u,v) - 1$. Les quantités $W(u,v)$ et $D(u,v)$ étant fixes (elles sont propres au graphe initial G), cela ajoute simplement de nouvelles contraintes linéaires au problème.

D'autre part, ils démontrent que la taille des signaux peut être simplement prise en compte en appliquant un coût $\beta(e)$ à chaque arête du graphe e proportionnel à la taille du signal qu'il représente. L'objectif du problème d'optimisation n'est plus de minimiser $\sum_{v \in V} r(v)(FI(v) - FO(v))$, mais devient alors minimiser la quantité
$\sum_{v \in V} r(v)(\sum_{e=(.,v) \in E} \beta(e) - \sum_{e=(v,.) \in E} \beta(e))$.

Enfin, ils sont capables de modéliser le partage du registre entre les opérateurs utilisant le même signal. Pour ce faire, il ajoute au graphe des nœuds et des arêtes factices. Nous ne rentrerons pas dans les détails de cette modélisation ici.

Comme annoncé précédemment, Leiserson et Saxe indiquent que ce problème est en fait équivalent à un problème de flot à coût minimum. Mais nous ne rentrerons pas dans les détails de cette équivalence, car dans l'implémentation que nous avons réalisé de leur approche nous nous sommes contenté de résoudre le problème d'optimisation linéaire.

4.3 Synthèse de pipeline à partir d'un modèle RTPN

Dans cette section, nous présentons notre approche pour résoudre le problème du pipeline optimal, c'est-à-dire la synthèse d'un pipeline assurant une fréquence de fonctionnement minimale et minimisant les bascules consommées. Elle s'appuie sur une modélisation du circuit synchrone par un Réseau de Petri Temporisé avec reset et transitions retardables (RTPN) et déduit le pipeline à partir de l'application de sa sémantique.

4.3.1 Bibliographie

Comme l'ont introduit Leiserson et Saxe, un circuit synchrone peut-être modélisé par un graphe dirigé pondéré. L'intuition est en fait un Graphe de Marquage (aussi appelé Graphe d'Événement) qui est une sous-classe des Réseaux de Petri dans laquelle chaque place possède un seul arc entrant et un seul arc sortant. En raison de leur nature concurrente, les Réseaux de Petri ont été largement utilisés pour analyser et optimiser des propriétés temporelles de circuits synchrones et asynchrones : [Buf+07 ; Cam+92 ; NB13 ; SB06].

Ils se sont avérés particulièrement efficaces dans les systèmes insensibles à la latence, c'est-à-dire les systèmes dont le comportement fonctionnel ne dépend pas de la latence de chacun des sous-systèmes. Bufistov et al. [Buf+07] étendent les travaux de Leiserson et Saxe sur les systèmes insensibles à la latence, en combinant le *retiming* et le *recycling*, c'est-à-dire l'insertion de bulles (registres sans valeur informative), afin de réduire le nombre total de registres tout en garantissant un débit minimum. Plus récemment, Josipovic et al. [Jos+20] proposent une optimisation temporelle des circuits générés à partir d'une description High Level Synthesis (HLS) avec des structures de *flot de contrôle*. Ils extraient des sous-circuits *choice-free* (sans structure de contrôle) et leur appliquent l'approche de Bufistov et al. [Buf+07].

Des progrès ont également été réalisés sur les circuits asynchrones, par le biais du *slack matching* qui consiste à insérer un registre tampon pour éviter les attentes. Najibi et al. [NB13] concentrent leur travail sur les circuits asynchrones conditionnels avec plusieurs modes de fonctionnement et une probabilité d'utiliser chaque mode. Ils décomposent le problème du *slack matching* de tels systèmes en Chaînes de Markov pour le changement de mode et en Réseaux de Petri pour le placement des tampons.

Tous ces travaux partagent la même approche de résolution : déduire les contraintes temporelles de la structure du Réseau de Petri, et se ramener à un problème d'optimisation linéaire. Notre approche consiste au contraire à utiliser la sémantique des PNs et synthétiser le pipeline directement à partir de ses états. L'exploration explicite des états offre ainsi la possibilité de vérifier des propriétés logiques ou temporelles sur le circuit pipeliné produit.

4.3.2 Modélisation d'un circuit synchrone par un RTPN

Le circuit est modélisé par un RTPN dont les transitions représentent les opérateurs et les places représentent les signaux. Le marquage du réseau représente les positions des registres de l'étage courant du pipeline, l'action reset fixe les positions d'un étage. Un pipeline complet du circuit est construit à partir d'une exécution du RTPN. Parmi toutes les exécutions du modèle, nous pourrons choisir celle qui engendre un pipeline optimal.

Pour la construction du pipeline, nous visons la minimisation du nombre total de bascules (registre 1 bit), et nous étendons donc notre modèle avec des coûts qui représentent le nombre de bascules d'un pipeline donné. Rappelons que les circuits considérés sont finis avec des boucles dépliées, nous nous concentrons alors uniquement sur les exécutions finies des RTPNs : l'ajout d'un coût uniquement croissant n'affectera pas la terminaison.

RTPN à coût La classe RTPN est étendue avec un coût associé à chaque place et une fonction de coût pour un marquage du réseau.

Définition 29 (CRTPN)**.** *Un RTPN à coût (CRTPN) est un tuple* $(\mathcal{N}, \mathcal{C}, \omega)$ *où* $\mathcal{N} = (P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\mathsf{reset}}, M_0)$ *est un RTPN et*

— $\mathcal{C} \colon P \to \mathbb{N}$ *représente le coût associé à chaque place ;*

— $\omega \colon \mathbb{N}^P \to \mathbb{N}$ *est la fonction de coût de marquage (un marquage* $M \in \mathbb{N}^P$*).*

Une fonction classique de marquage est $\omega(M) = \sum_{p \in P} M(p) \cdot \mathcal{C}(p)$ qui est la somme des marquages pondérés par les coûts. Mais cette fonction n'est pas nécessairement linéaire, comme nous le verrons dans la modélisation des branchements (lorsqu'un signal du circuit est utilisé par plusieurs opérateurs).

Nous définissons de plus le coût d'une exécution, comme étant le coût cumulé des marquages à la suite d'un reset.

Définition 30 (Coût d'une exécution)**.** *Le coût* $\Omega(\rho)$ *d'une exécution* ρ *est le coût de marquage cumulé des états immédiatement après chaque transition* reset*, en commençant par le coût du marquage initial.*

Elle est définie inductivement sur une exécution $\rho_n = \rho_{n-1} \xrightarrow{\alpha_n} q_n$*, avec* $\alpha_n \in \mathbb{R}_{+*} \cup 2^T \cup \{\mathsf{reset}\}$ *et* $q_n = (M_n, v_n, \chi_n) \in Q$ *par :*

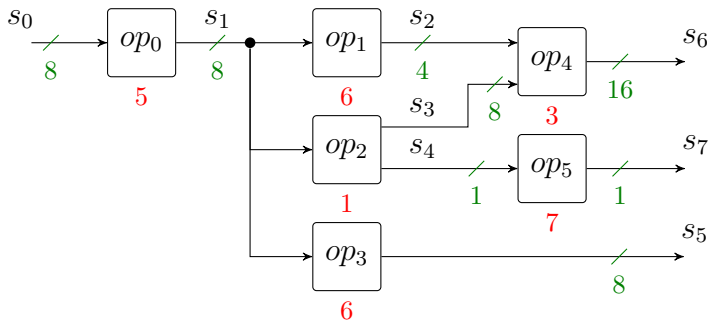
— $\Omega(q_0) = \omega(M_0)$

— $\Omega(\rho_n) = \begin{cases} \Omega(\rho_{n-1}) + \omega(M_n) & \text{si } \alpha_n = \{\mathsf{reset}\} \\ \Omega(\rho_{n-1}) & \text{sinon} \end{cases}$

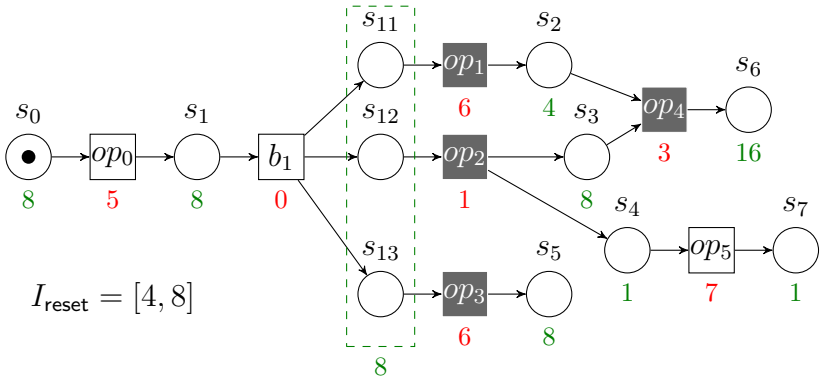
Du circuit vers le modèle Pour modéliser un circuit synchrone avec un RTPN à coût, nous nous appuyons sur un ensemble de *règles*.

Intéressons-nous encore une fois à l'exemple de circuit C_1 dessiné dans la Figure 4.19a.

Dans la suite, un circuit est considéré comme étant un graphe dirigé pondéré (V, E, d, σ), dans lequel $V = Op \uplus B$ est l'ensemble des opérateurs Op joint à l'ensemble des points de branchements B, et E est l'ensemble des signaux. À la suite d'un point de branchement, des signaux supplémentaires sont définis pour représenter les arêtes du graphe (sur l'exemple de la Figure 4.19a le signal s_1 donne trois signaux s_{11}, s_{12} et s_{13} après le branchement). Enfin, les poids d et σ représentent respectivement les temps de propagation des opérateurs $d(op)$ et les tailles des signaux $\sigma(s)$ (nombre de bits).



(a) Circuit C_1



(b) Modèle CRTPN $\mathcal{N}_1$ du circuit C_1

FIGURE 4.19 – Modélisation d'un circuit avec un CRTPN

Le CRTPN $((P, T, T_D, {}^\bullet(.), (.)^\bullet, \delta, I_{\text{reset}}, M_0), \mathcal{C}, \omega)$ produit à partir du circuit de la Figure 4.19a est représenté sur la Figure 4.19b et est obtenu *via* 7 règles. Les quatre premières règles permettent d'assurer la préservation des éléments du circuit, ainsi que les interconnexions :

règle 1 : $\exists \phi_e \colon E \mapsto P$ une bijection, avec $\forall s \in E$, $\mathcal{C}(\phi_e(s)) = \sigma(s)$;

règle 2 : $\exists \phi_v \colon V \mapsto T$ une bijection, avec $\forall op \in Op$, $\delta(\phi_v(op)) = d(op)$ et $\forall b \in B$, $\delta(\phi_v(b)) = 0$;

On notera $T = T_{Op} \uplus T_B$ avec $T_{Op} = \phi_v(Op)$ et $T_B = \phi_v(B)$.

règle 3 : Si $s \in E$ est un signal entrant de $v \in V$, alors ${}^\bullet t(p) = 1$ avec $t = \phi_v(v)$ et $p = \phi_e(s)$;

règle 4 : Si $s \in E$ est un signal sortant de $v \in V$, alors $t^\bullet(p) = 1$ avec $t = \phi_v(v)$ et $p = \phi_e(s)$;

Un signal et sa taille sont respectivement modélisés par une place et le coût associé. Dans la Figure 4.19b, les coûts sont représentés en vert en dessous des places. Un opérateur et son délai de propagation sont modélisés par une transition et le délai de tir de celle-ci.

De plus, à chaque point de branchement est associé une transition avec un délai de tir nul (b_1 dans cet exemple). Son objectif est de permettre le placement d'un étage de pipeline soit avant le branchement (s_1), soit sur une branche particulière de sortie (s_{11}, s_{12} ou s_{13}). Les règles 3 et 4 définissent les fonctions d'incidence qui préservent la structure du réseau.

Tous les signaux d'entrée sont considérés synchrones, ce qui équivaut à les avoir tous sur le premier étage du pipeline. Dans le modèle, ceci s'exprime par le marquage initial M_0 et est lié à la règle 5 :

règle 5 : Si s est un signal d'entrée (ne sortant d'aucun opérateur), alors $M_0(p) = 1$ sinon $M_0(p) = 0$, avec $p = \phi_e(s)$;

L'opération reset correspond au passage d'un étage de pipeline au suivant et réinitialise ainsi les horloges du CRTPN pour l'étage suivant. La règle 6 définit la limite maximale de l'intervalle de réinitialisation :

règle 6 : $\overline{I_{\mathsf{reset}}} = \frac{1}{f}$;

Le temps écoulé depuis la dernière réinitialisation est stocké dans $v(\mathsf{reset})$. La sémantique impose qu'une réinitialisation ne peut se produire que si $v(\mathsf{reset}) \in I_{\mathsf{reset}}$, et donc si la borne maximale est fixée à $\frac{1}{f}$, alors le pipeline produit peut fonctionner au moins à la fréquence f. Ici $\frac{1}{f}$, et dans la suite $\frac{1}{2f}$, sont supposés être dans $\mathbb{N}$, mais ils peuvent être rationnels sans changer les résultats (il suffit d'appliquer un facteur à tous les délais du modèle pour se ramener à des valeurs entières).

La fonction de coût donne le nombre total de bascules nécessaires dans l'étage courant du pipeline :

règle 7 : On définit $P_{Op} = \{p \in P \mid \exists t \in T_{Op}, t^\bullet(p) = 1\}$ et $P_B(p) = \{p' \in P \mid \exists t \in T_B, {}^\bullet t(p) = 1 \text{ et } t^\bullet(p') = 1\}$.
Alors $\forall M \in \{0,1\}^P$, $\omega(M) = \sum_{p \in P_{Op}} \mathcal{C}(p) \cdot \max(M(p), \max_{p' \in P_B(p)}(M(p')))$.

En effet, le coefficient de coût d'une place correspond à la taille du signal, et par conséquent indique le nombre de bascules nécessaires. Le calcul du coût prend en compte le cas particulier des points de branchement et la mutualisation éventuelle des registres en sortie d'un point de branchement. C'est pourquoi le coût des places après une transition t modélisant un branchement à la suite de la place p est $\mathcal{C}(p) \cdot \max_{p' \in P_B(p)}(M(p'))$. Le coût partagé entre les places $\phi_e(s_{11})$, $\phi_e(s_{12})$ et $\phi_e(s_{13})$ est représenté dans la Figure 4.19b par un rectangle vert en pointillé dont le coût est défini en dessous.

Enfin, dans un premier temps toutes les transitions du réseau sont définies comme étant retardables : $T = T_D$.

Ces règles de traduction sont suffisantes pour définir le modèle du circuit. Celui-ci va permettre de déterminer l'ensemble des pipelines possibles, et par conséquent trouver le pipeline optimal, c'est-à-dire celui qui minimise les ressources tout en assurant une fréquence minimale de fonctionnement. Cependant, en pratique, nous sommes rapidement confrontés à une explosion combinatoire lors de l'évaluation de l'espace d'état. Des heuristiques sont proposées dans la suite pour limiter le nombre de transitions retardables et pour définir une borne inférieure de l'intervalle de reset. Ceci explique pourquoi dans la Figure 4.19b, les opérateurs op_0 et op_5 ne sont pas associés à des transitions retardables et l'intervalle de reset a une borne inférieure non nulle.

4.3.3 Construction du pipeline

Notre approche se base sur la sémantique des CRTPNs pour synthétiser le pipeline du circuit. Nous montrons ici comment le pipeline est généré à partir d'exécutions du modèle.

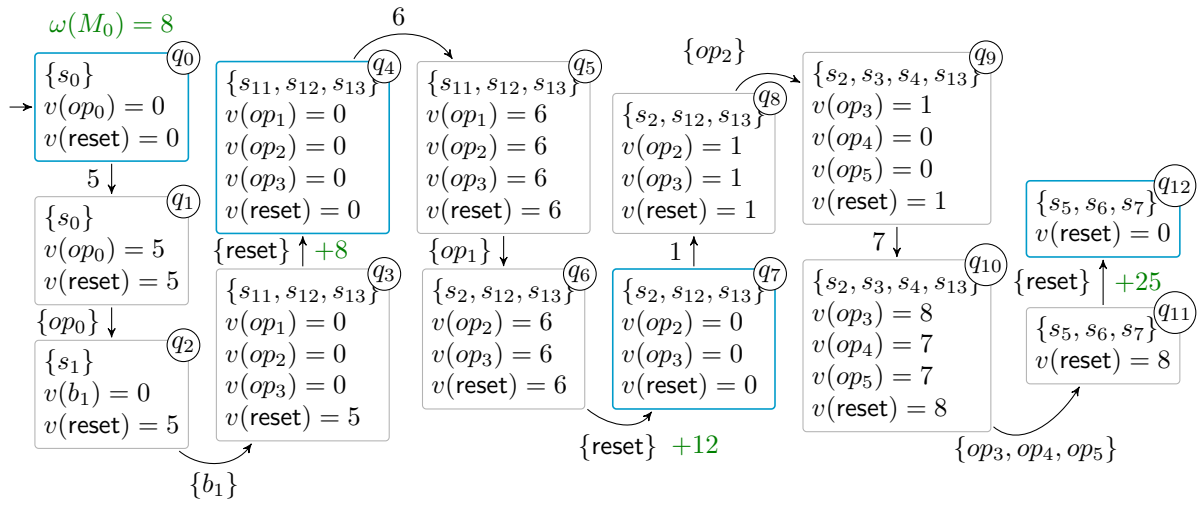
Synthèse du pipeline Remarquons dans un premier temps que la sémantique des CRTPNs assure le maintien de la synchronisation des signaux dans le circuit. En effet, un opérateur du circuit décompte son temps de calcul dès lors que tous les signaux en amont sont disponibles, ce qui se traduit dans le modèle par la sensibilisation des transitions à partir du moment où les places en amont ont un jeton.

Le marquage, c'est-à-dire la position des jetons dans le réseau, représente le placement des registres dans le circuit. Chaque état atteignable du modèle représente un étage de pipeline possible du circuit réel. La valeur d'horloge du reset représente le temps écoulé dans l'étage de pipeline courant. L'action reset définit le passage d'un étage de pipeline au suivant, c'est-à-dire le placement d'un ensemble de registres fixant les limites d'un étage. Le pipeline complet est construit en parcourant une branche du graphe d'état et en accumulant les opérations de reset.

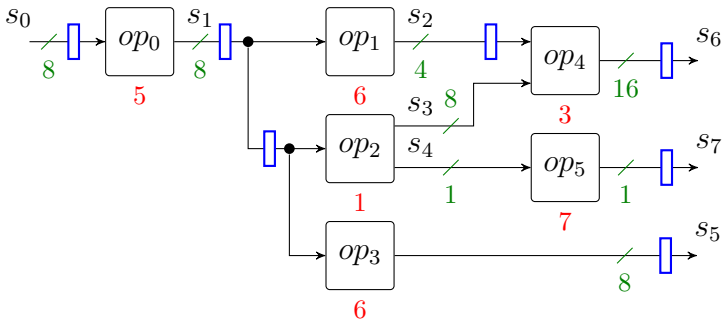
L'exécution optimale (avec le coût minimal) ρ_1 du CRTPN $\mathcal{N}_1$ est dessinée dans la Figure 4.20a. Elle est enrichie de l'évolution de son coût cumulé, en vert.

Le pipeline synthétisé à partir de cette exécution est dessiné dans la Figure 4.20b. Les registres délimitant les étages de ce pipeline correspondent aux marquages de l'exécution qui succèdent chaque reset. Dans la Figure 4.20a les états succédant un reset sont encadrés en cyan.

Notons M_i le marquage de chaque état q_i. Le coût de l'exécution ρ_1 se calcule de la



(a) Exécution optimale ρ_1 du CRTPN $\mathcal{N}_1$



(b) Pipeline correspondant du circuit C_1

FIGURE 4.20 – Synthèse d'un pipeline avec un CRTPN

manière suivante :

$$\begin{aligned}\Omega(\rho_1) &= \omega(M_0) + \omega(M_4) + \omega(M_7) + \omega(M_{12}) \\ &= (\mathcal{C}(s_0)) + (\mathcal{C}(s_1)) + (\mathcal{C}(s_1) + \mathcal{C}(s_2)) + (\mathcal{C}(s_5) + \mathcal{C}(s_6) + \mathcal{C}(s_7)) \\ &= (8) + (8) + (8+4) + (8+16+1) = 53\end{aligned}$$

Le coût du marquage M_4 est obtenu par $\omega(M_4) = \mathcal{C}(s_1) \cdot \max(M_4(s_1), M_4(s_{11}), M_4(s_{12}), M_4(s_{13})) = \mathcal{C}(s_1)$, et de même pour le marquage M_7.

Heuristiques Le modèle CRTPN produit par les règles définis précédemment, dans lequel toutes les transitions sont retardables, permet de synthétiser le pipeline optimal,

c'est-à-dire celui qui minimise les bascules tout en assurant une fréquence de fonctionnement minimale f. Cependant en pratique, cette construction conduit à une explosion de l'espace d'états. Nous proposons donc des heuristiques de construction qui se concentrent sur des points essentiels pour économiser des ressources, tout en réduisant considérablement la taille de l'espace d'états.

Les transitions retardables permettent de relaxer les contraintes du modèle et ainsi d'explorer plus d'états (et donc plus de placement de registres). Nous proposons deux utilisations de ces transitions retardables.

heuristique 1 : $\forall t \in T_{Op}$, si $\sum_{p \in P} \mathcal{C}(p) \cdot {}^\bullet t(p) < \sum_{p \in P} \mathcal{C}(p) \cdot t^\bullet(p)$, alors $t \in T_D$.

Lorsqu'un opérateur a un bus d'entrée plus large que son bus de sortie, il peut être avantageux de placer les registres sur son bus d'entrée plutôt que son bus de sortie. Traduire cet opérateur par une transition retardable permet d'explorer cette possibilité. En effet, au cours d'une exécution les jetons peuvent être conservés en amont d'une transition retardable, jusqu'au prochain reset. Dans l'exemple de la Figure 4.19b, les opérateurs op_2 et op_4 sont modélisés par des transitions retardables en suivant cette heuristique.

heuristique 2 : $\forall t \in T_{Op}$, si $\exists t_B \in T_B, p \in P$ tel que
$t_B^\bullet(p) = {}^\bullet t(p) = 1$, alors $t \in T_D$.

Les opérateurs qui succédent un point de branchement peuvent mutualiser leurs registres (ils utilisent le même signal) dans l'objectif d'économiser des bascules. Ainsi traduire ces opérateurs par des transitions retardables permet d'explorer des pipelines dans lesquel les registres sont maintenus en amont, et donc offre la possibilité de les mutualiser. Dans l'exemple de la Figure 4.19b, les opérateurs op_1, op_2 et op_3 sont modélisés par des transitions retardables en suivant cette heuristique.

L'intervalle de reset offre plus de flexibilité qu'un singleton. Il permet d'explorer d'autres configurations de pipeline avec des étages plus court (en termes de chemin critique) que la période de fonctionnement. Cependant, si les étages sont trop courts, cela a pour effet d'augmenter le nombre d'étages et par conséquent le nombre de registres. La borne inférieure de l'intervalle de reset peut être fixée à $\frac{1}{2f}$, ce qui donne :

heuristique 3 : $I_{\mathsf{reset}} = \left[\frac{1}{2f}, \frac{1}{f}\right]$

Pour comprendre l'intuition derrière cette heuristique, on peut faire une analogie avec le théorème d'échantillonage de Shannon où le reset correspond à l'échantillonage : si l'on fait un reset avant $\frac{1}{2f}$ il va y avoir un chevauchement d'états, c'est-à-dire que nous visiterons le même état (avec un reset supplémentaire) deux fois.

Enfin, lorsque nous explorons le graphe d'états d'un CRTPN, nous élagons à la volée les branches pour lesquelles le coût est moins bon. Autrement dit, si nous atteignons un état déjà visité avec un coût moins bon, nous n'explorons pas ses successeurs.

4.3.4 Résultats expérimentaux

Notre approche est évaluée par une phase expérimentale sur des circuits générés par l'outil FloPoCo [DP11]. Nous comparons les résultats obtenus par un algorithme glouton, la méthode de Leiserson et Saxe et notre approche basée sur les CRTPNs.

FloPoCo est un générateur de circuits performant des opérations arithmétiques sur nombre à virgules flottantes (entre autres). Il génère des circuits décomposés en de nombreux opérateurs élémentaires dont les délais sont estimés par des heuristiques internes. Pour les expériences, nous nous basons sur FloPoCo version 5 (version non stable sur git au moment des expériences) pour générer un circuit avec un pipeline de base assurant une fréquence de fonctionnement minimale.

Le pipeline est synthétisé par une approche « dès que possible » (ou gloutonne). Avec notre approche basée sur le modèle CRTPN, nous partons d'un circuit nu, nous n'avons pas besoin d'un pipeline préalablement synthétisé.Un pipeline initial est par contre utilisé dans l'approche de Leiserson et Saxe.

Notre échantillon est composé de quatre opérateurs sur des flottants en précision simple (8 bits pour l'exposant et 23 bits pour la mantisse) : un additionneur, un multiplieur, un diviseur et une racine carrée. Les résultats sont présentés dans le Tableau 4.1. Les opérateurs arithmétiques contiennent un grand nombre d'opérateurs (de 108 à 189) et de signaux internes (de 165 à 316) ce qui défini la granularité et la scalabilité des approches. Les fréquences objectifs sont choisies pour être proches des fréquences maximales faisables par le circuit, avec les estimations des délais fournies par FloPoCo sur une cible Xilinx Virtex 6. L'objectif est de montrer l'intérêt d'optimiser le pipeline sur des circuits fortement contraints, avec un grand nombre d'étages.

Toutes les approches produisent le même nombre d'étages de pipeline dans chaque exemple de circuit. Le pourcentage d'amélioration est évalué par rapport aux résultats fournis par l'approche gloutonne. L'approche basée sur un CRTPN réduit le nombre de bascules (notés FF pour *flip-flop*) de 12% jusqu'à 34%, et ne donne jamais un résultat pire que celui de l'approche gloutonne. C'était attendu, car l'approche gloutonne est en fait une exécution possible du CRTPN.

L'heuristique 1 ajoute des transitions retardables pour les opérateurs qui ont un bus

TABLE 4.1 – Synthèse de pipelines des opérateurs arithmétiques sur des flottants générés par FloPoCo. Résultats obtenus par les trois approches : algorithme glouton, CRTPNs et Leiserson & Saxe. Les paramètres des opérateurs sont les tailles des signaux d'entrée (exposant, mantisse). *(s,del)* donne le nombre d'états analysés et le nombre de transitions retardables. Lorsque les heuristiques 1 et 2 sont utilisées, le nombre de transitions retardables est limité manuellement.
Le nombre de bascules est mesuré sur les modèles de circuits, et non pas sur les circuits synthétisés. Les optimisations réalisées par un outil EDA ne sont donc pas prises en compte ici.

		FPAdd(8,23) 500 MHz	FPMult(8,23) 500 MHz	FPDiv(8,23) 500 MHz	FPSqrt(8,23) 500 MHz
taille circuit (ops, sigs)		(108,165)	(151,237)	(116,197)	(189,316)
Étages de pipeline		16	7	30	25
Glouton	Time (s)	0.01	0.01	0.01	0.01
	Nb FF	2080	671	3480	2085
CRTPN	Time (s)	0.06	0.02	0.10	0.06
(sans	states analysed	418	110	1711	215
retardable)	Nb FF	1999	671	3182	2082
Amélioration (%)		**3.9%**	**0.0%**	**8.6%**	**0.1%**
CRTPN	Time (s)	1.35	0.47	0.44	4.12
(avec	s - del	8368 - 8	2001 - 7	4711 - 13	15946 - 51
heuristique 1)	Nb FF	1852	437	3158	1595
Amélioration (%)		**11.0%**	**34.9%**	**9.3%**	**23.5%**
CRTPN	Time (s)	270.8	170.48	325.5	238.76
(avec	s - del	410513 - 25	223741 - 23	489423 - 55	362375 - 70
heuristiques 1+2)	Nb FF	1815	437	2816	1590
Amélioration (%)		**12.7%**	**34.9%**	**19.1%**	**23.7%**
Leiserson	Time (s)	0.83	1.90	0.76	1.49
et Saxe	Nb FF	1703	437	2599	1306
Amélioration (%)		**18.1%**	**34.9%**	**25.3%**	**37.4%**

d'entrée plus large que leur bus de sortie. Elle donne de très bons résultats, elle permet une grande réduction des bascules avec peu de transitions retardables ajoutées (à part dans le cas de l'opérateur `FPSqrt`). Ceci permet d'obtenir un pipeline rapidement avec une réduction des bascules considérable (entre 9.3% et 34.9%). L'opérateur `FPSqrt` se voit attribuer plus de transitions retardables, mais cela ne dégrade pas pour autant le temps de calcul. Ceci est dû à sa structure très séquentielle (les opérations sont réalisées les une à la suite des autres).

L'heuristique 2 permet de mutualiser les registres au niveau d'un point de branchement. Avec cette heuristique beaucoup de transitions deviennent retardables, et cela mène à une explosion du temps de calcul. Les meilleurs résultats sont obtenus en combinant l'heuristique 2 avec l'heuristique 1. Le nombre total de transitions retardables est limité à la main, ce dans l'objectif de trouver un compromis qui permet d'ajouter suffisamment de transitions retardables tout en limitant le temps de calcul à quelques minutes. Les résultats sont toujours améliorés en ajoutant l'heuristique 2, mais de manière très hétérogène. L'opérateur `FPDiv` semble bien adapté pour la mutualisation des registres et présente une réduction significative du nombre de bascules (de 3158 à 2816). Le nombre total de transitions retardables est limité en choisissant de manière arbitraire les transitions éliminées dans l'heuristique 2, ce qui peut également expliqué les écarts d'amélioration entre les exemples.

L'heuristique 3 défini la borne inférieure de l'intervalle de reset à $\frac{1}{2f}$. Elle est appliquée dans toutes les expériences, car une borne inférieure nulle (ou plus petite) augmente le temps de calcul, sans aucune amélioration.

Les trois approches basées sur un CRTPN sont à mettre en perspective avec les résultats produits par la méthode de Leiserson et Saxe. Cette dernière a été implémentée par une modélisation ILP dont la solution est donnée par le solveur Gurobi [Gur22] (en utilisant le wrapper ScaLP [Sit+18]). Comme attendu, les approches basées sur CRTPN ne donnent pas de solution optimale, sauf pour l'exemple de l'opérateur `FPMult`. En effet, les heuristiques limitent l'espace d'exploration de solutions et mènent donc (dans le cas général) à une solution sous-optimale. Cependant, malgré le fait que l'implémentation de l'approche CRTPN soit un prototype non optimisé (thread unique, structures de données naïves, ..), on constate qu'en un temps de calcul raisonnable (quelques minutes au maximum), les solutions obtenues sont proches de l'optimale (au plus 13.7 points d'écart).

Notre approche permet donc de synthétiser un pipeline en optimisant sa consommation de ressources, en un temps raisonnable. Mais le plus grand intérêt de cette nouvelle

approche est qu'elle se base sur une modélisation formelle sur laquelle de nombreux résultats existent déjà. Nous allons d'ailleurs mettre à profit cette avantage dans la Section 4.4 pour s'attaquer au problème de pliage.

4.4 Synthèse de pliage

Dans cette section, nous nous intéressons au problème de pipeline pliant optimal, c'est-à-dire le problème de synthèse de pipeline permettant le pliage et minimisant les bascules consommées. Le problème du pliage est présent dans la littérature et des avancées ont déjà été réalisés. Nous présentons ici notre approche basée sur le modèle RTPN.

4.4.1 Bibliographie

Le pliage ou multiplexage temporel consiste à fusionner des portions de circuits qui sont présentes à plusieurs reprises, puis à séquencer l'accès à la portion effectivement implémentée. Il existe en fait plusieurs variantes du pliage.

Pour commencer, le pliage peut être effectué sur des opérateurs, plutôt que des groupes d'opérateurs. Les ressources (unités logiques) économisées sont en général moins importantes qu'avec le pliage de groupes d'opérateurs, car le ratio nombre de (dé)multiplexeurs par rapport au nombre d'opérateurs pliés est plus grand. Cependant, le problème est plus facile à résoudre, car dans le cas du pliage d'un groupe d'opérateurs, il peut y avoir des signaux internes au groupe qui sont utilisés à l'extérieur du groupe, ce qui complique le placement des registres.

Un deuxième axe d'étude est le pliage de k opérateurs (ou portions de circuit) parmi n. Supposons un circuit avec n répétitions du même opérateur (ou groupe d'opérateurs), le problème consiste à n'en implémenter que k (avec $k < n$) et à répartir l'accès dans le temps à ses k implémentations entre les portions de circuits qui en ont besoin. L'intérêt de cette approche est qu'elle permet de réduire les ressources consommées, sans trop augmenter la latence du circuit. Cela peut être un bon compromis dans certains cas.

Enfin, il est possible dans le cas de pliage de portions de circuit, de plier des portions elles-mêmes déjà pliées. Autrement dit, il est possible d'avoir des motifs de pliage imbriqués. Ceci permet de réduire grandement les ressources consommées, mais complexifie considérablement le problème.

Plusieurs travaux ont été réalisés sur le multiplexage temporel, notamment sous la

forme d'un problème appelé modulo ordonnancement (*modulo scheduling*). Ce problème vise à établir une latence minimale du circuit multiplexé dans le temps, étant donné un nombre limité de ressources (opérateurs arithmétiques ou logiques) disponibles. Les auteurs de [Sit+18] proposent une formulation ILP (Integer Linear Programming) qui combine des contraintes d'ordonnancement, des limites sur les ressources disponibles, la minimisation du nombre de registres consommés par le pipeline, et la mutualisation des registres lorsque cela est possible. Plus récemment, une stratégie en deux étapes est démontrée par [Sit+19] : ils détectent d'abord les configurations mutualisables, puis ils ordonnancent chaque configuration de partage sélectionnée. Cette approche permet de partager des portions de circuit contrairement aux précédentes qui ne partageaient que les opérateurs un par un. Les approches de [Sit+18 ; Sit+19] ont été implémentées dans un outil appelé Origami dont nous reparlerons dans la Section 5.1.

Nous proposons une approche permettant le pliage de portions de circuit en minimisant le nombre de bascules consommées par le pipeline. Cette approche peut aisément être étendue pour le pliage de k portions de circuit parmi n, ainsi que pour les motifs de pliage imbriqués.

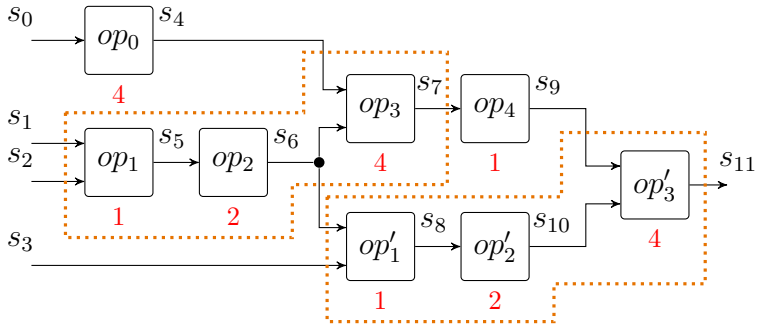
Notons que nous ne nous intéressons pas à la sélection de motifs de pliage, ce qui constitue un problème en soit qui a déjà été abordée dans la littérature sous le nom de problème d'identification automatique de sous-graphes isomorphes [Sit+17b]. Nous considérons dans notre approche que le motif de pliage a déjà été déterminé par une méthode existante. Dans notre cas d'étude, nous bénéficions du haut niveau d'abstraction de Matlab/Simulink et certaines parties des modèles évalués présentent des redondances relativement évidentes à déterminer. D'autre part, l'utilisation de bibliothèques facilite l'identification des parties partageables.

4.4.2 Synthèse de pliage avec RTPN

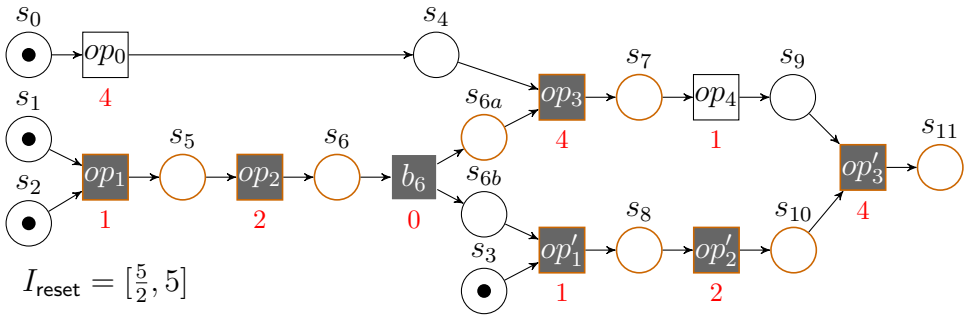
En se basant sur la synthèse de pipeline avec CRTPN, il est possible de résoudre le problème de pliage de circuit. Plus précisément, il est possible de construire un pipeline qui assure que le pliage est faisable, tout en assurant une fréquence cible et en minimisant le nombre de bascules. Pour ce faire, l'exploration des états du modèle est soumise à des contraintes supplémentaires qui garantissent que le pipeline produit permet le pliage. Ces contraintes sont exprimées via la logique LTL. Cependant, il ne s'agit pas de faire de la vérification sur le modèle, mais uniquement de sélectionner la partie de son graphe d'états vérifiant certaines propriétés. Ces propriétés concernent une exécution du modèle à la fois,

et non pas tous les branchements possibles à partir d'un état. La logique LTL semble donc plus appropriée que la logique CTL.

Sans la formaliser, nous appliquons cette solution à l'exemple de circuit dessiné dans la Figure 4.21a. Pour simplifier cette démonstration, nous ignorons les tailles des signaux et donc les coûts associés dans le modèle, mais ces derniers peuvent être intégrés comme cela a été fait dans la Section 4.3.



(a) Circuit C_2 pliable



(b) Modèle RTPN $\mathcal{N}_2$ du circuit C_2

FIGURE 4.21 – Modélisation d'un circuit pliable avec un RTPN

Pour rappel l'objectif est de construire un pipeline tel que lors de l'exécution du circuit pipeliné les portions de circuit à fusionner (encadré en pointillés oranges) ne s'exécutent pas en même temps. Autrement dit, il faut que les *opérateurs jumelés* (les opérateurs qui seront fusionnés) ne soient pas dans le même cycle du pipeline. Ce pipeline doit en fait répondre à deux contraintes :

1. La première est une propriété d'exclusion mutuelle. Elle assure que les ressources ne peuvent pas être accessibles au même moment par plusieurs demandeurs.
2. La deuxième concerne le placement des registres. Elle assure que les portions de circuits à plier possèdent des registres aux mêmes endroits.

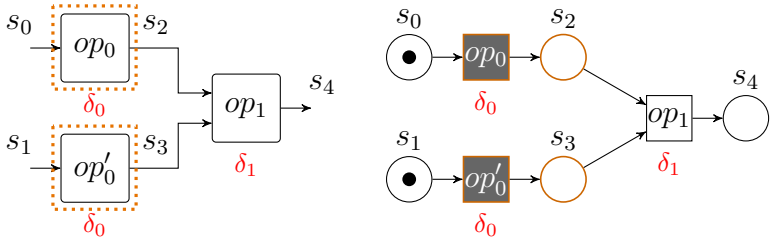
Le modèle RTPN du circuit C_2 est dessiné dans la Figure 4.21b, il est construit en utilisant les règles de modélisation données dans la Section 4.3. Notons que les transitions représentant les opérateurs partagés sont retardables pour relaxer l'exploration et permettre de satisfaire les contraintes de pliage. Les places et les transitions dessinées en orange sont celles qui seront soumises à des contraintes.

Exclusion mutuelle Pour commencer, il faut comprendre que dans la modélisation proposée dans la Section 4.3, le comportement temporel du RTPN correspond à celui du circuit pipeliné. En effet, l'ordre de tir (temps logique) et la date de tir (temps dense) des transitions correspondent à l'ordre et la date d'exécutions des opérateurs dans le circuit pipeliné. Pour modéliser l'exclusion mutuelle des opérateurs à partager, il suffit donc d'appliquer une exclusion mutuelle sur les transitions du réseau.

Une première approche consiste à contraindre le marquage des places qui sont alimentées par une transition modélisant un opérateur à partager. Sur l'exemple du RTPN $\mathcal{N}_2$, cette propriété s'écrit : $\widetilde{\phi}_{mutex} = (M(s_5)+M(s_8) \leq 1) \wedge (M(s_6)+M(s_{10}) \leq 1) \wedge (M(s_{6a})+M(s_{10}) \leq 1) \wedge (M(s_7)+M(s_{11}) \leq 1)$. Elle consiste à interdire la présence de jetons simultanément dans les places *jumelées* (les places alimentées par deux transitions modélisant deux opérateurs à partager). La présence de jeton dans ces places modélise la présence d'une donnée produite dans le circuit. La propriété $\widetilde{\phi}_{mutex}$ vérifie donc que les signaux s_5, s_6, s_{6a} et s_7 respectivement jumelés à s_8, s_{10}, s_{10} et s_{11} ne contiendront jamais des données en même temps que leur signal jumelé. Cette propriété est suffisante pour assurer l'exclusion mutuelle de l'accès aux opérateurs, c'est-à-dire pour assurer que les opérateurs partagés ne seront pas exécutés sur le même cycle.

Cependant, la propriété $\widetilde{\phi}_{mutex}$ impose en fait trop de contrainte et empêche de trouver des solutions dans certains cas particuliers. C'est notamment le cas de l'exemple du circuit C_3 dessiné dans la Figure 4.22. Dans cet exemple les opérateurs à partager sont op_0 et op'_0, la propriété d'exclusion mutuelle s'écrit alors $\widetilde{\phi}_{mutex} = (M(s_2)+M(s_3) \leq 1)$. L'exploration des états du modèle qui valident cette propriété ne permet pas de trouver d'exécutions finales, car la transition op_1 ne sera jamais tirable. Alors qu'il existe en effet un pipeline permettant le pliage : il suffit de placer un registre de plus en amont de op_0 qu'en amont de op'_0 (ou l'inverse).

Il faut donc contraindre le tir des transitions et non pas le marquage des places du réseau. Il est possible de construire un observateur qui indique si une transition vient d'être tirée en ajoutant une place observatrice $p_{obs(t)}$ en aval de chaque transition t du réseau



(a) Circuit C_3 pliable (b) Modèle RTPN $\mathcal{N}_3$ du circuit C_3

FIGURE 4.22 – Contre-exemple propriété $\widetilde{\phi}_{mutex}$

et une transition $t_{drain(t)}$ non-retardable qui consomme les jetons de $p_{obs(t)}$ en temps nul. Formellement, cela s'écrit $t^\bullet(p_{obs(t)}) = 1$, ${}^\bullet t_{drain(t)}(p_{obs(t)}) = 1$, $t_{drain(t)} \notin T_D$ et $\delta(t_{drain(t)}) = 0$. Le marquage de la place $p_{obs(t)}$ indique que la transition t vient d'être tirée dans le pas maximal précédent et le pas maximal suivant contiendra obligatoirement la transition $t_{drain(t)}$ ce qui videra la place $p_{obs(t)}$. Notons que même dans le cas où t est tirée deux fois de suite en temps nul, comme $t_{drain(t)}$ n'est pas retardable et a un délai nul, le deuxième tir de t se fera en même temps (dans le même pas maximal) que $t_{drain(t)}$, et $p_{obs(t)}$ contiendra encore un seul jeton indiquant que t vient d'être tirée. Pour simplifier dans la suite, on notera $fire(t)$ la propriété correspondant à la propriété de marquage $M(p_{obs(t)}) = 1$. Cet observateur peut également être appliqué au tir du reset, car comme démontré dans la Section 2.5, le reset peut être modélisé explicitement par une transition du réseau.

L'exclusion mutuelle se traduit dans le pipeline par le placement des deux opérateurs partagés dans deux cycles différents. Dans le modèle RTPN cela s'exprime par l'obligation de tirer le reset entre les tirs des transitions modélisant les opérateurs partagés. Sur l'exemple du RTPN $\mathcal{N}_2$ pour les opérateurs op_1 et op'_1, cela s'écrit : $(fire(op_1) \neq fire(op'_1)) \wedge ((fire(op_1) \Rightarrow \neg(fire(op'_1))\mathbf{U} fire(\mathsf{reset})) \vee (fire(op'_1) \Rightarrow \neg(fire(op_1))\mathbf{U} fire(\mathsf{reset})))$. De manière informelle, il n'y a jamais de tir de pas maximal contenant à la fois op_1 et op'_1 et s'il y a un tir de op_1 alors il n'y a pas de tir de op'_1 avant qu'il y ait eut un tir du reset ou bien s'il y a un tir de op'_1 alors il n'y a pas de tir de op_1 avant qu'il y ait eut un tir du reset. La propriété d'exclusion mutuelle sur l'exemple du modèle $\mathcal{N}_2$ s'écrit alors comme une

conjonction de telle contrainte :

$$\begin{aligned}\phi_{mutex} =& \big(fire(op_1) \neq fire(op'_1)\big) \wedge \\ & \big((fire(op_1) \Rightarrow \neg(fire(op'_1))\mathbf{U}fire(\mathsf{reset})) \vee (fire(op'_1) \Rightarrow \neg(fire(op_1))\mathbf{U}fire(\mathsf{reset}))\big) \wedge \\ & \big(fire(op_2) \neq fire(op'_2)\big) \wedge \\ & \big((fire(op_2) \Rightarrow \neg(fire(op'_2))\mathbf{U}fire(\mathsf{reset})) \vee (fire(op'_2) \Rightarrow \neg(fire(op_2))\mathbf{U}fire(\mathsf{reset}))\big) \wedge \\ & \big(fire(op_3) \neq fire(op'_3)\big) \wedge \\ & \big((fire(op_3) \Rightarrow \neg(fire(op'_3))\mathbf{U}fire(\mathsf{reset})) \vee (fire(op'_3) \Rightarrow \neg(fire(op_3))\mathbf{U}fire(\mathsf{reset}))\big)\end{aligned}$$

Cohérence La seconde propriété à vérifier est une propriété de cohérence du placement des registres sur les portions de circuits à partager. Il s'agit simplement de vérifier que les registres sont placés « aux mêmes endroits » dans les portions à partager. Autrement dit, il faut s'assurer que les signaux jumelés, qui vont être fusionnés lors du pliage, traversent le même nombre de registres.

Dans la Section 2.5, un motif permettant de modéliser explicitement le reset est donné. Ce dernier possède une place observatrice p_i^{obs} associée à chaque place p_i du réseau, qui compte le nombre de jetons ayant subi un reset dans la place. Autrement dit, cette place observatrice accumule les jetons présents dans p_i à chaque action reset. Pour simplifier dans la suite, on notera le nombre de jetons présents dans cette place observatrice : $reset(p_i) = M(p_i^{obs})$.

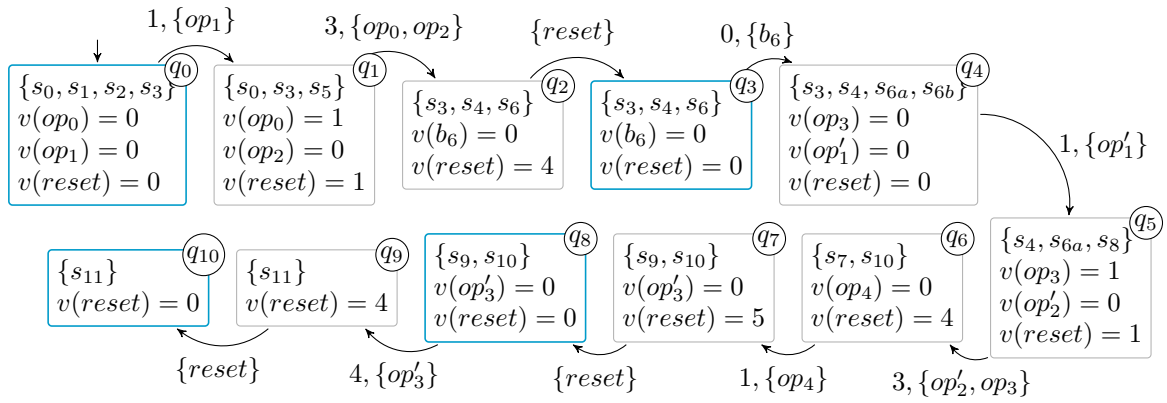
La propriété de cohérence du placement des registres entre les places jumelées s'écrit donc : $\phi_{consist} = (M(s_{11}) = 1) \wedge (reset(s_5) = reset(s_8)) \wedge (reset(s_6) + reset(s_{6a}) = reset(s_{10}))$. Elle vérifie en effet que dans l'état final $(M(s_{11}) = 1)$, les places jumelées auront reçu le même nombre de reset. Elle ne concerne que les places modélisant les signaux internes aux portions à partagées, car ce seront les signaux qui vont être fusionnés.

Propriété de pliage La propriété finale assure que ϕ_{mutex} soit maintenue jusqu'à ce qu'elle soit vérifiée en même temps que $\phi_{consist}$ (lorsque l'état final est atteint) : $\phi_{fold} = \mathbf{G}(\phi_{mutex}\mathbf{U}(\phi_{mutex} \wedge \phi_{consist}))$.

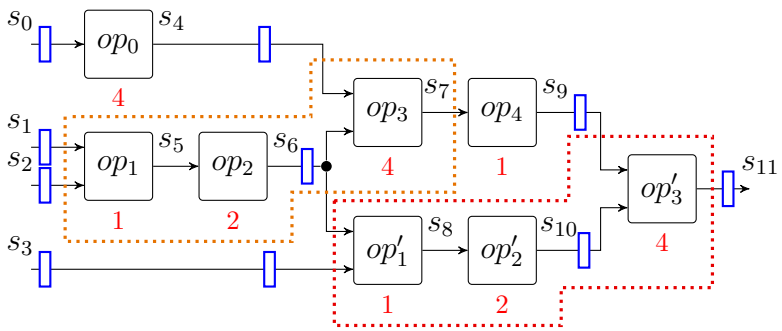
Pour résoudre le problème de pliage, notre approche consiste à explorer tous les états du réseau qui vérifie la propriété ϕ_{fold}. L'exploration est la même que celle proposée à la Section 4.3, mais lorsqu'un état invalidant la propriété est atteint la branche d'exploration est élaguée. Il ne s'agit donc pas de vérification à proprement parler, mais d'une synthèse de pipeline guidée par une propriété LTL.

Une exécution dont tous les états vérifient la propriété ϕ_{fold} est présentée sur la Fi-

gure 4.23a. Le pipeline synthétisé à partir de cette exécution correspond à celui de la Figure 4.23b. C'est le même qui a été présenté dans la Section 4.1, nous ne réexpliquons donc pas ici pourquoi ce pipeline permet effectivement le pliage.



(a) Exécution de $\mathcal{N}_2$ permettant le pliage



(b) Circuit C_2 avec pipeline pliant

FIGURE 4.23 – Multiplexage temporel basé sur un modèle RTPN

4.4.3 En pratique

Comme précisé précédemment, nous ne nous occupons pas du problème d'identification des portions de circuits pliables, nous les sélectionnons dans un premier temps manuellement. Une fois que les portions à partager ont été choisies, une limite minimale de la fréquence du pipeline f_p peut être déduite de la fréquence d'échantillonnage minimale souhaitée f_s. Soit n le nombre maximal de fois qu'une portion de circuit est partagée, alors la fréquence du pipeline doit satisfaire $f_p \geq n \cdot f_s$, ce qui nous donne une borne supérieure pour l'intervalle de reset. Si le motif de pliage contient plusieurs portions de circuit partagées différentes, il faut prendre le maximum du nombre d'occurrences de chaque

portion. Cette limite se généralise également au cas du partage de parties imbriquées, en multipliant le nombre maximal d'occurrences entre les niveaux imbriqués. Cependant, cette limite ne garantit pas que le circuit plié résultant aura la fréquence d'échantillonnage minimale f_s. Une sélection a posteriori des pipelines produits, basée sur leur latence (qui correspond à la période d'échantillonnage), est donc nécessaire. En d'autres termes, la limite inférieure de la fréquence du pipeline f_p n'est utilisée que pour réduire l'exploration à quelques solutions potentielles. Cependant, un traitement en amont permet d'éliminer certains cas où il n'y aurait pas de solution : il suffit de mesurer le chemin critique (latence) maximal parmi toutes les portions de circuits à partager l_{max} et le nombre de fois que cette portion est partagée n, la fréquence d'échantillonnage doit alors vérifier $\frac{1}{f_s} \geq n \cdot l_{max}$, sinon il n'y a pas de pliage possible.

Notre approche de synthèse de pipeline pour le pliage a été implémentée dans un outil prototype. Les premiers tests sont encourageants, et sur l'exemple du circuit C_2 présenté, l'outil produit le pipeline attendu en quelques millisecondes : 24 ms sur un processeur Intel Core i7-6700HQ.

En pratique pour vérifier la propriété ϕ_{fold}, les différentes places observatrices ne sont pas réellement implémentées dans le RTPN. Les états du réseau sont simplement enrichis avec le nombre de reset subis pour les places concernées par une propriété de cohérence $\phi_{consist}$ et un marqueur indiquant si une transition a été tirée pour les transitions concernées par une propriété d'exclusion mutuelle ϕ_{mutex}. Ceci pose un nouveau problème d'explosion de l'espace d'états, car il devient plus difficile de fusionner les états par égalité (cela arrive moins souvent). L'intuition est qu'une partie des informations décrivant l'exécution menant à un état est maintenant enregistrée dans l'état.

L'exploration des solutions est donc particulièrement longue, même pour un exemple simple comme celui du circuit C_2. On s'appuie donc sur une méthode donnant des solutions sous-optimales qui ne cessent de s'améliorer tant qu'on ne l'arrête pas. Dans cet exemple, on fait une première exploration des pipelines possibles en ignorant les contraintes de pliage, on obtient ainsi une approximation du nombre de bascules optimal pour le pipeline de ce circuit. Cette approximation est inférieure (ou égale) au nombre de bascules nécessaires pour le pipeline pliant. Puis, on explore les pipelines permettant le pliage et on s'arrête dès qu'on a obtenu un pipeline avec un nombre de bascules suffisamment proche de l'approximation précédemment calculée (moins de n% de bascules supplémentaires). Dans notre exemple, le meilleur pipeline possible possède 10 bascules, et c'est également le cas du pipeline pliant trouvé.

Chapitre 5

Compilation Simulink vers VHDL

Ce chapitre s'intéresse aux outils de synthèse de VHDL à partir d'un modèle Simulink.

La Section 5.1 fait un état de l'art des outils pouvant faire partie d'une chaîne de High Level Synthesis (HLS) allant d'un modèle Matlab/Simulink vers une description d'un circuit en VHDL. Dans la Section 5.2, nous présentons l'outil que nous avons conçu et qui intègre l'approche de pliage basée sur les RTPNs, présentée dans le Chapitre 4. Nous décomposons chacunes des parties de l'outil qui ont été développées. Enfin, la Section 5.3 propose une utilisation de bout en bout de notre outil, jusqu'à l'implémentation sur un FPGA. Nous comparerons notre approche avec un outil de Matlab appelé HDL Coder, en termes de ressources consommées par les circuits engendrés.

5.1 État de l'art

Dans cette section, nous nous intéressons aux outils de HLS existant au moment de l'écriture de ce manuscrit. En particulier, nous nous concentrons sur les outils de synthèse de VHDL à partir d'un modèle Matlab/Simulink.

Un modèle Matlab/Simulink est déjà relativement proche d'un circuit : il y a des signaux transportant des données et des blocs opérant sur les signaux. Il est alors possible d'implémenter directement un circuit réalisant la commande décrite par le modèle Simulink. Les signaux se voient attribuer des tailles de données fixées par défaut (des flottants simple ou double précision) et chaque bloc est traduit par un circuit générique. Cette solution est simple, mais elle engendre un circuit peu optimisé en termes de consommation d'unités logiques et de latence. De plus, elle ne garantit aucune spécification supplémentaire : débit minimal, ressources limitées, etc.

Quelques étapes d'optimisation sont souvent nécessaires pour accomplir la traduction.

La première étape est le choix des formats de données transportées par les signaux. En effet, Simulink est un outil de simulation de systèmes dynamiques et il effectue par défaut ses simulations sur des données en virgule flottante. Cependant, concevoir un circuit en

virgule flottante est très couteux en ressources (unité logique) et souvent non nécessaire pour obtenir les précisions attendues. Le choix des formats de données est donc primordial, car il aura une influence sur les ressources consommées et sur les temps de calcul de tous les opérateurs du circuit.

Ensuite, si un certain débit de fonctionnement du circuit est spécifié, la deuxième étape est l'ajout d'un pipeline. Elle nécessite néanmoins d'avoir obtenu au préalable une estimation des délais des opérations dans le circuit.

Pour finir, d'autres étapes d'optimisation permettent d'atteindre la limite de ressources spécifiée.

Il est possible d'optimiser l'implémentation des blocs dans le circuit, suivant les spécifications. Par exemple, un additionneur peut-être implémenté simplement par un ensemble d'additionneurs un bit avec propagation de retenue (*Ripple Carry Adder*), ou bien, pour réduire le temps de propagation, il est possible d'anticiper les retenues (*Carry Lookahead Adder*) mais cela au détriment d'une plus grande consommation de ressources. Un autre exemple est le gain (multiplication par une constante) qui peut être implémenté par un multiplieur classique avec une entrée constante, ou bien par un ensemble de décalages et additions (*Shift&Add*), ou bien par une table de vérité (*KCM* [Cha94]). Les différentes implémentations sont des choix de conception dépendant généralement des spécifications, par exemple un gain KCM est adapté à l'implémentation du circuit sur un FPGA car ce dernier est constitué de tables de vérité.

Une autre optimisation permettant la réduction des ressources consommées est le multiplexage temporel (pliage). Il est particulièrement adapté à la granularité d'un modèle Simulink : le partage d'un ensemble d'opérateurs devient le partage d'un ensemble de blocs (qui peut être représenté par un sous-système dans Simulink).

5.1.1 Choix des formats de données

Un format de données est défini par un intervalle de variation et une précision. Le problème du choix des formats de données consiste à déterminer les formats internes du circuit étant donné des formats d'entrées et de sorties et une erreur maximale autorisée. L'erreur maximale est généralement corrélée aux formats de sorties spécifiés.

De même que pour la conception de systèmes informatiques en général, il existe deux approches pour déterminer les formats des données internes au circuit : la première s'appuie sur une série de tests réalisés à différentes étapes de la conception, la deuxième consiste en une vérification formelle réalisée en amont de la conception. La première,

largement plus démocratisée dans l'industrie, a l'avantage d'être plus facile à mettre en œuvre, cependant elle ne permet de tester qu'un nombre fini de cas. La deuxième présente l'avantage d'apporter une certification mathématique et un domaine de validité possiblement infini.

Fixed-Point Designer [Mat] est un outil d'aide au choix des formats de données, basé sur une série de tests réalisés en simulation. *FiXiF* [Vol18], pour sa part, implémente une approche formelle de choix des formats de données pour les filtres linéaires.

Fixed-Point Designer Dans Matlab/Simulink un système de propagation des formats de données existe nativement. Il est basé sur de l'arithmétique d'intervalle. Cependant, dès lors que le modèle possède une boucle, la propagation des formats de données en avant peut boucler indéfiniment. Le choix des développeurs de Matlab/Simulink a été de stopper la propagation à partir d'un tour de boucle. Or, cela produit généralement des formats avec une grande précision qui pourrait être réduite pour économiser des ressources, sans pour autant perdre la précision attendue en sortie.

C'est dans cet objectif que l'outil Fixed-Point Designer est venu compléter Simulink. Ce dernier permet au concepteur de modifier manuellement les formats internes au modèle. Puis, il simule le modèle avec ces nouveaux formats avec des entrées correspondant aux cas d'usage. Enfin, des outils de mesure (visuels et numériques) permettent de déduire l'erreur par rapport à la simulation en virgule flottante. Généralement, des jeux de test sont conçus pour une grande quantité de cas d'usage et les formats peuvent être éprouvés sur chacun d'eux.

FiXiF [Vol18] FiXiF s'intéresse aux commandes linéaires, c'est-à-dire composées de gains (multiplications par une constante), d'additions et de retards (registres à la fréquence d'échantillonnage). Pour ce type de commande, il est possible de représenter l'algorithme matriciellement via le modèle Specialized Implicit Framework (SIF). Les auteurs de [VHL20] montrent que sur cette représentation matricielle, il est possible de calculer l'intervalle de variation et la précision des signaux internes en fonction d'une précision attendue en sortie. Pour ce faire, ils basent leur approche sur la mesure du Worst-Case Peak Gain (WCPG) qui fournit une borne sur la sortie d'un filtre (stable).

L'intérêt de cet outil est qu'il fournit une certification formelle sur la précision obtenue en sortie, car il considère le pire cas. De plus, il optimise les ressources consommées en choisissant les formats internes « minimaux » permettant d'obtenir la précision voulue.

Enfin, d'un point de vue pratique FiXiF est capable de construire un modèle SIF à partir d'un modèle Simulink. Il est de plus couplé avec l'outil FloPoCo (dont nous reparlons plus loin) ce qui lui permet de générer un circuit en VHDL. Toutefois, le circuit engendré est propre à la forme SIF, c'est donc un produit matriciel et non plus le flot de données du modèle de départ.

5.1.2 Synthèse de VHDL

Dans cette sous-section nous nous concentrons sur les outils capables d'engendrer du VHDL à partir d'un modèle Simulink.

HDL Coder HDL Coder est une boîte à outils intégrée dans Matlab/Simulink. Il fait suite à Simulink Coder qui permet de générer une implémentation en C/C++ d'un modèle Simulink, pour des cibles de type microcontrôleur. HDL Coder génère au choix du VHDL ou du Verilog.

L'outil est plutôt orienté vers une implémentation sur FPGA. En effet, il est possible de spécifier une cible FPGA et d'appeler un outil de synthèse de circuit propre à la cible (par exemple Xilinx Vivado) à la suite de la génération de code VHDL. Mais il peut tout à fait être utilisé uniquement pour la génération de VHDL (c'est ce que nous ferons dans la partie expérimentale).

Enfin l'outil permet également de spécifier la fréquence d'échantillonnage, ainsi qu'une fréquence de fonctionnement. Il engendre alors un pipeline permettant d'atteindre la fréquence de fonctionnement attendue.

Origami Origami [Sit+17a] est un outil de synthèse de VHDL à partir de modèle Simulink. Il peut également prendre en entrée un modèle Ptolemy [BL10] qui est une alternative open-source à Simulink.

Après avoir lu le modèle d'entrée, Origami peut effectuer des transformations sur le modèle : pipeline, pliage et optimisation d'opérateurs. Il possède notamment un algorithme de sélection de motif pliable, c'est-à-dire une solution au problème de détection de sous-graphes isomorphes. Puis, il peut soit générer directement du VHDL grâce à un moteur interne (et avec l'aide de FloPoCo pour des opérateurs spécifiques), soit générer un nouveau modèle Simulink qui pourra engendrer ensuite du VHDL via HDL Coder.

La force de cet outil est sa capacité à sélectionner automatiquement les motifs pliables. Les travaux de recherche qui ont guidé la conception de cet outil s'intéressent en particulier

à la sélection de portions de circuits pliables, plutôt qu'au pliage d'opérateurs un par un [Mö+15]. En se basant sur un algorithme de sélection de clique maximale, les auteurs de [Sit+17b] sont capables d'évaluer des combinaisons de motifs de pliage (possiblement imbriqués), dans l'objectif de minimiser la consommation de ressources. Cette solution est implémentée dans Origami.

Les concepteurs de Origami ont également travaillé sur le problème de modulo scheduling, comme expliqué dans la Section 4.4. Dans [Sit+18], ils proposent une formulation ILP (Integer Linear Programming) pour la synthèse d'un pipeline permettant le pliage d'opérateur un par un. Cette dernière combine des contraintes d'ordonnancement, des limites sur les ressources disponibles, la minimisation du nombre de registres consommés par le pipeline, et la mutualisation des registres lorsque cela est possible. Dans [Sit+19], ils proposent une stratégie en deux étapes : ils détectent d'abord les motifs pliables pouvant contenir plusieurs opérateurs, puis ils ordonnancent chaque motif de pliage sélectionné par une formulation ILP. Cette seconde approche ne propose toutefois aucune garantie sur le nombre de registres consommés.

FloPoCo FloPoCo [DP11] est un générateur de circuits réalisant des opérations arithmétiques. Il se concentre principalement sur les opérations en virgule flottante, mais embarque aussi des opérations sur des données en virgule fixe. Ses cibles principales sont les FPGAs, les opérateurs sont donc en grande partie optimisés pour ces architectures spécifiques. Mais il possède également des opérateurs non spécifiques au FPGA, qui peuvent être implémentés sous forme d'ASIC.

Le principal intérêt de FloPoCo est qu'il travaille sur des formats de données arbitraires. Il est capable d'optimiser les ressources consommées et/ou la latence d'opérations arithmétiques avec des valeurs de mantisses et exposants (en virgule flottante) ou MSB et LSB (en virgule fixe) arbitraires.

Il permet de plus de spécifier une cible FPGA dans l'objectif de réaliser des optimisations propres à la cible, notamment en fonction de la taille de ses LUTs.

Enfin, il est capable de construire un pipeline garantissant un certain débit de fonctionnement. Pour ce faire, il possède des fonctions d'estimations des délais d'opérateurs élémentaires pour chaque cible.

Notons que FloPoCo n'est pas capable de récupérer un flot de données d'un modèle Simulink. Mais il peut être considéré comme une étape pour aller d'un modèle Simulink vers un circuit en VHDL.

5.2 Notre outil de compilation

Dans le cadre du travail avec l'entreprise automobile Renault, nous cherchons à optimiser la synthèse de VHDL pour une cible FPGA, à partir d'un projet Matlab/Simulink. Plus précisément, l'objectif est d'implémenter un circuit synchrone avec un minimum de ressources, à la fois pour les unités logiques et les bascules, tout en garantissant que l'ensemble du calcul est effectué dans un laps de temps limité (la période d'échantillonnage). Un outil a été développé, afin de proposer une chaîne complète pour l'implémentation matérielle des lois de commande sur FPGA. Cette section présente les différentes parties de cet outil.

Le compilateur est classiquement organisé autour d'une représentation interne, indépendante des représentations d'entrée (Simulink) et de sortie (VHDL). Ce modèle interne sert de pivot à tous les outils d'optimisations. L'architecture générale de l'outil est représentée sur la Figure 5.1.

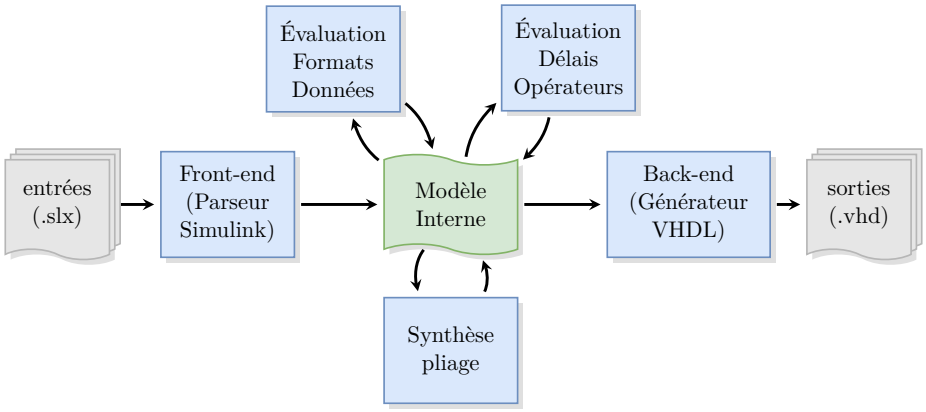


FIGURE 5.1 – Compilateur Simulink vers VHDL

Le front-end parse les fichiers d'entrée pour les transformer en ce modèle interne. Au cours de cette phase, quelques simplifications sont réalisées sur le modèle. À l'opposé, le back-end permet de générer la description du circuit (uniquement en VHDL à ce jour). Ces fichiers de sortie peuvent ensuite être synthétisés par outil EDA comme Xilinx Vivado.

Plusieurs étapes de raffinage sont alors exécutées sur le modèle interne. Sur le schéma dessiné dans la Figure 5.1, seules les 3 étapes principales sont indiquées.

La première passe concerne l'évaluation de la taille des signaux et de l'encodage associé (virgule fixe). Elle est actuellement majoritairement manuelle et est basée sur un étiquetage des signaux. Cette passe peut être remplacée par les outils internes de Matlab comme l'outil Fixed-Point Designer.

La passe suivante est l'évaluation des délais des opérateurs, elle est nécessaire pour la modélisation avec le CRTPN.

La dernière passe est la synthèse du pliage via une modélisation CRTPN présentée dans la Section 4.4.

5.2.1 Parseur Simulink

Le fonctionnement du parseur est décrit dans la Figure 5.2.

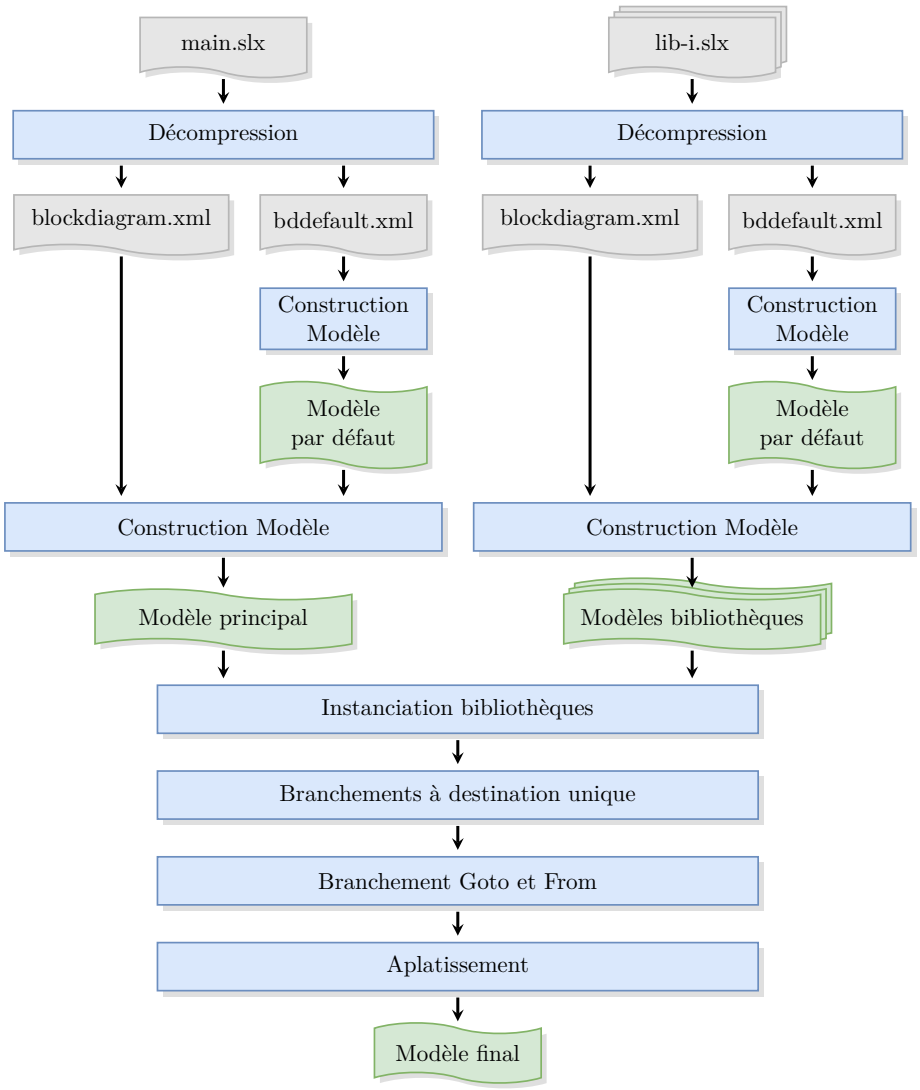


FIGURE 5.2 – Parseur Simulink

Un modèle Simulink est une structure imbriquée composée de blocs élémentaires et de sous-systèmes, reliés par des branchements portant les données. Certains sous-systèmes

sont parfois définis dans des fichiers séparés, appelés bibliothèques, dans l'objectif de permettre une réutilisation. Sur la Figure 5.2 les bibliothèques correspondent aux fichiers `lib-i.slx`.

Un fichier `.slx` est une archive, qui contient deux fichiers au format XML : `bddefault.xml` et `blockdiagram.xml`. Le premier fichier `bddefault.xml` contient les valeurs par défaut de tous les opérateurs utilisés dans le modèle. Par exemple, il contient la valeur d'un gain dans le cas où elle n'est pas spécifié. Le second fichier `blockdiagram.xml` contient le modèle avec tous les blocs, les sous-systèmes et les branchements. Il nous faut donc commencer par construire un modèle avec les valeurs par défaut. Puis utiliser ces valeurs, lorsqu'elles ne sont pas spécifiées, pour construire le modèle à partir du fichier `blockdiagram.xml`.

Ces opérations sont effectuées pour le fichier principal `main.slx` et également pour chacune des bibliothèques `lib-i.slx`. Le résultat est un modèle principal et un modèle par bibliothèque parsée. L'opération suivante est alors l'instantiation des blocs faisant référence à une bibliothèque dans le modèle principal.

Plusieurs opérations de simplification du modèle sont ensuite réalisées :

— Chaque branchement ayant plusieurs destinations est remplacé par un ensemble de branchement avec une destination unique.

— Les blocs `Goto` et `From` sont des simplifications visuelles pour brancher des blocs distants dans la représentation de Simulink. Ils sont donc supprimés et remplacés par un simple branchement.

— Un modèle Simulink est une structure imbriquée avec des sous-systèmes composés eux-mêmes d'un ensemble de blocs et de branchements. La structure est aplatie en ramenant le contenu des sous-systèmes au premier niveau d'imbrication.

Le modèle obtenu en sortie du parseur est celui que l'on nommera par la suite « modèle interne ».

5.2.2 Évaluation des formats de données

La Figure 5.3 présente une description de l'évaluation des formats de données.

Les formats de données des entrées/sorties et des constantes sont fournis au compilateur à l'aide d'un fichier appelé `dico.csv`. Il est parsé et les valeurs obtenues servent à initialiser les formats de données dans le modèle. Les formats manquants sont déduit par une propagation en avant, basée sur l'arithmétique des intervalles. Si le modèle possède des boucles, comme dans Simulink, la propagation s'arrête à partir d'un tour de boucle.

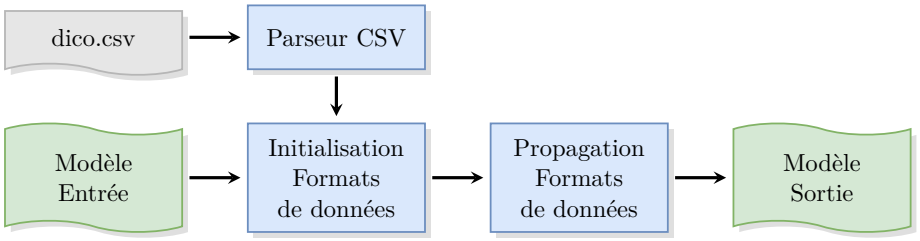


FIGURE 5.3 – Évaluation des formats de données

Les formats ainsi calculés sont sous-optimaux (ils peuvent consommer plus de ressources que nécessaires), mais ils garantissent la précision maximale atteignable.

Il est cependant possible de spécifier des formats de données internes dans le fichier `dico.csv`. Ceci peut permettre de limiter l'explosion de la taille des formats de données obtenue par une propagation en avant.

À terme, il serait intéressant de détecter les parties linéaires du modèle et de calculer les formats internes de ces dernières à l'aide de FiXiF. Il pourrait également être intéressant d'étudier analytiquement les sous-systèmes les plus fréquemment utilisés dans les bibliothèques, pour construire une méthode de choix des formats de données qui leur serait spécifique.

5.2.3 Évaluation des délais des opérateurs

La Figure 5.4 présente une description de l'évaluation des délais des opérateurs.

Le niveau d'abstraction des opérateurs dans Simulink ne permet pas de calculer précisément ces délais. L'approche utilisée est de synthétiser chaque opérateur pour déterminer son chemin critique. Le délai de chaque opérateur évalué est ensuite enregistré dans une base de données, car cette synthèse est gourmande en temps de calcul. FloPoCo implémente une approche similaire [ID17] sur des opérateurs élémentaires, puis raffine en construisant des modèles de temps de calcul pour les opérateurs plus complexes.

Le délai des opérateurs va nécessairement varier au cours de la synthèse du projet final (optimisations internes, routage, ...). Ce n'est qu'après la synthèse finale qu'il est possible de valider que tous les délais sont respectés. Les délais que nous stockons sont donc une approximation, mais il n'existe pas à notre connaissance de meilleure solution dans la littérature.

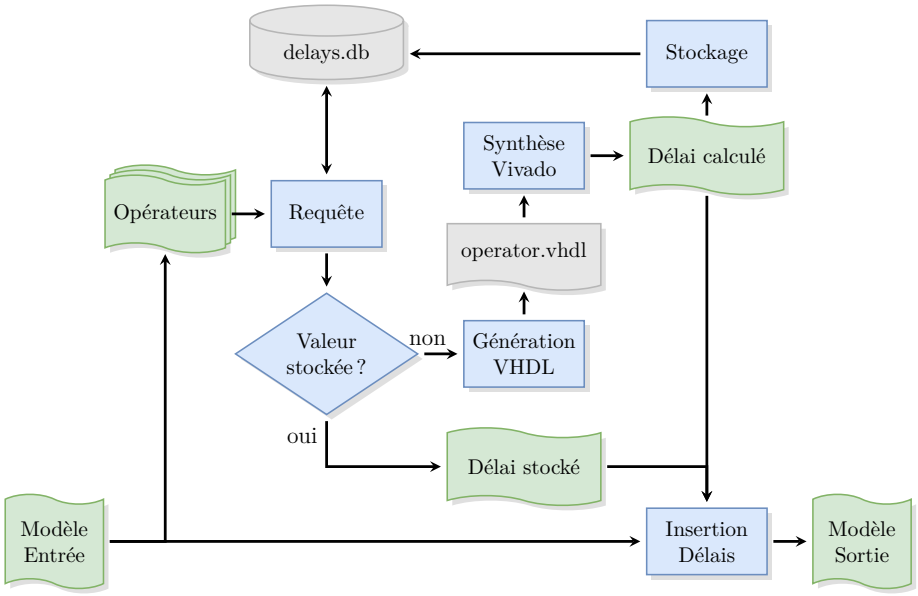


FIGURE 5.4 – Évaluation des délais des opérateurs

5.2.4 Synthèse de pliage

La Figure 5.5 présente une description de la synthèse de pliage.

Les parties des circuits partageables sont actuellement sélectionnées à la main, elles sont spécifiées dans un fichier au format XML appelé `folding.xml`. À terme, il serait intéressant d'automatiser cette sélection avec des méthodes d'identification de sous-graphes isomorphes. Dans notre étude de cas, nous bénéficions de l'abstraction de haut niveau de Simulink et certaines parties des modèles évalués ont des redondances assez évidentes à déterminer. D'autre part, l'utilisation de bibliothèques facilite l'identification des parties partageables.

Le modèle CRTPN est synthétisé à partir du modèle interne comme présenté dans la Section 4.3. Les contraintes de pliage sont obtenues à partir du motif de pliage spécifié comme expliqué dans la Section 4.4. L'exploration des états du modèle est effectuée par les algorithmes fournis dans la Section 3.5, adaptés pour prendre en compte les contraintes de pliage. D'une part les états explorés qui ne satisfont pas la contrainte d'exclusion mutuelle sont éliminés (élagage). D'autre part les états sont enrichis d'informations permettant de compter le nombre de jetons ayant subit un reset dans chaque place, ce qui permet de vérifier que la contrainte de cohérence est satisfaite dans l'état final. Puis, pour obtenir les meilleures exécutions du CRTPN (en termes de coût), nous utilisons un algorithme de type Dijkstra. Autrement dit, les états restants à explorer sont triés en fonction de leur

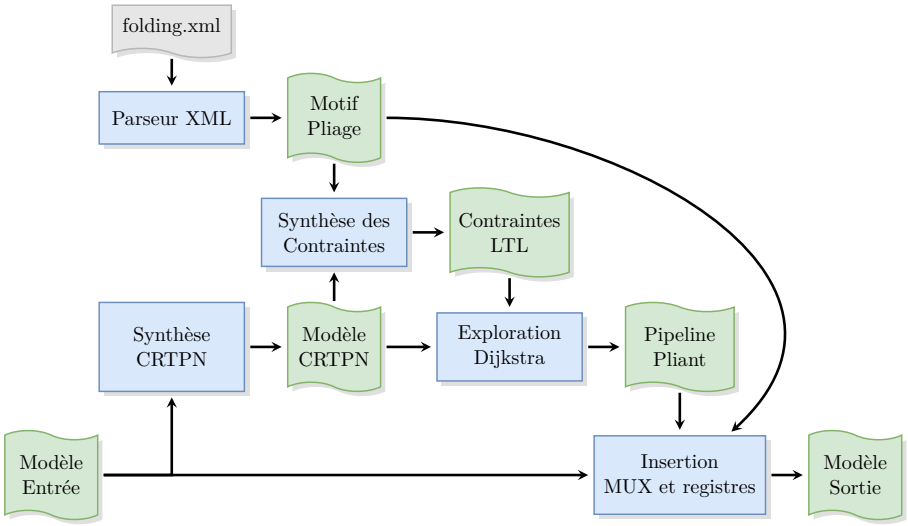


FIGURE 5.5 – Synthèse de pliage

coût et lorsqu'on atteint un état déjà visité, on ne conserve que celui qui a le coût minimal (l'autre branche d'exploration est élaguée).

Enfin, à partir du pipeline pliant obtenu, sont déduit les registres et les multiplexeurs permettant le pliage du modèle.

5.2.5 Génération du VHDL

La synthèse du VHDL à partir du modèle interne se base sur la composition. Chaque opérateur produit un composant VHDL dans un fichier séparé. Puis un composant principal implémente chacun des composants et des signaux internes, et effectue les branchements.

Un diviseur d'horloge permet d'obtenir une première horloge à la fréquence d'échantillonnage qui servira pour les registres fonctionnels (ceux présents dans le modèle Simulink). Un deuxième diviseur d'horloge génère l'horloge du pipeline qui servira à la fois pour les registres du pipeline et pour le signal de sélection des multiplexeurs.

Cette approche par composant simplifie grandement le débogage, car à chaque bloc Simulink correspond un composant VHDL (excepté ceux qui ont été fusionnés lors du pliage). Elle a cependant le défaut de générer une grande quantité de fichier.

5.3 Expérimentations

L'objectif de cette section est de montrer l'utilisation de notre outil sur un exemple concret, en faisant le traitement de bout en bout : depuis un modèle Simulink jusqu'à un circuit implémenté sur FPGA. Nous souhaitons nous concentrer sur une compilation complète et fonctionnelle, plutôt que sur une étude exhaustive à partir d'une grande quantité de modèles Simulink.

L'implémentation est effectuée sur une carte de développement qui possède un nombre limité de périphériques. Nous choisissons donc une application simple avec un seul signal d'entrée et un seul signal de sortie : un filtre numérique.

5.3.1 Application cible : un filtre numérique

Le filtre numérique que nous implémentons est un filtre passe-bas à Réponse Impulsionnelle Infinie (IIR). Son gabarit est dessiné dans la Figure 5.6 et ses spécifications sont les suivantes :

— fréquence d'échantillonnage : 10 kHz ;
— fréquence de coupure : $f_p = 1$ kHz ;
— gain minimum en bande passante : $\alpha_p = -1$ dB ;
— fréquence de frontière : $f_a = 3$ kHz ;
— gain maximum en bande coupée : $\alpha_a = -50$ dB.

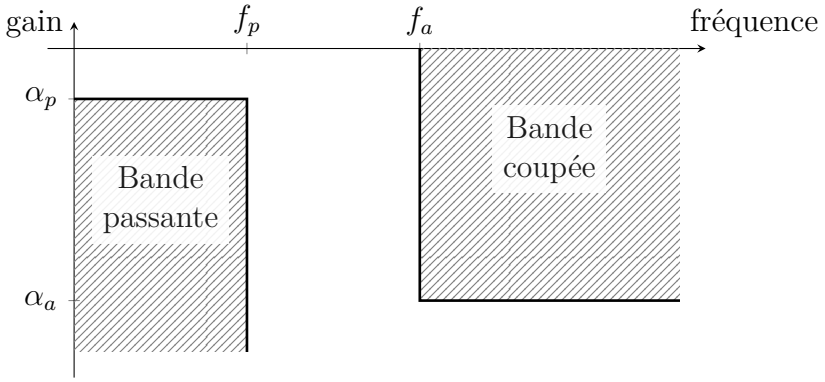


FIGURE 5.6 – Gabarit du filtre H

La fréquence d'échantillonnage est choisie en fonction des capacités des composants de conversions analogique/numérique et numérique/analogique. Les fréquences de coupure et de frontière sont ensuite choisies en fonction de cette fréquence d'échantillonnage.

Pour répondre à ses spécifications, nous concevons un filtre de Butterworth avec l'outil Filter Designer de Matlab. Nous obtenons un filtre d'ordre 5, dont la fonction de transfert est la suivante :

$$H(z) = 10^{-2} \frac{0.217 + 1.08z^{-1} + 2.17z^{-2} + 2.17z^{-3} + 1.08z^{-4} + 0.217z^{-5}}{1 - 2.708z^{-1} + 3.251z^{-2} - 2.063z^{-3} + 0.6839z^{-4} - 0.09378z^{-5}}$$

Le modèle Simulink correspondant à ce filtre est dessiné dans la Figure 5.7.

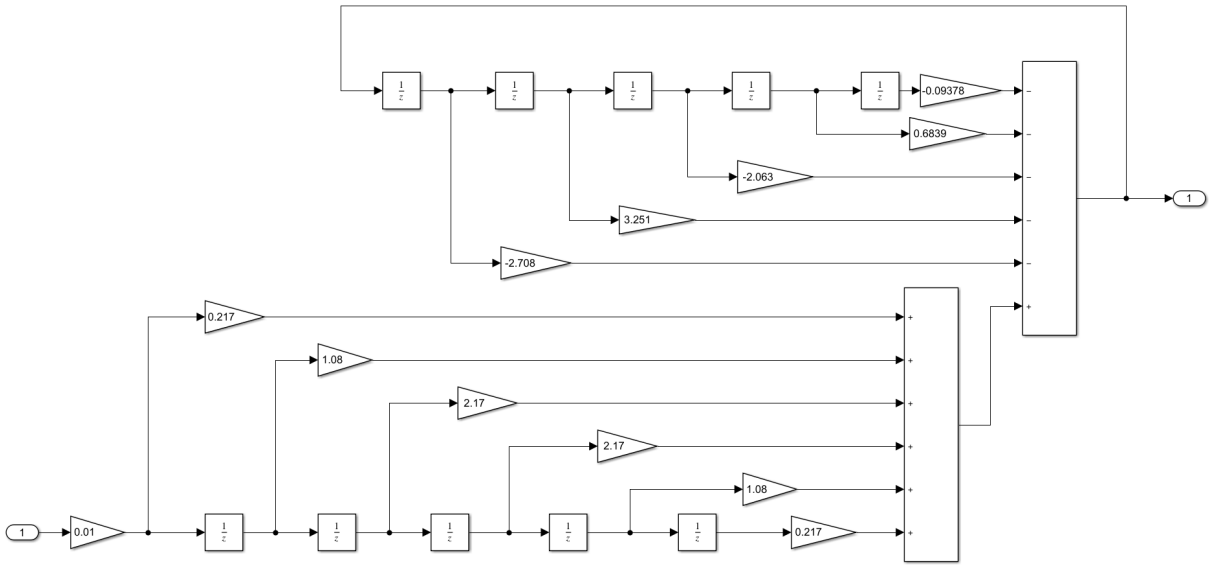


FIGURE 5.7 – Modèle Simulink du filtre H

On remarque qu'au numérateur de la fonction de transfert, il y a des paires de gains égaux. Ce sont ces derniers que nous allons partager.

5.3.2 Matériel

Nous implémentons ce filtre sur une carte de développement Basys 3 qui embarque un FPGA de la famille Artix-7 conçu par Xilinx. Le signal d'entrée est converti à l'aide d'un convertisseur analogique/numérique MCP3002 et le signal de sortie est converti à l'aide d'un convertisseur numérique/analogique MCP4902. Nous utilisons deux cartes d'extension embarquant ces convertisseurs qui se connectent à la carte de développement par les ports PMOD. On peut voir le système branché en fonctionnement dans la Figure 5.8. La carte d'extension branchée sur la droite contient le convertisseur analogique/numérique (signal d'entrée) et celle branchée sur la gauche contient le convertisseur

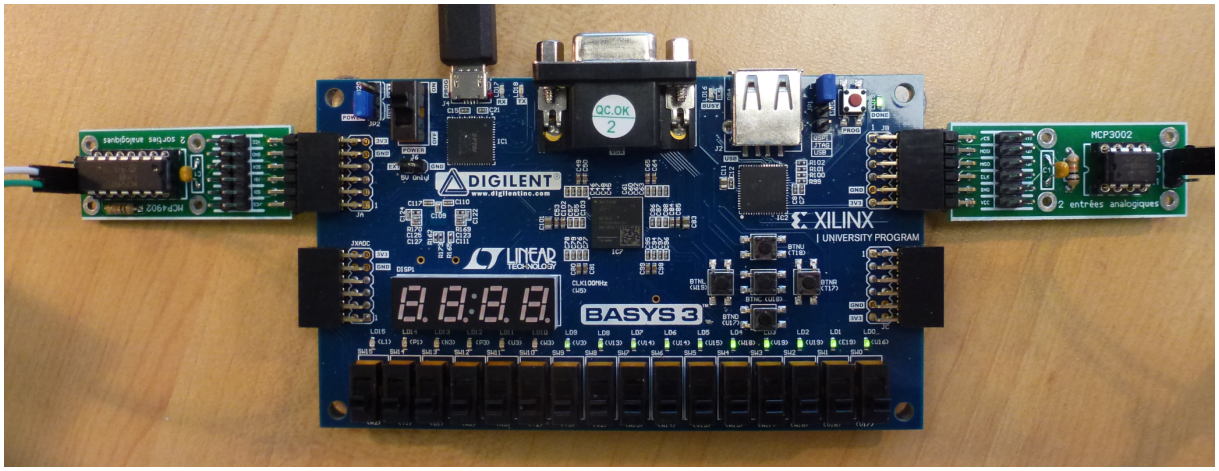


FIGURE 5.8 – Carte Basys 3 et ses deux cartes d'extension

numérique/analogique (signal de sortie).

Un générateur de basses fréquences nous permet de générer un signal d'entrée à la fréquence de notre choix. Le signal d'entrée et le signal en sortie sont affichés sur un oscilloscope. Nous faisons varier la fréquence du signal d'entrée pour vérifier le bon fonctionnement du filtre. Nous observons la diminution de l'amplitude du signal de sortie avec l'augmentation de la fréquence du signal d'entrée, jusqu'à la fréquence de coupure.

5.3.3 Configuration expérimentale

Pour notre expérience, nous souhaitons comparer les ressources consommées par un circuit généré avec notre outil (avec et sans pliage) et un circuit généré par HDL Coder.

Étant donné que notre outil n'implémente pas de solution d'optimisation des formats de données internes, nous utilisons les mêmes formats de données dans HDL Coder et dans notre outil. Les formats de données d'entrée et de sortie sont calculés à partir des capacités des convertisseurs. Les convertisseurs utilisent tous les deux des valeurs numériques sur 10 bits qui encode une tension allant de 0 à 3.3V. Nous fixons donc les formats d'entrée/sortie sur un format non-signé sur 10 bits, avec 2 bits pour la partie entière et 8 pour la partie fractionnaire.

Nous synthétisons les fichiers VHDL implémentant le filtre avec chacune des approches : HDL Coder, notre outil sans pliage et notre outil avec pliage. Puis les fichiers obtenus sont importés dans un projet Vivado.

Les deux composants MCP3002 et MCP4902 communiquent via un protocole SPI.

Leurs pilotes respectifs ont donc été développés en dehors des outils de synthèse de VHDL. De plus, un composant principal permet de brancher et de synchroniser les pilotes des convertisseurs et le filtre. Ces fichiers VHDL supplémentaires sont donc importés dans chaque projet Vivado.

Les communications avec les convertisseurs ont une durée qu'il faut prendre en compte. Le convertisseur MCP3002 a une trame de communication sur 16 cycles à une fréquence de $\frac{100}{64}$ MHz, et donc une durée de $10.24\mu s$. Le convertisseur MCP4902 a une trame de communication sur 32 cycles à une fréquence de $\frac{100}{32}$ MHz, et donc également une durée de $10.24\mu s$. Notre filtre est échantillonné à une fréquence de 10 kHz, il doit donc recevoir une nouvelle valeur toutes les $100\mu s$. Le cycle de $100\mu s$ est alors découpé en 8 portions de $12.5\mu s$ ce qui laisse le temps aux convertisseurs d'effectuer leur communication en une portion de cycle.

Le diagramme de Gantt dessiné dans la Figure 5.9 représente deux cycles de fonctionnement de notre système. La communication avec le convertisseur analogique/numérique

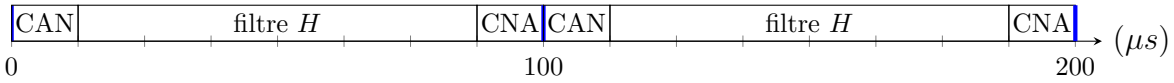


FIGURE 5.9 – Deux cycles de fonctionnement de l'implémentation du filtre numérique H

(CAN) est effectuée juste avant l'exécution du filtre, et celle avec le convertisseur numérique/analogique (CNA) est effectuée juste après. Ainsi le filtre reçoit bien une nouvelle mesure en entrée toutes les $100\mu s$.

Nous nous plaçons dans deux configurations pour permettre deux pliages différents :

1. Les 3 paires de gains du numérateur de H sont pliées, ce qui produit une instance pour chaque valeur de gain (donc 3 *gains* de 0.217, 1.08 et 2.17). Cette configuration de pliage est dessinée dans la Figure 5.10, avec les trois pliages en rouge, orange et bleu.

2. Chaque gain du numérateur est remplacé par un bloc *constante* et un bloc *multiplieur*, puis les 6 blocs *multiplieurs* sont pliés en une seule instance (donc 6 *constantes* et 1 *multiplieur*). Cette configuration de pliage est dessinée dans la Figure 5.11, avec un seul pliage en rouge pour les 6 multiplieurs.

Pour chaque configuration, nous synthétiserons trois circuits : le premier avec HDL Coder, le deuxième avec notre outil sans pliage et le troisième avec notre outil avec pliage.

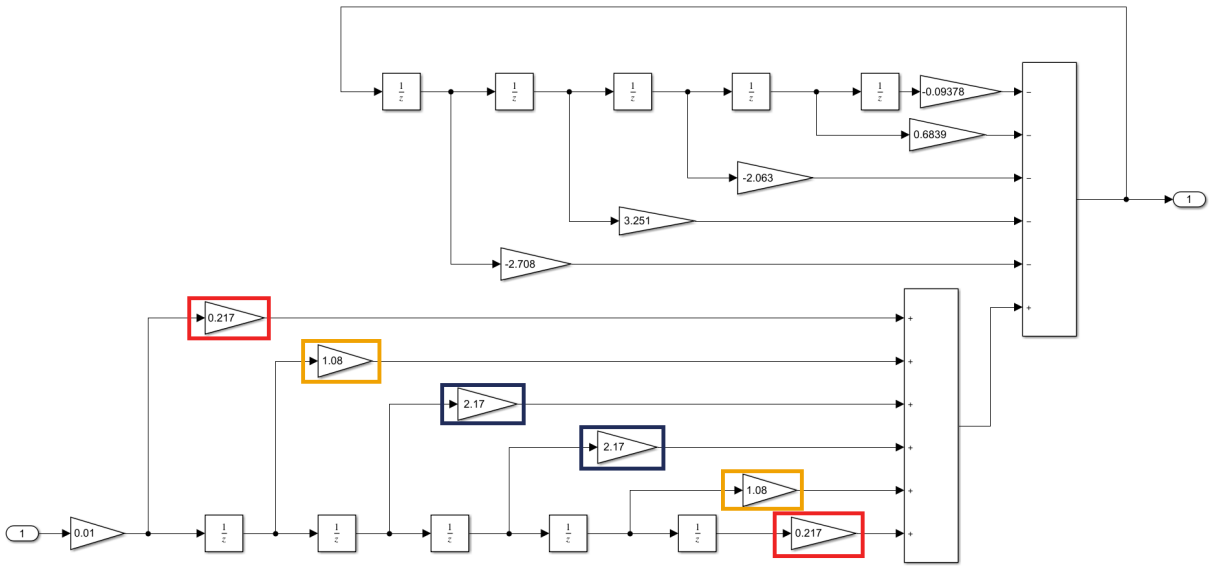


FIGURE 5.10 – Modèle Simulink du filtre H avec configuration de pliage 1

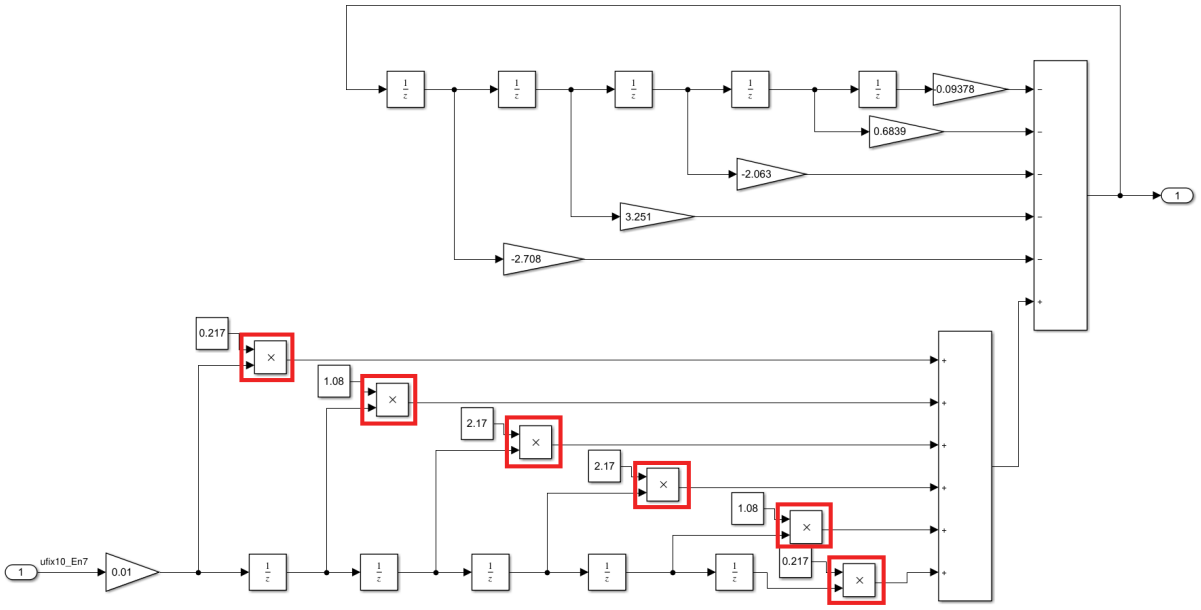


FIGURE 5.11 – Modèle Simulink du filtre H avec configuration de pliage 2

Chaque projet est ensuite synthétisé et placé sur la carte pour vérifier son comportement. La Figure 5.12 présente une des implémentations en fonctionnement. Le générateur basses fréquences, en haut de la photo, produit un signal à la fréquence 1.2kHz. L'oscilloscope affiche, de haut en bas, le signal d'entrée et le signal de sortie. Nous pouvons voir la fréquence d'échantillonnage par le crénelage du signal de sortie. Nous pouvons également

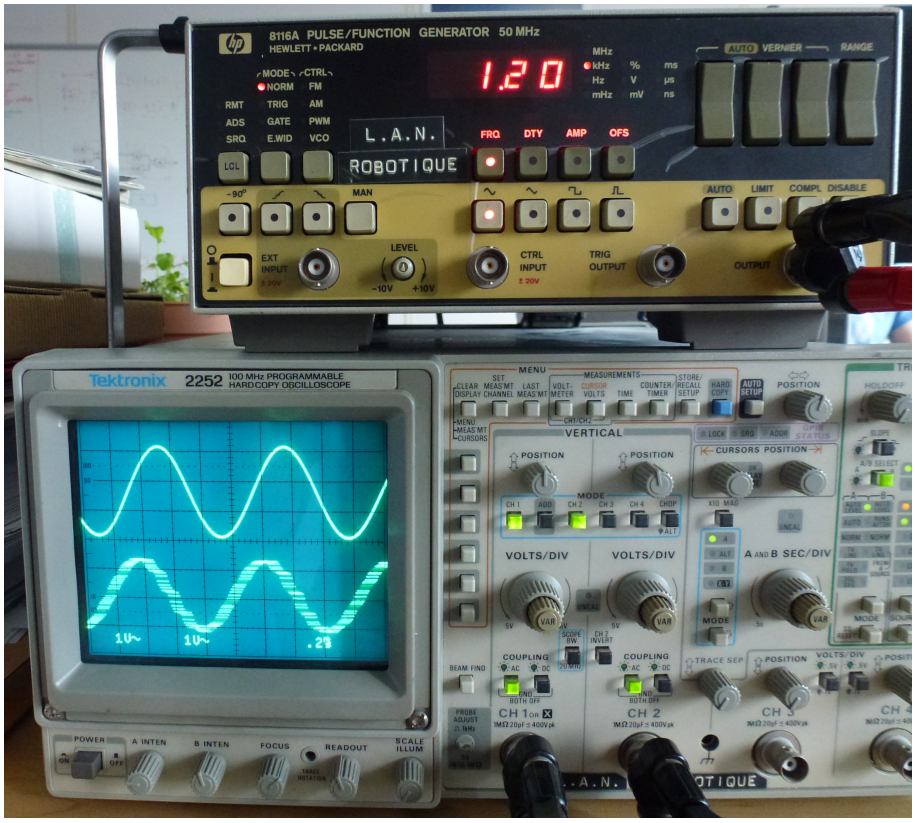


FIGURE 5.12 – Filtre numérique H en fonctionnement

observer l'atténuation de l'amplitude sur le signal de sortie.

5.3.4 Résultats

Pour comparer les ressources consommées par chaque approche, nous nous fions aux mesures fournies par Vivado. Le Tableau 5.1 présente ces mesures dans la première configuration, c'est-à-dire lorsque les gains du numérateur sont implémentés par des blocs *gains* sur Matlab/Simulink et sont pliés deux à deux. Le Tableau 5.2 présente les mesures

		LUT	FF	IO	BUFG	temps (s)
HDL Coder		1519	950	20	1	~ 20
Notre outil	sans pliage	802	256	20	1	0.03
	avec pliage	726	478	20	1	14.53

TABLE 5.1 – Ressources consommées par les circuits implémentant le filtre numérique H avec des blocs *gains* (configuration 1)

dans la deuxième configuration, c'est-à-dire lorsque les gains du numérateur sont implémentés par des couples composés d'un bloc *constante* et d'un bloc *multiplieur*. Dans cette deuxième configuration, le pliage est effectué sur les 6 multiplieurs dans l'objectif de n'en implémenter qu'un seul.

		LUT	FF	IO	BUFG	temps (s)
HDL Coder		1365	486	20	1	~ 20
Notre outil	sans pliage	801	256	20	1	0.04
	avec pliage	630	1202	20	1	286.74

TABLE 5.2 – Ressources consommées par les circuits implémentant le filtre numérique H avec des blocs { *constante* + *multiplieur* } (configuration 2)

Les tableaux comparatifs affichent la consommation des unités logiques (LUT), des bascules (FF) et des entrées/sorties (IO). La colonne BUFG correspond à des tampons intermédiaires permettant de limiter les signaux avec un fan-out trop élevé, c'est-à-dire les signaux utilisés par de nombreux opérateurs (typiquement un signal d'horloge).

Une partie du circuit est commune à tous les exemples, car les pilotes des convertisseurs ainsi que le composant principal sont utilisés dans chaque projet. Comme attendu, nous obtenons donc le même nombre d'entrées/sorties dans chaque circuit synthétisé.

Sous Xilinx Vivado nous désactivons les optimisations permettant de réduire la consommation de ressources qui sont spécifiques à certaines implémentations. D'une part, la cible FPGA que nous utilisons embarque des blocs DSP48 qui sont tout simplement des unités logiques et arithmétiques et qui permettent donc de réduire la consommation de LUTs. D'autre part, Xilinx Vivado permet d'utiliser les LUTs comme de la RAM distribuée ou des registres à décalage. Il appelle cela LUTRAM. Un LUTRAM est notamment utilisé lorsque beaucoup de registres sont placés en série, ce qui permet de réduire la consommation de FFs. Dans l'objectif de faire une comparaison équitable de la consommation de LUTs et de FFs, nous désactivons l'utilisation de blocs DSP48 et de LUTRAM.

La première chose que nous pouvons noter est que les temps de synthèse avec les trois outils sont raisonnables, au plus de l'ordre de quelques minutes. À noter que pour HDL Coder, le temps mesuré est approximatif et il semble qu'un système de cache peut permettre d'accélérer la synthèse, car une deuxième exécution prend quelques secondes ($\sim$ 7s).

D'autre part, on remarque que le circuit généré par HDL Coder consomme beaucoup plus de LUTs et de FFs que notre outil (dans les deux configurations). Ce résultat

s'explique notamment par l'implémentation de fonctionnalités supplémentaires non spécifiées : une entrée `enable` qui permet d'activer ou désactiver le filtre et une entrée `reset` qui réinitialise les registres du filtre.

Enfin, nous constatons que notre outil avec le pliage permet de réduire les unités logiques consommées, de l'ordre de 9.5% dans la première configuration et de 21.3% dans la deuxième, par rapport à notre outil sans pliage. En revanche, le pliage nécessite un plus grand nombre de bascules, une augmentation de 86.7% dans la première configuration et 369.5% dans la deuxième configuration.

Pour rappel, dans la première configuration, nous plions les 6 gains au numérateur deux à deux. Ce pliage ne semble donc pas permettre une grande réduction de ressources consommées, car seulement 3 gains sont économisés et 3 multiplexeurs sont ajoutés en contrepartie. D'un autre côté, dans la deuxième configuration, nous plions les 6 multiplieurs au numérateur en une seule instance. Cette configuration de pliage permet alors une plus grande réduction de ressources consommées, car 5 multiplieurs sont économisés. Cependant, le nombre de registres nécessaires explosent, car une seule instance du multiplieur est partagée 6 fois.

Il faut noter que les ressources consommées par chaque gain sont dépendantes du choix des formats des constantes et du type d'implémentation (multiplieur classique, Shift&Add, KCM, ...). Il est donc possible d'altérer grandement la réduction obtenue par ces pliages. Pour obtenir une comparaison équitable, les formats des constantes au numérateur sont tous les mêmes (dans les deux configurations) et les gains sont implémentés par des multiplieurs classiques.

Il est intéressant de noter que HDL Coder engendre un circuit différent si les gains sont représentés par un couple {constante + multiplieur}, comme représenté dans la Figure 5.11. Ceci est dû au mécanisme implémentant l'activation/désactivation du circuit (avec l'entrée `enable`). Ce dernier est différent pour un bloc *constante* et pour la constante d'un bloc *gain* (la fréquence de rafraîchissement n'est pas la même). Ceci explique pourquoi les ressources consommées par HDL Coder dans les deux configurations sont très différentes.

La Figure 5.13 montre le modèle interne généré par notre outil avant et après le pliage, dans le cas de la configuration de pliage 1. Le code couleur pour les blocs de ce graphe de flot de données est le suivant :

— en cyan les entrées/sorties ;
— en vert pomme les gains ;

— en violet les sommateurs (ou additionneurs) ;
— en vert foncé les blocs de redimensionnement des données (changement de format) ;
— en rouge brique les registres ;
— en beige les diviseurs d'horloge et les compteurs ;
— en orange les multiplexeurs.

Sur le modèle de gauche, avant pliage, nous pouvons observer les 6 gains avec les trois pliages encadrés en rouge, orange et bleu. Notons que les registres déjà présents avant le pliage sont les registres fonctionnels. Ils sont cadencés par l'horloge d'échantillonnage générée par le diviseur d'horloge en haut du modèle (en beige).

Sur le modèle de droite, après pliage, il n'y a plus qu'une seule instance de chaque gain, encadrés en rouge, orange et bleu. Ont été ajoutés trois multiplexeurs (en orange) ainsi que des registres de pipeline (en rouge brique) nommés `Pipeline_Reg`i. Les registres de pipeline sont cadencés par une deuxième horloge, l'horloge de pipeline. Enfin, les multiplexeurs reçoivent un signal de sélection généré par un compteur, appelé sélecteur (en beige). Ce dernier est également cadencé par l'horloge de pipeline.

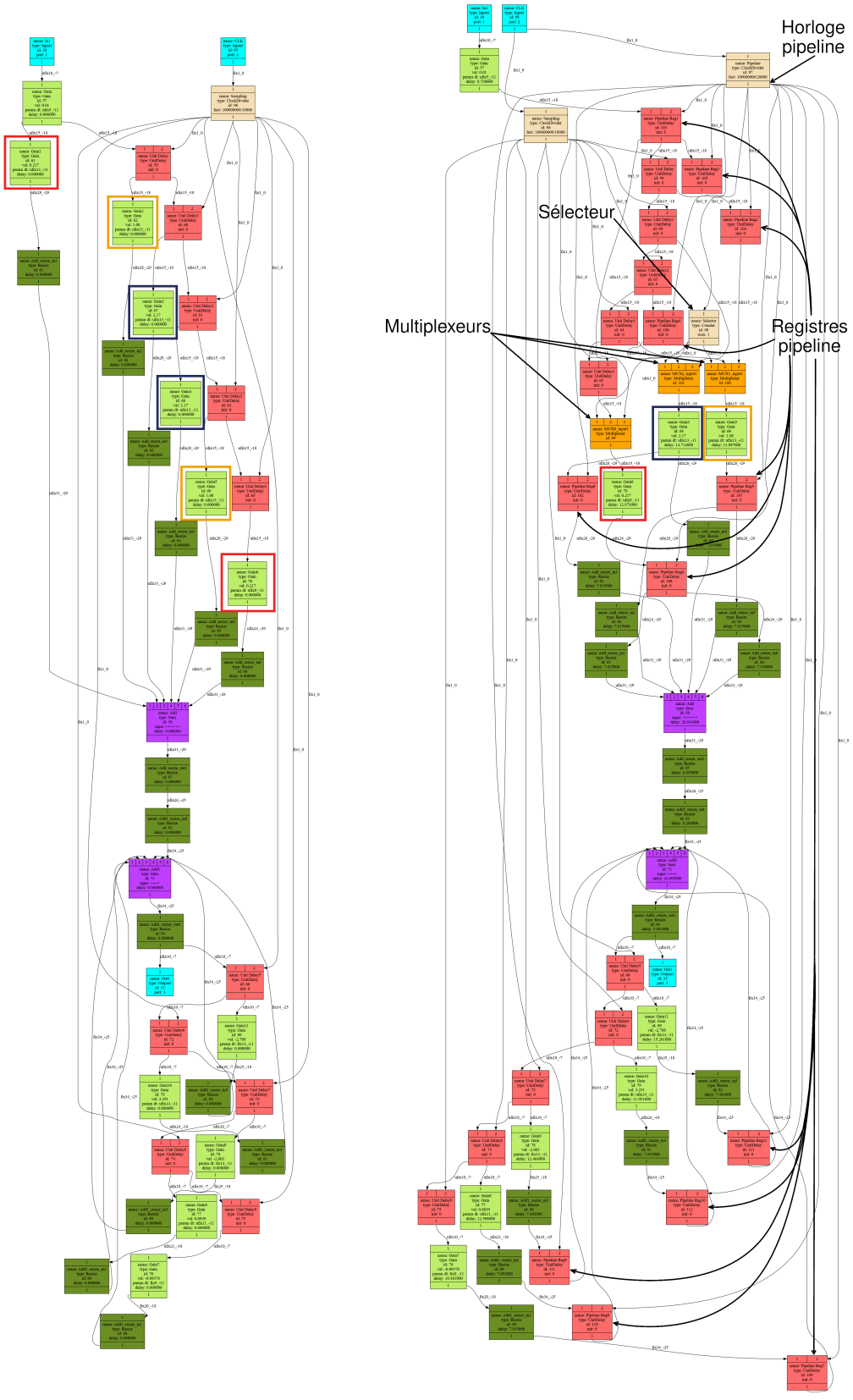


FIGURE 5.13 – Modèles internes du filtre H avant et après pliage (configuration 1)

Conclusion

Bilan

Les systèmes embarqués trouvent tous les jours de nouvelles applications. Chaque utilisation nécessite une conception particulière, leur étude est alors un enjeu majeur de l'industrie. Pour répondre à des fortes contraintes de vitesse de calcul, les industriels se tournent de plus en plus vers les FPGAs. L'atout de ces derniers est leur capacité à effectuer les traitements naturellement en parallèle. Cependant leur utilisation nécessite d'utiliser un niveau d'abstraction suffisamment bas proche des circuits logiques avec des langages de description dédiés comme VHDL ou Verilog. Des outils de High Level Synthesis (HLS) ont ainsi été introduits dans l'objectif de configurer les FPGAs à partir d'une spécification plus haut niveau.

Pour répondre aux attentes de notre partenaire industriel Renault, nous avons conçu un outil de HLS permettant de générer un circuit décrit en VHDL à partir d'une spécification sous Matlab/Simulink. Le cas d'étude proposé par Renault, nous imposait des contraintes strictes sur le circuit généré. Ce dernier ne doit pas dépasser une latence maximale, c'est-à-dire qu'il doit exécuter son calcul dans une certaine fenêtre de temps. De plus, des contraintes économiques imposent des cibles FPGA les plus petites possibles possible, il faut alors minimiser les ressources consommés par le circuit généré.

Pour répondre à ces contraintes, nous nous sommes intéressés au multiplexage temporel (ou pliage). Cette technique consiste à fusionner des portions de circuits présentes à plusieurs reprises et à séquencer leur accès. Pour effectuer ce multiplexage temporel, nous avons montré qu'il suffit de générer un pipeline particulier sur le circuit. Un pipeline est un placement de mémoires temporaires dans le circuit appelées registres. Dans le cas du multiplexage temporel, les registres permettent de sauvegarder l'état des données entre plusieurs accès à la ressource partagée.

Pour assurer le bon fonctionnement du circuit généré, nous avons basé notre travail sur une modélisation formelle. Cette modélisation permet de s'assurer de certaines propriétés sur le circuit généré, comme par exemple l'utilisation exclusive des portions partagées temporellement. Nous avons proposé une nouvelle classe de modèle pour les cir-

cuits synchrones pipelinés. Cette dernière s'appelle Réseau de Petri Temporisé avec reset et transitions retardables (RTPN). Elle étend les Réseaux de Petri Temporisés (TPNs) de Ramchandani avec des transitions *retardables* pouvant rater leur date de tir et une action reset qui réinitialise les horloges de toutes les transitions sensibilisées.

Nous avons étudié les propriétés de cette nouvelle classe de modèle. Elle a été démontrée équivalente à un automate temporisé à une horloge et hérite donc de tous les résultats de décidabilité et de complexité de cette classe. De plus, nous avons montré que l'extension des TPNs avec seulement les transitions *retardables* permettait de modéliser l'action reset. La surclasse correspondante s'appelle Réseau de Petri Temporisé avec transitions retardables (DTPN). Nous avons comparé l'expressivité de ce modèle par rapport au modèle de Réseau de Petri Temporel ($\mathbb{T}$PN) de Merlin. Dans le cas de réseau borné, nous avons montré que les DTPNs sont incomparables avec les $\mathbb{T}$PNs en temps dense et strictement plus expressifs en temps discret, au regard de la bisimulation temporelle faible. Nous avons proposé une abstraction symbolique des états du modèle pour laquelle le calcul du successeur peut-être effectué en temps linéaire par rapport au nombre de transitions du réseau. Nous avons proposé un ensemble d'algorithmes pour la construction de l'espace symbolique et avons montré une implémentation de ces derniers.

En nous basant sur une modélisation des circuits synchrones en RTPN, nous avons conçu une nouvelle méthode de synthèse de pipeline. Cette dernière permet de résoudre le problème de pipeline optimal, c'est-à-dire le placement des registres permettant de minimiser le nombre de bascules consommées. À partir de cette approche, il est possible de caractériser le pipeline souhaité. Ainsi, nous avons proposé une solution au problème de multiplexage temporel en spécifiant les propriétés du pipeline souhaité via la logique LTL.

Nous avons jeté les bases théoriques permettant de synthétiser un circuit qui assure une latence maximale tout en offrant une optimisation des ressources consommées à travers le pliage (le multiplexage temporel) de portions de circuits. Tous ces travaux sont basés sur des modèles formels qui assurent par construction le respect des propriétés du circuit. De plus, un outil a été développé qui permet de synthétiser un circuit décrit en VHDL à partir d'un modèle Matlab/Simulink. Ce premier prototype est prometteur et valide l'utilisation de bout en bout de notre approche basée sur les RTPNs.

Perspectives

Les travaux réalisés dans cette thèse donnent de premiers résultats encourageants et posent des bases pour des recherches futures.

Exploration des états des modèles Les classes de modèle RTPN et DTPN s'avèrent intéressantes pour notre cas d'étude, mais pourraient également être utiles à d'autres applications. Cependant, notre implémentation de l'exploration de l'espace d'états proposés est limitée et ne passe pas à l'échelle sur de gros exemples. Nous pensons donc qu'il reste encore du travail à effectuer. Pour commencer, l'exploration vise à trouver une exécution optimale en termes de coût, nous avons alors opté pour un algorithme Dijkstra. Mais il pourrait d'être intéressant de passer à un algorithme A^*, en déterminant des heuristiques qui sous-estiment le coût restant. Ensuite, nous pensons que l'algorithme de calcul des pas maximaux peut encore être amélioré. En effet, celui que nous avons proposé dans ce manuscrit se contente d'explorer les états d'un Réseau de Petri sans temps. Nous pensons qu'il est possible d'améliorer la complexité en se ramenant à un problème de clique maximale. Enfin, dans notre prototype, l'implémentation des structures de données est naïve et mériterait un plus grand travail de réflexion. Ceci pourrait drastiquement améliorer les performances de cet outil.

Cas particuliers de multiplexage temporel Nous avons montré dans ce manuscrit l'utilisation de notre approche basée sur les RTPNs pour le multiplexage temporel. Cependant, nous nous sommes placé dans un cas particulier pour lequel la fréquence du pipeline n'a pas d'importance, c'est la latence du circuit qui compte. Nous pensons donc que les recherches futures doivent porter sur le guidage de la recherche de solutions par la latence du circuit pipeliné. Dans ce contexte, il serait intéressant d'observer les effets de notre approche dans le cas de partage à la fois de portions de circuits avec un petit temps de propagation et de portions de circuits avec un grand temps de propagation. Ceci forcerait le découpage des grandes portions à partager, c'est-à-dire ceci entrainerait le placement de registres dans les grandes portions à partager. En effet, la fréquence du pipeline est fixée par la taille maximale des étages. Il faut donc trouver un compromis entre taille des étages et nombre d'étages, pour minimiser la latence. Nous pensons que nous pouvons également appliquer cette approche dans le cas où la fréquence du pipeline est capitale, c'est-à-dire lorsque nous souhaitons partager des portions de circuits tout en garantissant un débit de fonctionnement minimal. Ceci nous ramène au problème de

modulo scheduling [SWN07]. La difficulté dans notre approche serait d'être capable de modéliser l'insertion de « bulles », c'est-à-dire des temps où le pipeline reçoit en entrée une donnée vide.

Outil HLS L'outil de synthèse de VHDL à partir de spécification Matlab/Simulink que nous avons développé est une preuve de faisabilité de notre approche, mais il pourrait être largement amélioré. Pour commencer, nous n'avons pas eu le temps de nous attarder sur d'autres axes de recherche qui pourraient être impactants sur la taille du circuit synthétisé. Le choix des formats de données en est un. En effet, notre outil effectue une propagation naïve des formats de données en avant. Il pourrait être intéressant d'étudier les formats de données optimaux dans des sous-circuit particuliers. Par exemple, il serait intéressant d'utiliser l'approche de FiXiF [Vol18] pour tous les sous-circuits linéaires qui ont une boucle de rétroaction. Ensuite, dans notre approche du multiplexage temporel, nous n'avons pas pris en compte la sélection des portions de circuits à partager. Nous pensons que cette dernière pourrait être automatisée, soit par une méthode propre à notre cas d'étude en se basant sur la structure imbriquée de Simulink, soit par des méthodes de sélection de sous-graphes isomorphes.

Autres applications L'approche que nous avons développée est capable de synthétiser des pipelines qui répondent à des contraintes particulières. Nous pensons que cette méthode offre beaucoup de possibilités du fait de la flexibilité du modèle et de l'expressivité des logiques que l'on peut lui appliquer. Ainsi, nous pensons qu'elle peut résoudre d'autres problèmes de synthèse de pipeline en spécifiant simplement les propriétés attendues.

Acronymes

Méthodes Formelles

𝕋PN Réseau de Petri Temporel (Time Petri Net) 28, 45–49, 51, 62, 130

CRTPN RTPN à coût (Timed Petri Net with reset, delayable transitions and cost) 89–93, 95, 97, 99, 113, 116

CTL Computation Tree Logic 24, 100

DBM Difference Bound Matrix 63

DKS Structure de Kripke Temporisée (Durational Kripke Structure) 49

DTPN Réseau de Petri Temporisé avec transitions retardables (Timed Petri Net with Delayable transitions) 27, 41–56, 58, 60–64, 66–68, 70, 130, 131

ILP Integer Linear Programming 99, 111

LTL Linear Temporal Logic 18, 24, 99, 100, 130

PN Réseau de Petri (Petri Net) 27–32, 34, 36, 45, 46, 61, 62, 67, 87, 88, 131

RTPN Réseau de Petri Temporisé avec reset et transitions retardables (Timed Petri Net with reset and delayable transitions) . . 17–19, 27, 37–41, 43–45, 55–63, 71, 87–89, 91, 93, 95, 97–99, 101, 102, 105, 107, 130, 131

TA Automate Temporisé (Timed Automata) 55, 57–60, 62

TCTL Timed Computation Tree Logic 18, 24, 45, 62

TPN Réseau de Petri Temporisé (Timed Petri Net) 17, 27–29, 32–39, 44, 47, 61–63, 130

TS Système de Transition (Transition System) 21–23, 30, 31, 55

TTS Système de Transition Temporisé (Timed Transition System)22, 23, 35, 40, 44, 46, 47, 58

Conception de Circuits

ASIC Application-Specific Integrated Circuit 14, 25, 111

EDA Electronic Design Automation 25, 26, 112

FF Flip-Flop 26, 124

FPGA Field-Programmable Gate Array . . . 14–16, 19, 25, 26, 77, 80, 81, 107, 108, 110–112, 118, 119, 124, 129

HLS High Level Synthesis 15, 18, 88, 107, 129, 132

LSB Least-Significant Bit 25, 111

LUT Look-Up Table 26, 111, 124

MSB Most-Significant Bit 25, 111

RTL Register Transfer Level 80

SPI Serial Peripheral Interface 120

VHDL VHSIC Hardware Description Language (Very High Speed Integrated Circuit Hardware Description Language) 15, 18, 25, 26, 81, 107, 110–112, 117, 120, 121, 129, 130, 132

Traitement des Signaux

IIR Réponse Impulsionnelle Infinie (Infinite Impulse Response) 118

SIF Specialized Implicit Framework 109, 110

WCPG Worst-Case Peak Gain 109

Divers

XML Extensible Markup Language 114, 116

BIBLIOGRAPHIE

[Abd+08] Parosh Aziz Abdulla, Johann Deneux, Joël Ouaknine, Karin Quaas et James Worrell, « Universality Analysis for One-Clock Timed Automata », in : *Fundam. Informaticae* 89.*4* (2008), p. 419-450.

[ACD93] R. Alur, C. Courcoubetis et D. Dill, « Model-Checking in Dense Real-time », in : *Information and Computation* 104.*1* (1993), p. 2-34.

[AD94] Rajeev Alur et David L. Dill, « A Theory of Timed Automata », in : *Theoretical Computer Science* 126 (1994), p. 183-235.

[B+05] Beatrice Bérard, Franck Cassez, Serge Haddad, Didier Lime et Olivier H. Roux, « Comparison of the expressiveness of Timed Automata and Time Petri Nets », in : *3rd International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 05)*, t. 3829, Lecture Notes in Computer Science, Springer, sept. 2005, p. 211-225.

[BD91] Bernard Berthomieu et Michel Diaz, « Modeling and Verification of Time Dependent Systems Using Time Petri Nets », in : *IEEE transactions on software engineering* 17.*3* (1991), p. 259-273.

[Ber] Berkeley Logic Synthesis and Verification Group, *ABC : A System for Sequential Synthesis and Verification, Release 70930*, url : `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[BGR09] Hanifa Boucheneb, Guillaume Gardey et Olivier H. Roux, « TCTL model checking of Time Petri Nets », in : *Journal of Logic and Computation* 19.*6* (déc. 2009), p. 1509-1540.

[BL10] Christopher Brooks et Edward A. Lee, *Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java*, Poster presented at the 2010 Berkeley EECS Annual Research Symposium (BEARS)., 2010, url : `http://chess.eecs.berkeley.edu/pubs/655.html`.

[BR07] Hanifa Boucheneb et Hind Rakkay, « A more efficient time Petri net state space abstraction preserving linear properties », in : *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), 10-13 July 2007, Bratislava, Slovak Republic*, sous la dir. de Twan Basten, Gabriel Juhás et Sandeep K. Shukla, IEEE Computer Society, 2007, p. 61-70.

[Buf+07] D. Bufistov, J. Cortadella, M. Kishinevsky et S. Sapatnekar, « A general model for performance optimization of sequential systems », in : *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007.

[BV03] Bernard Berthomieu et François Vernadat, « State Class Constructions for Branching Analysis of Time Petri Nets », in : *Tools and Algorithms for the Construction and Analysis of Systems*, sous la dir. d'Hubert Garavel et John Hatcliff, Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 442-457, isbn : 978-3-540-36577-8.

[Cam+92] J. Campos, G. Chiola, J. M. Colom et M. Silva, « Properties and performance bounds for timed marked graphs », in : *IEEE Transactions on Circuits and Systems I : Fundamental Theory and Applications* 39.*5* (1992), p. 386-401, doi : 10.1109/81.139289.

[CEP95] A. Cheng, J. Esparza et J. Palsberg, « Complexity results for 1-safe nets », in : *Theoretical Computer Science* 147 (1995), p. 117-136.

[CES86] E. M. Clarke, E. A. Emerson et A. P. Sistla, « Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications », in : *ACM Trans. Program. Lang. Syst.* 8.*2* (1986), 244–263, issn : 0164-0925, doi : 10.1145/5397.5399, url : https://doi.org/10.1145/5397.5399.

[Cha94] K. D. Chapman, *Fast integer multipliers fit in FPGAs*, mai 1994.

[DP11] Florent de Dinechin et Bogdan Pasca, « Designing Custom Arithmetic Data Paths with FloPoCo », in : *IEEE Design & Test of Computers* 28.*4* (juill. 2011), p. 18-27.

[Gur22] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2022, url : https://www.gurobi.com.

[HMB07] A. P. Hurst, A. Mishchenko et R. K. Brayton, « Fast Minimum-Register Retiming via Binary Maximum-Flow », in : *Formal Methods in Computer Aided Design (FMCAD'07)*, 2007, p. 181-187, DOI : 10.1109/FAMCAD.2007.31.

[ID17] Matei Istoan et Florent de Dinechin, « Automating the pipeline of arithmetic datapaths », in : *Design, Automation & Test in Europe Conference & Exhibition (DATE 2017)*, Lausanne, Switzerland, 2017, p. 704-709.

[Jos+20] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne et Jordi Cortadella, « Buffer Placement and Sizing for High-Performance Dataflow Circuits », in : *Proc. of the 2020 ACM/SIGDA Int. Symposium on Field-Programmable Gate Arrays*, FPGA '20, Seaside, CA, USA : Association for Computing Machinery, 2020, 186–196, ISBN : 9781450370998, DOI : 10.1145/3373087.3375314, URL : https://doi.org/10.1145/3373087.3375314.

[KRP16] Yuliya Kalmykova, Leonardo Rosado et João Patrício, « Resource consumption drivers and pathways to reduction : economy, policy and lifestyle impact on material flows at the national and urban scale », in : *Journal of Cleaner Production* 132 (2016), Absolute Reductions in Material Throughput, Energy Use and Emissions, p. 70-80, ISSN : 0959-6526, DOI : https://doi.org/10.1016/j.jclepro.2015.02.027, URL : https://www.sciencedirect.com/science/article/pii/S0959652615001407.

[Lar14] Arturo Lara, « From complex mechanical system to complex electronic system : The case of automobiles », in : *Int. J. of Automotive Technology and Management* 14 (jan. 2014), p. 65 -81, DOI : 10.1504/IJATM.2014.058366.

[LMS02] F. Laroussinie, N. Markey et Ph. Schnoebelen, « On model checking durational Kripke structures », in : *5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'2002)*, t. 2303, lncs, Grenoble, France : Springer Verlag, avr. 2002, p. 264-279.

[LS91] Charles E. Leiserson et James B. Saxe, « Retiming Synchronous Circuitry », in : *Algorithmica* 6.*1-6* (juin 1991), p. 5-35, ISSN : 0178-4617.

[Mat] MathWorks, Inc., *Fixed-Point Designer*, URL : https://fr.mathworks.com/help/fixedpoint/.

[Mer74] P. M. Merlin, « A study of the recoverability of computing systems », thèse de doct., University of California, Irvine, CA : Dep. of Information et Computer Science, 1974.

[Mol82] M. Molloy, « Performance Analysis Using Stochastic Petri Nets », in : *IEEE Transactions on Computers* 31.*09* (1982), p. 913-917, ISSN : 1557-9956, DOI : 10.1109/TC.1982.1676110.

[Mö+15] Konrad Möller, Martin Kumm, Charles-Frederic Müller et P. Zipf, « Model-based Hardware Design for FPGAs using Folding Transformations based on Subcircuits », in : (août 2015).

[NB13] Mehrdad Najibi et Peter A. Beerel, « Slack Matching Mode-Based Asynchronous Circuits for Average-Case Performance », in : *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, San Jose, California : IEEE Press, 2013, 219–225, ISBN : 9781479910694.

[OW04] J. Ouaknine et J. Worrell, « On the language inclusion problem for timed automata : closing a decidability gap », in : *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* 2004, p. 54-63.

[Par+22] Rémi Parrot, Hanifa Boucheneb, Mikaël Briday et Olivier H. Roux, « Expressiveness and analysis of Delayable Timed Petri Net », in : *16th International Workshop on Discrete Event Systems (WODES'22)*, Prague, Czechia : IFAC, sept. 2022.

[PBR21a] Rémi Parrot, Mikaël Briday et Olivier H. Roux, « Pipeline Optimization using a Cost Extension of Timed Petri Nets », in : *The 28th IEEE International Symposium on Computer Arithmetic (ARITH 2021)*, IEEE, juin 2021.

[PBR21b] Rémi Parrot, Mikaël Briday et Olivier H Roux, « Réseaux de Petri temporisés pour la conception et vérification de circuits pipelinés », in : *Modélisation des Systèmes Réactifs (MSR'21)*, Paris, France, nov. 2021, URL : https://hal.archives-ouvertes.fr/hal-03587736.

[PBR21c] Rémi Parrot, Mikaël Briday et Olivier H. Roux, « Timed Petri Nets with Reset for Pipelined Synchronous Circuit Design », in : *The 42th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2021)*, t. 12734, Lecture Notes in Computer Science, Springer, juin 2021.

[PBR22] Rémi Parrot, Mikaël Briday et Olivier H. Roux, « Design and verification of pipelined circuits with Timed Petri Net », in : *Discrete Event Dynamic Systems - Theory and Applications (DEDS)* (2022).

[Pnu77] Amir Pnueli, « The Temporal Logic of Programs », in : *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, IEEE Computer Society, 1977, p. 46-57, DOI : 10.1109/SFCS.1977.32, URL : https://doi.org/10.1109/SFCS.1977.32.

[Pop13] Louchka Popova-Zeugmann, *Time and Petri Nets*, Springer, 2013, ISBN : 978-3-642-41114-4.

[Pop91] Louchka Popova, « On Time Petri Nets. », in : *Journal Inform. Process. Cybern., EIK (formerly : Elektron. Inform. verarb. Kybern.)* 27.*4* (1991), p. 227-244.

[Ram74] C. Ramchandani, « Analysis of asynchronous concurrent systems by timed Petri nets », thèse de doct., Cambridge, MA : Massachusetts Institute of Technology, 1974.

[SB06] Sangyun Kim et P. A. Beerel, « Pipeline optimization for asynchronous circuits : complexity analysis and an efficient optimal algorithm », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.*3* (2006), p. 389-402.

[Sif77] Joseph Sifakis, « Use of Petri Nets for Performance Evaluation », in : *Proceedings of the Third International Symposium on Measuring, Modelling and Evaluating Computer Systems*, North-Holland Publishing Co., 1977, 75–93.

[Sit+17a] Patrick Sittel, Martin Kumm, Konrad Möller, Martin Hardieck et P. Zipf, « High-Level Synthesis for Model-Based Design with Automatic Folding including Combined Common Subcircuits », in : *MBMV* 20 (fév. 2017), p. 102.

[Sit+17b] Patrick Sittel, Konrad Möller, Martin Kumm, P. Zipf, Bogdan Pasca et Mark Jervis, « Model-Based Hardware Design based on Compatible Sets of Isomorphic Subgraphs », in : déc. 2017, DOI : 10.1109/FPT.2017.8280140.

[Sit+18] P. Sittel, M. Kumm, J. Oppermann, K. Möller, P. Zipf et A. Koch, « ILP-Based Modulo Scheduling and Binding for Register Minimization », in : *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, p. 265-2656, DOI : 10.1109/FPL.2018.00053.

[Sit+18] Patrick Sittel, Thomas Schönwälder, Martin Kumm et Peter Zipf, « ScaLP : A Light-Weighted (MI)LP Library », in : *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, 2018, p. 1-10.

[Sit+19] P. Sittel, N. Fiege, M. Kumm et P. Zipf, « Isomorphic Subgraph-based Problem Reduction for Resource Minimal Modulo Scheduling », in : *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, p. 1-8, DOI : 10.1109/ReConFig48160.2019.8994768.

[SWN07] W. Sun, M. J. Wirthlin et S. Neuendorffer, « FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing », in : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.*2* (2007), p. 254-265.

[VHL20] Anastasia Volkova, Thibault Hilaire et Christoph Lauter, « Arithmetic approaches for rigorous design of reliable Fixed-Point LTI filters », in : *IEEE Transactions on Computers* 69.*4* (avr. 2020), p. 489 -504, DOI : 10.1109/TC.2019.2950658, URL : https://hal.archives-ouvertes.fr/hal-01918650.

[Vol18] Anastasia Volkova, « FiXiF toolbox : validated numerics for sound digital filter implementations », in : *18th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics.* Tokyo, Japan, sept. 2018, URL : https://hal.archives-ouvertes.fr/hal-02064338.

[Wik21] Wikipedia, *Synchronous circuit — Wikipedia, The Free Encyclopedia*, [Online ; accessed 20-June-2022], 2021, URL : https://en.wikipedia.org/wiki/Synchronous_circuit.

[Wik22] Wikipedia, *Asynchronous circuit — Wikipedia, The Free Encyclopedia*, [Online ; accessed 20-June-2022], 2022, URL : https://en.wikipedia.org/wiki/Asynchronous_circuit.

5.4 Publications

Les travaux présentés dans ce manuscrit ont fait l'objet des publications suivantes :

[**PBR21a**] Rémi Parrot, Mikaël Briday et Olivier H. Roux, « Pipeline Optimization using a Cost Extension of Timed Petri Nets », in : *The 28th IEEE International Symposium on Computer Arithmetic (ARITH 2021)*, IEEE, juin 2021

[**PBR21c**] Rémi Parrot, Mikaël Briday et Olivier H. Roux, « Timed Petri Nets with Reset for Pipelined Synchronous Circuit Design », in : *The 42th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2021)*, t. 12734, Lecture Notes in Computer Science, Springer, juin 2021

[**PBR21b**] Rémi Parrot, Mikaël Briday et Olivier H Roux, « Réseaux de Petri temporisés pour la conception et vérification de circuits pipelinés », in : *Modélisation des Systèmes Réactifs (MSR'21)*, Paris, France, nov. 2021, URL : `https://hal.archives-ouvertes.fr/hal-03587736`

[**Par+22**] Rémi Parrot, Hanifa Boucheneb, Mikaël Briday et Olivier H. Roux, « Expressiveness and analysis of Delayable Timed Petri Net », in : *16th International Workshop on Discrete Event Systems (WODES'22)*, Prague, Czechia : IFAC, sept. 2022

[**PBR22**] Rémi Parrot, Mikaël Briday et Olivier H. Roux, « Design and verification of pipelined circuits with Timed Petri Net », in : *Discrete Event Dynamic Systems - Theory and Applications (DEDS)* (2022)

CENTRALE NANTES

Titre : Réseaux de Petri temporisés pour la synthèse de circuits pipelinés

Mot clés : Réseau de Petri Temporisé, synthèse de pipeline, pliage de circuit

Résumé : Dans cette thèse, nous nous intéressons à l'optimisation des ressources consommées par un circuit implémentant une loi de commande pour la charge de véhicules électriques sur FPGA.

Tout d'abord, nous proposons une nouvelle solution au problème de la synthèse de pipeline minimisant les bascules et garantissant une fréquence minimale de fonctionnement. En se basant sur cette même approche, nous sommes capable de construire un pipeline permettant le pliage (ou multiplexage temporel) du circuit, c'est-à-dire qui permet la fusion de portions du circuit identiques en séquençant leur accès. Ainsi, les ressources consommées sont réduites à la fois en nombre de bascule et en nombre d'unités logiques.

Notre approche est basée sur un modèle de Réseau de Petri Temporisé avec des transitions *retardables*, pouvant rater leur date de tir, et une action spécifique appelée *reset* qui réinitialise les horloges de toutes les transitions. Ce modèle s'avère équivalent à un automate à une horloge. Une surclasse de ce modèle, les Réseaux de Petri Temporisés avec transitions retardables (sans reset), s'avère être incomparable, en terme d'expressivité en sémantique faible, avec les classes de Réseaux de Petri Temporels ou Temporisés en temps dense ou discret. Enfin, une exploration symbolique de ce modèle ainsi que des résultats de complexité théorique et pratique sont étudiés.

Title: Timed Petri nets for the synthesis of pipelined circuits

Keywords: Timed Petri Net, pipeline synthesis, circuit folding

Abstract: In this thesis, we are interested in the optimization of the resources consumed by a circuit implementing a control law for the charging of electric vehicles on FPGA.

First, we propose a new solution to the pipeline synthesis problem that minimizes the number of flip-flops and guarantees a minimum operating frequency. Based on this same approach, we are able to build a pipeline that allows the folding (or time multiplexing) of the circuit, i.e., that allows the merging of identical circuit portions by sequencing their access. Thus, the consumed resources are reduced both in number of flip-flops and in number of logical units.

Our approach is based on a Timed Petri Net model with *delayable* transitions that can miss their firing date, and a specific action called *reset* that resets the clocks of all transitions. This model is shown to be equivalent to a one-clock automaton. An overclass of this model, the Timed Petri Nets with delayable transitions (without reset), turns out to be incomparable, in terms of expressivity in weak semantics, with the classes of Temporal or Timed Petri nets in dense or discrete time. Finally, a symbolic exploration of this model and results on theoretical and practical complexity are studied.