THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Lucien RAKOTOMALALA

le mardi 15 février 2022

Titre :

Preuve formelle en calcul réseau

École doctorale et discipline ou spécialité :

ED MITT : Informatique et Télécommunications

Unité de recherche :

Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

M. Marc BOYER (directeur de thèse)
M. Pierre ROUX (co-directeur de thèse)

Jury :

M. Jean-Paul BODEVEIX Maître de Conférences Université Toulouse III - Président
M. Yves BERTOT Directeur de recherche INRIA Sophia Antipolis - Rapporteur
Mme Sylvie BOLDO Directrice de recherche INRIA Saclay - Examinatrice
M. Marc BOYER Ingénieur de recherche ONERA Toulouse - Directeur de thèse
M. Emmanuel GROLLEAU Professeur ISAE-ENSMA - Rapporteur
M. Jean-Yves LE BOUDEC Professeur EPFL - Examinateur
Mme Sophie QUINTON Chargée de recherche INRIA Grenoble - Examinatrice
M. Pierre ROUX Ingénieur de recherche ONERA Toulouse - Co-directeur de thèse

Remerciements

Mes premiers remerciements sont tout d'abord attribués à mes encadrants de thèse, Marc BOYER et Pierre ROUX, sans qui tout ce travail n'aurait pas été possible. Je tiens aussi à remercier les personnes qui ont accompagnés ces travaux : les rapporteurs de cette thèse : Emmanuel GROLLEAU et Yves BERTOT, l'ensemble des membres du projet Rt-Proof, les membres du jury de cette thèse. De plus, je souhaite remercier Euriell pour ces interventions tout au long de cette thèse.

Je tiens aussi à remercier l'ensemble de mes collègues de l'ONERA, doctorants comme stagiaires et membres de l'équipe COVNI. Je ne prendrai pas le risque de tous les citer tellement ils ont été nombreux à avoir créer cet ensemble chaleureux qui a aussi permis d'établir ces résultats.

Enfin, j'adresse mes plus grands remerciements à ma famille et mes amis. Merci à mes parents et un immense remerciement à ma Nono, leur soutien a été présent dès le début de cette thèse et jusqu'à la fin et je n'aurais jamais assez de place pour les en remercier. Mes amis ont quand à eux réussi à me sortir et me changer les idées quand cela été nécessaire. Il serait compliqué de citer toutes ces personnes une par une. Néanmoins, je tiens à remercier personnellement Louis, Quentin, Lucie, Mégane et Maeva, merci à eux d'avoir réussi à me soutenir pendant ces 3 années.

Résumé

Résumé français

De nos jours les avions ne peuvent se passer d'un important réseau embarqué pour faire communiquer les nombreux capteurs et actionneurs qui y sont disséminés. Ces réseaux ayant une fonction critique, en particulier pour les commandes de vol, il est important d'en garantir certaines propriétés telles que des délais de traversée ou l'absence de débordement de buffers. Le calcul réseau est une méthode mathématique permettant de réaliser de telles preuves [BBLC18]. Elle a joué un rôle clé dans la certification du réseau AFDX, dérivé de l'ethernet, utilisé à bord des avions les plus récents (A380, A350).

Le Calcul Réseau se base sur des résultats mathématiques utilisant l'algèbre tropicale. Ces résultats sont relativement simple mais déjà bien assez subtiles pour qu'il soit très facile de commettre des erreurs ou des omissions lors de preuves papier ou de calcul de valeur concrètes. Par ailleurs, les assistants de preuve sont un bon outil pour réaliser une vérification mécanique de ce genre de preuves et obtenir un très haut niveau de confiance dans leurs résultats.

Nous formalisons donc avec un tel outil les notions et propriétés fondamentales de la théorie du Calcul Réseau. Ces résultats font intervenir des propriétés sur les nombres réels, tel que des bornes supérieures et des limites de fonctions linéaires donc nous souhaitons utiliser un outil de formalisation capable d'implémenter un tel niveau mathématique. Nous utilisons l'assistant de preuve Coq. Il s'agit d'un outil disposant déjà d'un long développement dont la librairie *Mathematical Components* [MT21] qui permet de formaliser de l'analyse sur les nombres réels et la construction de structures algébriques comme celles utilisées dans le Calcul Réseau.

Le calcul de valeurs effectives repose sur des opérations de l'algèbre min-plus sur des fonctions réelles. Des algorithmes sur des sous ensembles spécifiques peuvent être trouvés dans la littérature [BT08]. De tels algorithmes et leurs implémentations sont toutefois compliqués. Plutôt que de développer une preuve de la bonne implémentation de ces algorithmes, nous prenons une implémentation existante comme Oracle et nous donnons des critères de vérifications en Coq.

Références

[BBLC18] Anne Bouillard, Marc Boyer et Euriell Le Corronc. *Deterministic Network Calculus : From Theory to Practical Implementation.* John Wiley & Sons, Ltd, oct. 2018.

[BT08] Anne Bouillard et Eric Thierry. « An Algorithmic Toolbox for Network Calculus ». In : *Discret. Event Dyn. Syst.* 18.1 (2008), p. 3-49.

[MT21] Assia Mahboubi et Enrico Tassi. *Mathematical Components.* Zenodo, jan. 2021.

Résumé anglais

Nowadays aircrafts need a large on-board network to enable the many sensors and actuators disseminated in them to communicate with one another. Since these networks have a critical function, particularly for flight controls, it is important to guarantee certain properties such as crossing delays or the absence of buffer overflows. Network calculus is a mathematical method for performing such proofs [BBLC18]. It played a key role in the certification of AFDX networks, derived from ethernet, used on board the most recent aircrafts (A380, A350).

Network Calculus is based on relatively simple mathematical results but already quite subtle enough that it is very easy to get errors or omissions during pen and paper proofs. Moreover, proof assistants are a good tool for mechanically checking such proofs and obtaining a very high level of confidence in their results.

We formalize with this tool the fundamental properties underlying the theory of Network Calculus. These results involve relatively basic properties on real numbers, such as upper bounds or even limits of piecewise linear functions. We use the Coq proof assistant as well as the Mathematical components library [MT21].

Effective computations rely on operators from the min-plus algebra on real functions [BT08]. Algorithms on specific subsets can be found in the literature. Such algorithms and related implementations are however complicated. Instead of redeveloping a provably correct implementation, we take an existing implementation as an oracle and propose a Coq based verifier.

Références

[BBLC18] Anne Bouillard, Marc Boyer et Euriell Le Corronc. *Deterministic Network Calculus : From Theory to Practical Implementation*. John Wiley & Sons, Ltd, oct. 2018.

[BT08] Anne Bouillard et Eric Thierry. « An Algorithmic Toolbox for Network Calculus ». In : *Discret. Event Dyn. Syst.* 18.1 (2008), p. 3-49.

[MT21] Assia Mahboubi et Enrico Tassi. *Mathematical Components*. Zenodo, jan. 2021.

Table des matières

Résumé **iii**

1 Introduction **1**
1.1 Problématique . . . 1
1.2 Analyse de traversée d'un réseau . . . 2
1.3 Méthodes formelles . . . 4
1.4 Bilan . . . 5

2 État de l'art **7**
2.1 Systèmes Temps réel . . . 7
2.2 L'analyse des systèmes temps réel . . . 8
2.3 Le *Calcul réseau* . . . 10
2.4 Méthodes formelles sur systèmes temps réel et réseaux . . . 11

3 Introduction au *Calcul réseau* **13**

4 L'assistant de preuve Coq **19**
4.1 Présentation de Coq . . . 19
4.2 Librairies . . . 27

5 Formalisation en Coq du *dioïde* des fonction *(min, +)* **29**
5.1 Théorie des *dioïdes* . . . 30
5.2 Le *dioïde* des fonctions *(min, +)* . . . 46
5.3 Conclusion . . . 55

6 Formalisation en Coq des définitions et théorèmes du *Calcul réseau* **57**
6.1 Formalisation des comportements réels . . . 58
6.2 Formalisation des courbes de contraintes . . . 65
6.3 Formalisation des bornes . . . 71
6.4 Cas d'étude . . . 76
6.5 Conclusion . . . 86

7 Vérification de calculs de fonctions *(min, +)* **89**
7.1 Introduction au calcul *(min, +)* . . . 89
7.2 Définitions des classes de fonctions . . . 94
7.3 Stabilités de $\mathcal{F}_{UPP}$ par les opérations *(min, +)* . . . 97
7.4 Stabilités de $\mathcal{F}_{UPP-PA}$ par les opérations *(min, +)* . . . 99
7.5 Vérification des opérations sur $\mathcal{F}_{UPP-PA}$ . . . 101
7.6 Implémentation . . . 103
7.7 Exemple sur le cas d'étude . . . 104
7.8 Conclusion . . . 107

8 Conclusion **109**

Annexe A Compilation du Code Coq **111**

Annexe B Licence NC-Coq **113**

9 Bibliographie **117**

Introduction

Les avions modernes sont conçus de nos jours avec de plus en plus d'électronique embarquée. Cette électronique, découpée en de nombreuses entités, assure le contrôle, la sécurité et le confort de l'avion. Les entités sont réparties sur toute la surface de l'avion, du bout du cockpit jusqu'à la queue de l'avion. Pour permettre à chaque entité de communiquer avec une autre, il est nécessaire d'installer un réseau informatique. Comme de plus en plus d'entités doivent être connectées, la taille de ce réseau augmente. Sur la figure 1.1, nous donnons un exemple d'architecture d'un réseau informatique embarqué dans un avion moderne. Chaque carré représente une entité du réseau et chaque trait représente un lien entre deux entités.

Un des enjeux dans la conception d'un tel réseau est d'assurer que les messages qui traversent ce réseau arrivent à temps, notons cette propriété P1, et qu'aucun message ne soit perdu faute de place, notons cette propriété P2. Pour modéliser et concevoir un tel réseau, les ingénieurs utilisent des méthodes d'analyses. Elles donnent des garanties sur les propriétés P1 et P2. Sur la figure 1.1, cela revient à avoir des garanties sur un temps de traversé d'un message de haut en bas de la topologie pour P1 et des garanties sur la capacité du centre de la topologie à gérer tout les messages traversant le réseau.

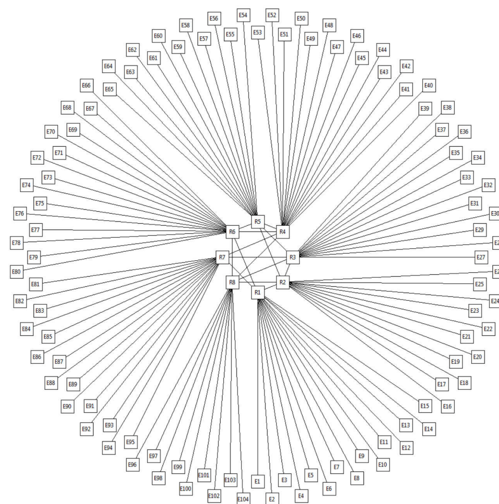


Figure 1.1 – Exemple de topologie

Pour avoir confiance sur ces garanties, des processus de certification sont déjà mis en place. Ils vérifient que les propriétés P1 et P2 sont respectées. L'objectif de cette thèse et de ce manuscrit est d'automatiser une partie de la certification. Cela revient à programmer la vérification de P1 et P2. Cependant, il nous faut un fort niveau de confiance dans ce programme : il ne doit pas contenir d'erreur. C'est pour cela que nous proposons d'utiliser un assistant de preuve pour augmenter ce niveau de confiance.

1.1 Problématique

Un réseau embarqué avionique est un système temps réel. Chaque entité a besoin de communiquer avec les autres en un temps borné : par exemple un capteur a besoin de communiquer une valeur qu'il relève à un calculateur dans l'avion.

Comme le nombre d'entités à connecter est de plus en plus grand, il est de moins en moins possible de connecter les entités une à une : une connexion directe pour chaque élément demanderai trop de lien. Donc, pour optimiser le nombre de connexion, un réseau est installé. Celui ci utilise des serveurs pour connecter les entités (cf. les carrés au centre de la Figure 1.1). Ces serveurs sont des ressources partagées. Il faut s'assurer que cet usage partagé n'empêche pas de satisfaire la contrainte temporelle.

Le réseau est utilisé dans un avion, c'est un système critique. Il est donc important de vérifier que cette contrainte temporelle est toujours satisfaite. L'objectif donc dans la conception de ce réseau est de valider ces contraintes.

Dans la section précédente, nous présentions deux propriétés qui doivent être validées par un réseau. Nous avions P1 pour assurer un temps de traversé du réseau et P2 pour assurer l'absence de perte d'informations. Pour la propriété P1, il faut être capable pour chaque message de garantir le temps qu'il va mettre pour traverser le réseau. Ce délai est souvent provoqué par les files d'attentes dans

les serveurs que le message va rencontrer. Pour garantir P2, il faut être capable de dire que ces files d'attentes sont suffisamment grandes pour contenir tout les messages. Si un message arrive dans une file d'attente pleine, celui ci est perdu. Une analogie avec le monde réel peut être donnée avec une voie de circulation routière et un embouteillage. Les voitures sont les messages, plus elles sont nombreuses et plus le trafic est ralenti. Par contre, si elles sont trop nombreuse et que le trafic est interrompu, alors le trop plein de voiture serait directement supprimé de la voie.

Il est donc important d'avoir des méthodes d'analyse pour que P1 et P2 soient vérifiées. Ces méthodes d'analyses sont mathématiques et sont implémentées par des outils. Ainsi, les ingénieurs vérifient P1 et P2 à l'aide de programmes implémentés à partir de ces théories mathématiques. Cependant, il est toujours possible que la méthode contienne des erreurs de raisonnement mathématique. Il est arrivé que ces erreurs soient trouvées bien après la publication de la méthode [Dav+07 ; Nel+15]. Ces erreurs mènent forcément à une erreur dans l'outil qui l'implémente. D'autres part, même en l'absence d'erreurs dans la théorie, des bugs peuvent apparaître lors de l'implémentation de l'outil.

Pour avoir confiance dans un outil, il faut donc à la fois vérifier la correction de la théorie et de son implémentation. Après avoir vérifié la méthode, il faut trouver un moyen de vérifier que les processus de calcul autour de la méthode d'analyse fonctionnent correctement et n'ajoutent pas d'erreur.

Pour augmenter la confiance dans la méthode d'analyse et dans l'implémentation de celle ci, un moyen efficace est d'utiliser des méthodes formelles avec un ordinateur. L'utilisation d'un ordinateur permet d'avoir une logique rigoureuse puisque cette logique sera imposé par l'ordinateur, elle est alors le coeur sur lequel repose la confiance dans la vérification. Ainsi, nous aurons le même niveau de confiance dans la méthode et dans son implémentation.

L'utilisation d'un ordinateur dans la vérification de la méthode requiert donc de reconstruire celle ci. Cette reconstruction passe par la formalisation et traduction dans le langage utilisé par l'ordinateur des définitions et propriétés de la méthode. Ensuite, pour vérifier l'implémentation, une technique consiste à utiliser une implémentation existante et de développer des critères pour vérifier que cette implémentation donne les bons résultats. Ces critères seront alors vérifiés de la même manière que la méthode initiale.

1.2 Analyse de traversée d'un réseau

Certains réseaux embarqué sont une catégorie de systèmes temps réel, nous les appelons réseaux temps réel. De tels réseaux partagent des ressources telles que des commutateurs, des médiums de communications ou des mémoires tampons (buffers) pour l'attente de transmission. Nous considérons que les informations qui circulent sur le réseau sont des messages, le contenu de chaque message est sans importance et seule la taille du message compte

L'analyse de réseau temps réel permet d'étudier la durée de transfert d'un message entre son entrée et sa sortie dans le réseau. Nous appelons cette durée le temps de traversée : la durée entre l'entrée d'un message dans le réseau et sa sortie de celui ci. Nous parlons alors aussi de temps de traversée de bout en bout. Pour cela, les méthodes d'analyse de réseau temps réel utilisent des techniques de modélisation pour formaliser le réseau puis donnent des propriétés mathématiques pour calculer cette durée. Ces méthodes d'analyse permettent aussi de déterminer si un message peut être perdu, par exemple en raison d'une mauvaise taille de mémoire tampon. L'objectif d'une analyse de réseau est double : déterminer un temps de traversée du message dans le réseau et assurer qu'aucun message ne sera perdu.

Pour obtenir le temps de traversé, il faut prendre en compte pour chaque message le temps passé dans chaque entité. Cette durée est un délai subi par le message. Le délai est donc la différence entre l'instant de réception d'un message par une entité du réseau et l'émission de ce message par cette même entité.

En connaissant le délai pour chaque message dans une entité et le rythme maximal d'arrivée de chaque message dans cette entité, il est possible de déterminer une borne sur le nombre de messages présents dans cette entité en même temps. Pour que l'entité puisse traiter chaque message correctement, elle doit disposer d'une mémoire suffisamment grande pour enregistrer tout les messages qui peuvent

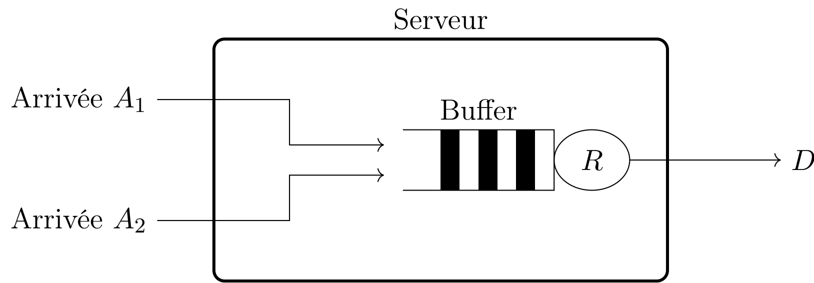


FIGURE 1.2 – Exemple d'un serveur avec deux entrées pour une sortie

être présent avant leur émission.

> **Remarque 1.1** (*Temps de traversée de bout en bout*) Dans la littérature et dans le reste de ce manuscrit, le terme *temps de traversée* représente une durée puisqu'il s'agit d'une différence entre deux instants.

Dans la Figure 1.2, nous prenons le schéma d'un serveur avec deux ports d'entrées pour des messages, nommés A_1 et A_2 et une sortie de messages nommée D. Le serveur sert les messages avec un débit R : c'est à dire qu'il ne peut traiter qu'une certaine quantité par rapport au temps. Dans cet exemple, le serveur fait office de multiplexeur des deux flux de messages d'entrées. L'élément au centre du serveur est un buffer : il sert de conteneur de message le temps que l'émetteur ait le temps d'envoyer les messages. Le temps passé dans ce conteneur pour un message est le délai imposé par le serveur.

La Figure 1.2 peut aussi nous servir à illustrer le délai qui peut retarder un message et même le pire cas de ce délai. Prenons par exemple l'arrivée d'un message sur A_1 et détaillons deux cas parmi ceux possibles. Dans un premier cas, l'arrivée A_2 reste muette. Alors le message de A_1 sera directement transmis sur la sortie D. Dans ce cas là, le délai que subit le message de A_1 ne dépend que du débit R du serveur. Celui ci est dans la majorité des cas suffisamment grand pour que le délai subi par A_1 soit très petit : le débit est bien souvent au moins 10 fois supérieur au débit d'arrivée du message. Dans un second cas, un message est en cours d'arrivée sur A_2. Alors, le message de A_1 sera retardé car, dans une politique où le premier arrivé est le premier servi, A_1 devra attendre que A_2 soit transmis sur D avant d'être à son tour transmis. Dans ce cas là, A_1 souffre d'un délai qui est pire que dans le premier cas car il somme la transmission du message de A_2 et sa propre transmission.

Dans l'ensemble des cas du dernier exemple, la mémoire doit pouvoir contenir tout les messages mis en attente. Dans notre exemple, prenons l'instant où le message de A_2 est déjà arrivé, et donc occupe une partie de la mémoire, et le message A_1 arrive. Alors, la mémoire doit pouvoir contenir le message de A_1 et la partie du message de A_2 qui n'a pas été émise. Donc, le débit de sortie R doit être assez grand pour vider rapidement l'espace mémoire de l'arrivée des messages de A_1 et A_2 et éviter le débordement de cette mémoire.

Une analogie de ce problème peut être donnée avec un seau d'eau percé au fond se remplissant par deux arrivées d'eau. La taille du perçage est le débit de sortie R de notre seau. Les deux arrivées sont les arrivées A_1 et A_2. Si les débits de A_1 et A_2 sont trop importants par rapport à la taille R, le seau va commencer à se remplir. Ce seau représente la mémoire de notre serveur, et si celle ci est mal dimensionnée, un débordement survient.

Pour résumer, la méthode d'analyse doit permettre de déterminer le délai que va subir un message entrant dans un élément réseau. Ce délai dépend de la vitesse à laquelle le message sera servi : le débit mais aussi la priorité accordée aux messages vis à vis des autres messages et donc de la politique d'ordonnancement. Connaître ce délai permet de dimensionner la taille mémoire nécessaire pour le stocker avant son émission : celle ci dépend donc du débit de sortie et des débits d'entrée.

Pour évaluer ces propriétés, les méthodes d'analyse utilisent deux approches différentes. Nous considérons alors les deux familles de méthodes d'analyse suivantes :

— les méthodes d'analyse de temps de réponse considèrent une ressource (par exemple un processeur), un schéma (*pattern*) de sollicitations (par exemple, des tâches périodiques, avec ou sans

décalage initial – *offset*) avec une échéance par requête, et cherchent à vérifier que chaque requête est traitée avant son échéance, généralement en calculant une valeur (ou une approximation) du temps de réponse de la requête.

— les méthodes avec courbes d'arrivée capturent le modèle de sollicitation par une courbe générique, et cherchent plutôt à borner le temps de réponse (localement ou sur une série de ressources) et à propager le modèle de sollicitation.

Lorsque l'on cherche à étudier un réseau, les méthodes d'analyse de temps de réponse sont peu adaptées car

1. les perturbations sur une ressource font que le schéma de sollicitation en sortie peut être différent de celui en entrée (ce qui oblige à changer de méthode, ou à choisir un méthode très générique, moins précise),
2. ces méthodes ne permettent que de calculer des temps de réponse par ressource, alors que l'on sait que le pire temps de réponse globale est plus petit que la somme des délais par ressource,
3. ces méthodes ont surtout été utilisées pour analyser des tâches exécutées sur des processeurs, suivant des politiques un peu différentes de ce que l'on trouve dans les entités réseau.

Dans cette étude, nous portons notre intérêt sur une méthode à base de courbe d'arrivée appelé le *Calcul réseau*. Il existe d'autres méthodes sur la même base telles que la méthode dite de *trajectory approach* ou base telles que l'analyse de performances compositionnelle notée CPA.

1.2.1 Le *Calcul réseau*

Le *Calcul réseau* (*Network Calculus* en anglais) fait partie des méthodes d'analyse de réseaux qui permettent d'obtenir des bornes sur les temps de traversée des messages [BBLC18 ; LBT01]. Des fonctions cumulatives de données sont utilisées pour modéliser le passage de messages à différents points du réseau. Le *Calcul réseau* permet de spécifier plusieurs bornes sur les quantités : sur les quantités maximales de données en certain points et sur les performances des serveurs. Avec ces spécifications, le *Calcul réseau* donne ensuite des théorèmes pour déterminer des bornes sur les délais subis par les messages et sur les temps de traversée du réseau.

Le *Calcul réseau* utilise un modèle mathématique particulier : le *dioïde* des fonctions *(min, +)*. Un *dioïde* est une structure algébrique tropicale [MG08], nous donnons des définitions et propriétés de celle ci au Chapitre 5. Les propriétés données par le *Calcul réseau* sont donc des expressions algébriques dépendantes de cette structure algébrique. Nous disons que ce sont des expressions algébriques *(min, +)*. Il faut ensuite utiliser un logiciel ou calculette pour obtenir des valeurs concrètes. Il faut donc utiliser un programme capable de calculer des expressions algébriques *(min, +)*.

Le *Calcul réseau* fait partie des méthodes utilisées pour borner les temps de traversée sur les réseaux AFDX, un réseau avionique utilisé dans la plupart des avions modernes. Le *Calcul réseau* a l'inconvénient d'être une méthode dite pessimiste par rapport à d'autres méthodes : les résultats obtenus avec cette méthode sont des bornes et donc ne représentent pas un cas réel. Des résultats obtenus avec une méthode de simulation, qui sont par définition plus optimistes, ne permettent pas d'avoir les mêmes garanties : il est impossible d'affirmer que tous les cas possibles ont été simulés. De plus, le *Calcul réseau* présente l'avantage de pouvoir s'effectuer sur des réseaux de grandes tailles (plusieurs centaines de noeuds) alors que des approches par vérification de modèle (*model checking*) ne peuvent pas [Cha+06].

1.3 Méthodes formelles

Certaines techniques informatique permettent de raisonner de manière mathématique. Ces outils sont généralement utilisés pour vérifier des propriétés sur un logiciel ou sur du matériel électronique. Ces techniques font partie des méthodes formelles.

Une grande partie de ces techniques sont utilisées à l'aide d'outils eux même informatiques. Certains d'entre eux permettent de vérifier par exploration automatique de tout les cas possibles qu'un modèle répond à l'ensemble des propriétés exposées par un utilisateur. Cette exploration, qui correspond à

une preuve mathématiques que la propriété est remplie, peut être entièrement automatique ou peut demander une rédaction de la part d'un utilisateur.

> **Remarque 1.2** (*Preuve Formelle*) Dans le reste du manuscrit, la référence à *preuve formelle* signifiera une preuve construite à l'aide ou vérifiée avec un outil de méthodes formelles.

Au cours de ce manuscrit, nous allons nous intéresser à un assistant de preuve. Un tel outil informatique fait partie des techniques de méthodes formelles. Nous allons dans le reste de cette section présenter cet ensemble d'outils puis nous allons présenter l'assistant de preuve que nous utiliserons dans ce manuscrit.

1.3.1 Les assistants de preuve

Un assistant de preuve est un logiciel informatique qui permet de vérifier des preuves. Son utilisateur écrit des définitions et des preuves dans le langage et la syntaxe du logiciel. Le logiciel valide ensuite la forme et la validité des définitions et des preuves.

Le langage du logiciel permet à un utilisateur de définir des objets mathématiques, d'énoncer des propriétés sur ces objets et de rédiger les preuves correspondantes aux propriétés.

Un assistant de preuve est basé sur un coeur logique de déduction que nous appelons noyau. Celui-ci vérifie la validité des différentes constructions apportées par l'utilisateur. Pour cela, le noyau utilise un langage complexe très fortement typé qui est très peu utilisable tel quel.

Pour écrire des définitions et des preuves, l'assistant de preuve met alors à disposition de l'utilisateur un ensemble de tactiques. Celles ci permettent de générer une traduction comprise par le noyau. Ainsi, l'assistant de preuve permet à l'utilisateur d'interagir avec son noyau : la traduction étant plus ou moins automatique, l'utilisateur peut essayer plusieurs tactiques pour une même preuve et adapter celle ci en fonction des réponses du noyau. De plus, l'utilisateur peut lui même créer ces propres tactiques, notamment quand il s'agit d'un groupement de tactiques répétitif.

Enfin, les assistants de preuve sont généralement accompagnés de nombreuse librairies : c'est une extension d'un outil. Pour les assistants de preuve, ces librairies sont souvent des théories mathématiques.

La vérification des preuves est donc effectuée par le noyau de l'outil. La confiance en la validité de la preuve repose sur deux éléments :

- les axiomes apportés par l'utilisateur,
- le noyau de l'assistant de preuve.

L'ajout de librairies ne modifie pas le noyau donc n'a pas d'influence sur la confiance.

1.3.2 Coq

Coq [Ber06] est un assistant de preuve développé par l'Inria depuis les années 80. Cet outil de recherche est encore développé aujourd'hui (version 8.13.2 au moment de la rédaction de ce manuscrit). Depuis sa création, de nombreuses librairies ont pu être ajoutées. Elles permettent d'avoir des raisonement mathématiques aussi développés que ceux dans les méthodes d'analyse de systèmes temps réel. Elles ont aussi permis la rédaction de preuves complexes comme le théorème des 4 couleurs [Gon07].

1.4 Bilan

Les réseaux embarqués peuvent être des systèmes critiques et des systèmes temps réel ce qui rend leurs vérification nécessaire. Les méthodes d'analyse permettent de décrire et prévoir les fonctionnements : elle permettent de déterminer des temps de traversée et de dimensionner les espaces mémoire nécessaires aux éléments réseaux. Le *Calcul réseau* est une méthode d'analyse pour des réseaux temps réel. Il donne des bornes sur les propriétés que nous voulons assurer sur un réseau embarqué. De telles certifications ont déjà été introduites dans les avions modernes. La vérification de ces derniers est donc aussi importante que les procédés qu'ils contrôlent.

Cependant, des erreurs peuvent apparaître dans la certification des bornes. Ce risque d'erreur fait baisser le niveau de confiance que nous pouvons avoir dans les bornes obtenues et rendent donc la certification longue. Nous décrivons deux sources d'erreurs. La première est la méthode d'analyse : si la méthode permettant de calculer une borne est incorrecte, le résultat concret sera incorrect lui aussi. Il faut donc vérifier dans un premier temps cette partie. La deuxième source d'erreur est l'implémentation de cette méthode. En effet, même une méthode vérifiée peut être mal implémentée. Une vérification de l'algorithme implémentant la méthode est donc aussi nécessaire.

Il existe des outils informatiques pour vérifier et augmenter le niveau de confiance. Cet ensemble d'outils permet de formaliser des propriétés pour raisonner logiquement. Les assistants de preuve font partie de ces outils et ils permettent de formaliser des théories mathématiques.

Cette étude consiste donc à formaliser la méthode d'analyse utilisée dans les réseaux embarqués : le *Calcul réseau*. Pour cela, les preuves mathématiques de la méthode seront traduites dans le langage de l'assistant de preuve Coq. Ensuite, pour vérifier l'implémentation de cette méthode, nous développerons un critère visant à vérifier que le calcul de valeurs effectives sur la méthode est correct. Ce critère sera lui aussi traduit en *Coq* donnant ainsi le même niveau de confiance.

1.4.1 Contribution

Dans un premier temps, l'objectif est de formaliser en Coq les définitions du *Calcul réseau* afin de vérifier la base mathématique de la méthode. Pour cela, l'idée est de formaliser les structures algébriques de *dioïde* et *dioïde complet* afin de formaliser un maximum de propriétés de l'algèbre *(min, +)*. Pour cela, nous utilisons des librairies existantes en Coq. Le but est de créer une librairie que nous pourrons réutiliser dans nos développement mais aussi qui sera réutilisable comme une extension aux librairies existantes.

L'objectif de la deuxième partie est de formaliser en Coq les bornes obtenues par le *Calcul réseau* sur les réseaux embarqués. Pour cela, l'idée est de décrire les notions et définitions propre à cette théorie en Coq. Cette formalisation nous permettra d'apporter un niveau de confiance supérieur dans la bonne construction de cette théorie. Nous formaliserons donc les preuves Coq des théorèmes principaux du *Calcul réseau* afin de les appliquer à un cas d'étude. Le cas d'étude imaginé reprend des éléments qui se retrouvent dans les réseaux embarqués dans les avions modernes. Ces deux premières parties ont fait l'objet d'un article [RBR19] dont nous étendrons les explications et résultat dans ce manuscrit.

L'objectif de a dernière partie de notre raisonnement va consister à introduire et développer des critères permettant la vérification et validation des calculs effectués en *Calcul réseau*. En effet, les expressions obtenues par la méthode sont algébriques et donc le calcul de valeurs concrètes doit faire appel à des algorithmes de calcul complexes. Pour cela, l'idée est d'établir des critères de vérification autour des algorithmes afin de vérifier la validité du calcul. L'idée est de developper ces critères en Coq afin d'obtenir le meme niveau de confiance que les preuves précédentes. Cette troisième partie a aussi fait l'objet d'un article paru dans [RRB21].

1.4.2 Démarche et plan

Avant de commencer à décrire nos contributions, nous donnerons dans le Chapitre 2 un état de l'art sur la vérification à l'aide de méthode formelle de propriétés temporelles dans un système ou un réseau temps réel.

Pour permettre la lecture de ce manuscrit, nous donnons dans le Chapitre 3 une introduction à la théorie du *Calcul réseau*. Cette introduction informelle permet de comprendre comment la théorie est utilisée pour déterminer des bornes dans les réseaux embarqués. Pour la même raison, nous introduirons l'assistant de preuve Coq dans le Chapitre 4. Nous expliquerons autour d'un exemple comment Coq permet d'assister la rédaction de preuve. Nous donnerons aussi la liste des librairies utilisées dans ce manuscrit.

Enfin, les Chapitres 5 et 6 sont les chapitres dédiés respectivement à la formalisation de la structure algébrique et à la formalisation de la théorie du *Calcul réseau*. Ces deux chapitres se terminent par une étude de cas. Cette étude de cas est ensuite reprise dans le Chapitre 7. Nous donnerons dans ce chapitre nos critères pour pouvoir vérifier les calculs effectués pour obtenir des valeurs concrètes.

État de l'art

Nous voulons connaître l'état des connaissances autour de la vérification des propriétés temporelles sur les réseaux embarqués introduites dans le chapitre précédent. Pour cela, nous allons donc parler de systèmes temps réel. En effet, les réseaux embarqués et le calcul du temps de traversée ressemblent beaucoup à l'analyse des temps de réponse sur les systèmes temps réel.

Nous voulons avoir confiance en ces méthodes d'analyse. De manière générale, il existe des processus de relecture humains pour éviter de commettre des erreurs. Cette étude ne porte pas sur de tels aspects car des erreurs humaines peuvent arriver. De plus, ces erreurs justifient l'utilisation de méthodes visant à formaliser cette partie. Cette étude porte sur l'utilisation de telles méthodes.

Dans cet état de l'art, nous allons présenter un peu plus en détail les systèmes temps réel et donc effectuer un historique rapide autour des méthodes d'analyse. Dans cet historique, nous joindrons les analyses communes entre les réseaux embarqués et les systèmes temps réel. Nous présenterons ensuite les justifications des formalisation apportées à ces méthodes, en faisant un historique des erreurs trouvées sur ces méthodes. Nous allons terminer par la présentation des formalisations et vérifications qui ont déjà été effectuées sur les méthodes d'analyse de systèmes temps réel et réseaux embarqués.

2.1 Systèmes Temps réel

Les systèmes temps réel sont des systèmes dans lesquels le temps nécessaire pour fournir un service ou un résultat est aussi important que la validité de ce résultat. Plusieurs niveaux d'importance sont accordés au temps déclinant ainsi plusieurs catégories de systèmes temps réel. Les systèmes temps réel souple sont une catégorie pour laquelle le service doit être rendu dans un temps qui n'est pas trop éloigné de celui imparti. Dans une seconde catégorie, les systèmes temps réel dur, le temps imparti ne peut pas être dépassé.

Nous avons expliqué en introduction que nous étions dans un contexte aéronautiques : les systèmes que nous allons étudier sont critiques, et donc dans la catégorie des systèmes temps réel dur. Le dépassement d'un service d'un système n'est pas tolérable. Donc il est important de bien connaître le fonctionnement du système et notamment de réussir à déterminer si les contraintes temporelles sont respectées dans tous les cas.

Cette analyse du respect des contraintes temporelles doit prendre en considération l'ensemble dans lequel le système interagit. Dans notre domaine, le système interagit avec un ensemble de mémoires, processeurs ou périphériques. Ce partage doit être connu pour déterminer le temps d'occupation de ces ressources partagées.

2.1.1 Tâche, échéance, ordonnancement

L'objectif est donc d'être capable de déterminer si un système temps réel peut remplir sa fonction dans le temps imparti. Pour cela, un modèle est nécessaire. Celui ci doit permettre de représenter le service que rend le système en fonction du temps, avec une date de début et une date de fin.

Dans les systèmes informatiques, les services que doit rendre le système sont découpés en tâches. Ces tâches sont des processus de calcul : elles utilisent un processeur pour être exécutées. Donc à chaque tâche est associée une quantité de calcul. Cette valeur représente la quantité de calcul dont la tâche a besoin pour être exécutée par le processeur.

Chaque tâche dispose d'une date de réveil : date à laquelle elle est prête à être exécutée. Cette notion s'appelle la date d'arrivée d'une tâche. Cette date précède forcément la date réelle d'exécution de la tâche.

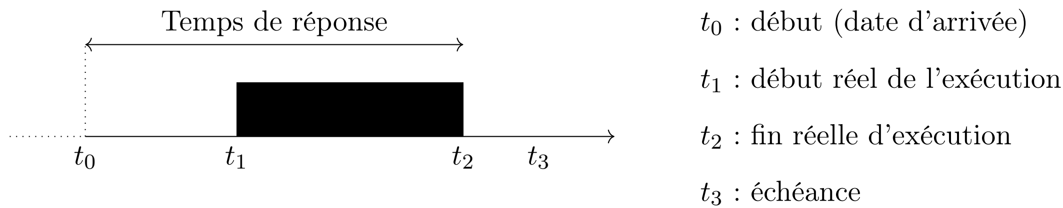


FIGURE 2.1 – Exemple de représentation d'une tâche

Elle possède ensuite un temps d'exécution : une durée entre le date de début et la fin de celle-ci. Une tâche a également une échéance : une date avant laquelle elle doit avoir été exécutée. Enfin, le temps de réponse d'une tâche est la durée entre le début de la tâche et la fin réelle de son exécution.

Nous traçons en Figure 2.1 la représentation d'une tâche. Le temps est décrit en abscisse. Nous avons noté les différents termes expliqués au paragraphe précédent sur cette figure.

Dans certains cas, les tâches sont classées selon plusieurs critères : importance, ressource utilisée... Il est alors assigné à chaque tâche une priorité. Cette priorité peut être fixée lors de la conception du système ou alors être modifiée au cours de l'évolution, comme pour l'accès à une ressource partagée par exemple.

Au sein du système, l'ordonnanceur permet de définir dans quel ordre les tâches doivent s'exécuter. Il peut choisir d'éxécuter les tâches par dans un certain ordre de priorité ou selon l'échéance la plus proche. Ces règles sont décrites dans l'algorithme d'ordonnancement. Cet ordonnancement peut être statique, fixé à l'avance, ou dynamique, déterminé au cours de l'exécution du système.

2.1.2 Réseaux temps réel

Les réseaux temps réel ressemblent aux systèmes temps réel : l'ordonnancement de messages dans un réseau s'apparente à l'ordonnancement de tâches sur un processeur. Dans un réseau, les messages peuvent être modélisés comme des tâches et ils utilisent des serveurs pour parcourir le réseau. Ces serveurs ont un débit qui permet de traiter un certain nombre de messages en fonction du temps. Une fois que le message a traversé le serveur, nous disons qu'il a été servi. L'ordonnancement d'un serveur permet de déterminer une priorité dans les messages qu'il doit traiter.

Dans un réseau, la gigue est la variation du délai que peut apporter un serveur. Notre définition de la gigue dans un réseau temps réel est la différence entre le délai maximal et minimal que peut apporter un serveur pour un message. Une telle variation n'est pas défini de cette manière dans les systèmes temps réel. De plus, les messages occupent un espace mémoire non fini. Dans un système temps réel, la tache ne prend pas plus d'espace mémoire quand elle est active ou inactive alors qu'un message utilise un buffer pour être stocké avant de le libérer. Ces différences expliquent pourquoi il n'est pas possible d'utiliser directement les mêmes méthodes sur les systèmes temps réel et sur les réseaux temps réel.

2.2 L'analyse des systèmes temps réel

L'analyse de systèmes temps réel permet d'étudier les temps de réponse des tâches. L'objectif de cette analyse est de +déterminer si chaque échéance est respectée. Pour cela, il existe trois +grandes familles de méthodes : les méthodes d'analyse de temps de +réponse, les méthodes à base de courbes de charge, et les méthodes à +base de trajectoires, succinctement évoquées dans la section 1.2.

Méthodes de calcul des pires temps de réponse

Une première méthode est donnée par LIU et LAYLAND [LL73]. Ils définissent le facteur d'utilisation d'un processeur en fonction des quantités de calcul et du temps d'activation de chaque tâche. Si ce facteur ne dépasse pas un certain seuil, la méthode garantit que le système est ordonnançable : toutes les tâches respecteront leurs échéances. Ces travaux donnent ainsi un test d'ordonnaçabilité pour des systèmes temps réel sous les hypothèses que les priorités sont constantes et les tâches sont non préemptibles. de plus, pour obtenir un tel critère, les auteurs définissent un instant critique : un

instant au cours duquel l'activation d'une tâche se produit en même temps que l'exécution de tâches plus prioritaires. Plus tard, les travaux dans [DG00] expliqueront que les hypothèses données par Liu et Layland n'étaient pas assez complètes.

Lehoczky [Leh90] propose une méthode exacte pour déterminer exactement le temps de réponse d'une tâche. L'auteur permet ainsi de déterminer si un ensemble de tâches périodiques contraintes par des échéances est ordonnançable. Cette méthode d'analyse est aussi connue sous le nom de *Response Time Analysis* (RTA) . De tels critères ont aussi été étudiés par Joseph et Pandya dans [JP86].

Les tâches peuvent partager des ressources communes comme des sémaphores. Lors de ce partage, un blocage peut subvenir lorsque une tâche moins prioritaire bloque l'accès à un de ces sémaphores pour une tâche plus prioritaire : cette tâche moins prioritaire devient plus prioritaire qu'une tâche qui à l'origine était plus prioritaire revenant ainsi à inverser la priorité. Pour éviter une telle inversion, Sha, Rajkumar et Lehoczky présentent dans [SRL90] le protocole *Priority Inheritance Protocols.* Ce protocole vise à augmenter la priorité d'une tâche qui bloquerait un sémaphore. Dans [SRL90], les auteurs donnent aussi une méthode pour analyser si le système est ordonnançable.

Pour Mok et Chen dans [MC97], la période des tâches n'est pas nécessairement une valeur fixe pour une tâches mais peut prendre un ensemble de valeurs. Ce phénomène sporadique permet d'élargir la modélisation des systèmes.

Dans les analyses que nous venons de citer, l'ordonnancement des tâches repose sur une affectation de priorités fixes aux tâches. Zhang et Burns proposent dans [ZB09] une méthode d'analyse pour un système ordonnancé en *Earliest Deadline First* (*EDF*) : les tâches dont l'échéance est la plus proche sont les plus prioritaires.

Le *Controller Area Network* (CAN) est un réseau embarqué très utilisé depuis les années 90 dans l'automobile et dans l'aéronautique. En 1994, Tindell, Hanssmon et Wellings dans [THW94] établissent une méthode pour analyser les temps de traversée dans un réseau CAN. Cette méthode se base sur RTA avec les résultats de Joseph et Pandya étendus. Plus tard, les travaux dans [Dav+07] montreront que les preuves de cette méthode sont parfois erronées et que le pire temps de traversé peut se montrer optimiste.

Méthodes à base de courbes d'arrivée

Comme nous l'avons évoqué en introduction à ce manuscrit, les méthodes pour obtenir des bornes ont l'avantage d'être réalisables sur des grands problèmes. Ces méthodes sont à base de courbe d'arrivée comme l'*event stream.* Cette méthode, premièrement présenté par Gresser avec [Gre93], est une technique de modélisation de systèmes temps réel pour décrire le comportement temporel d'évènements. Des fonctions d'arrivée sont utilisées pour compter le nombre d'évènements se produisant.

En 2000, Thiele, Chakraborty et Naedele dans [TCN00] utilisent les théories *Max-plus* et du *Calcul réseau* pour construire une méthode d'analyse nommé *Real-time Calculus* (RTC). Le modèle utilise des courbes cumulatives pour décrire des quantités de calcul demandés et/ou délivrés. Thiele et al. bornent ainsi deux quantités. La première est la quantité de calcul délivrée, i.e la quantité de calcul qui a pu être éxécuté à un temps t. La seconde est la quantité de calcul restante à être exécuté à un temps t.

En partant des travaux sur les *event stream* et des travaux sur RTC que nous venons de citer, Richter [Ric05] établit dans sa thèse l'analyse de performances compositionnelle (CPA). L'idée de l'analyse est de réaliser des analyses d'ordonnancement locales pour ensuite les propager.

De nombreux outils viennent alors s'intégrer dans les méthodes CPA et RTA. A l'université technique de Braunschweig, des outils comme SysmTA/S [Hen+05] et pyCPA [DAE12] font leurs apparition. Ces outils permettent d'analyser l'ordonnancement des *Systems on Chip* ou des systèmes distribués.

Méthodes à base de trajectoires

D'autre part, Martin et Minet [MM06 ; MMG04] donnent une méthode d'analyse pour les réseaux embarqués appelée *trajectory approach.* Cette approche consiste à déterminer les temps de traversée de bout en bout du réseau en examinant l'ordonnancement réel de chaque noeud du réseau. Cette première approche vise à réduire le pessimisme amené par le *Calcul réseau.*

La thèse de BAUER [Bau11] donne ensuite une analyse du réseau AFDX en utilisant la *trajectory approach*. Il améliore aussi les résultats initiaux de Martin et Minet pour prendre en compte la sérialisation des serveurs et la politique d'ordonnancement First-in-First-out déjà présente dans les réseaux AFDX. Cependant, les travaux ultérieur présenté dans [Kem+13a] expliqueront que cette approche peut se montrer optimiste et calculer des valeurs plus petites que le pire délai.

2.3 Le *Calcul réseau*

Le *Calcul réseau* est une méthode d'analyse de réseau temps réel. C'est une méthode à base de courbe d'arrivée avec la particularité qu'elle utilise une algèbre tropicale : le *dioïde* des fonctions *(min, +)*. Cette structure algébrique est présentée par MINOUX et GONDRAN dans [MG08].

Le *Calcul réseau* existe en deux versions. L'une d'elle est déterministe, elle concerne plutôt les réseaux embarqués à temps réel dur, et l'autre stochastique, pour du temps réel plus souple. Donc nous prenons comme version dans ce manuscrit la version déterministe. Nous citons toutefois quelques outils utilisant la version stochastique.

Les bases de la théorie du *Calcul réseau* ont été apportées par CRUZ dans [Cru91]. Puis, LE BOUDEC et THIRAN décriront dans [LBT01] l'association avec le *dioïde* des fonctions *(min, +)*. Ces travaux seront ensuite repris pour aboutir à l'ensemble d'outils et théorèmes présentés par BOUILLARD, BOYER et LE CORRONC dans [BBLC18]. Cette méthode est utilisée dans cette thèse et nous donnons dans le Chapitre 3 une introduction plus détaillée. Nous donnons dans la suite un présentation des usages du *Calcul réseau* et une liste des logiciels de calcul utilisant cette méthode.

Usage du *Calcul réseau*

Le *Calcul réseau* a été adapté pour plusieurs standards. Le plus connu d'entre eux est le standard *Avionics Full DupleX* (AFDX) : un réseau Ethernet embarqué sur de nombreux avions depuis l'A380. L'application du *Calcul réseau* sur des réseaux AFDX a été montrée par GRIEU dans [Gri04].

Le *Calcul réseau* est aussi utilisé pour de l'*Audio Video Bridging* (AVB) par AZUA et BOYER dans [AB14]. Ce standard sert à transférer en temps réel des flux multimédia.

DAIGMORTE, BOYER et ZHAO donnent une modélisation d'architecture *Time-Sensitive Networking* (TSN) qui utilise le *Calcul réseau* dans [DBZ18] et [BD19]. Dans ces travaux, les auteurs font un lien entre plusieurs standards dont le *Credit Based Schaper* ou le *Time-Triggered Shaper*.

Outils du *Calcul réseau*

Nous appelons outils du *Calcul réseau* dans ce manuscrit tout logiciel qui permet de calculer des valeurs concrètes sur la méthode du *Calcul réseau*. En effet, la méthode permet d'obtenir des expressions algébriques à partir d'un réseau en particulier et il faut donc utiliser un algorithme pour calculer des valeurs sur ces expressions.

Le *Calcul réseau* a été implémenté par de nombreux outils. Le premier outil que nous citons est une librairie Java développée premièrement par SCHMITT et ZDARSKY dans le projet DISCO [SZ06]. Elle continue à être développée par une équipe de l'université de Kaiserslautern. Il existe une version de la librairie, DISCO-SNC, permettant de faire du *Calcul réseau* stochastique [BS13] développée par BECK et SCHMITT.

Les membres du projet COINC (pour *COmputa-tional Issue in Network Calculus*) ont développé une *toolbox* pour SCILAB qui permet de calculer des valeurs effectives sur les fonctions du *dioïde (min, +)*. Dans l'algorithme, présenté en [BT08], les fonctions utilisées sont un sous ensemble de $\mathcal{F}$. En effet, le comportement souvent périodique et linéaire par morceaux des fonctions du *Calcul réseau* permet de regrouper et classifier les fonctions.

Le projet PEGASE [Boy+10] est une association entre des centres de recherches français et la société *RTaW* au sein duquel a été développé une librairie Java pour les réseaux embarqués aéronautiques et aérospatial utilisant le *Calcul réseau*. La librairie fonctionne pour des réseaux AFDX, des réseaux SpaceWire (un standard aérospatial) et des *Network On Chip*. De plus *RTaW* met à disposition le *Online Min-Plus Interpreter*. Cet outil en ligne est un interpréteur de commande qui permet de faire

des calculs avec le *dioïde* des fonctions *(min, +)*, et donc de faire du *Calcul réseau*. Il est développé par RTaW et est disponible à l'adresse donnée en [Rea10].

Wandeler a développé avec sa thèse et les travaux de Thiele, Chakraborty et Naedele [TCN00] une *toolbox* Matlab appelée *Real-Time Calculus Toolbox* [Wan06 ; WT06]. Cette librairie implémente une théorie proche du *Calcul réseau*, le *Real-Time Calculus* déjà cité dans ce chapitre.

Bisti et al. ont travaillé sur la librairie *DElay BOund Rating AlgoritHm* (Deborah). Elle est développée en C++ et permet de calculer des bornes en utilisant le *Calcul réseau* sur un réseau *FIFO* [Bis+10].

Mifdaoui et Ayed donnent dans [MA10] des résultats sur le dévelopement de l'outil *Worst-case Performance Analysis of Embedded Networks* (WoPANets). Cet outil Java utilise le *Calcul réseau* et des techniques d'optimisation pour analyser des réseaux embarqués.

Schioler, Schwefel et Hansen ont développé la librairie *Cyclic Network Calculus* (CyNC) [SSH07]. Cette *toolbox* Matlab utilise l'environnement Simulink. Elle permet de modéliser (graphiquement avec Simulink) et d'analyser des réseaux embarqués avec le *Calcul réseau*.

Enfin, nous citons les travaux de Boyer sur NC-Maude [Boy10]. Cette librairie propose un outil pour tester directement des propriétés du *Calcul réseau* grâce à l'expressivité du langage utilisé.

2.4 Méthodes formelles sur systèmes temps réel et réseaux

Nous présentons dans cette partie les travaux de formalisations de systèmes temps réels avec des méthodes formelles. Dans un premier temps, nous citons les motivations à réaliser de tels travaux.

Devillers et Goossens montrent dans [DG00] que le raisonnement de Liu et Layland n'est pas complet. Le résultat n'est ici pas invalidé mais démontre que la preuve donnée par Liu et Layland dans [LL73] manquait de détail.

Dans les systèmes temps réels, les tâches auto-suspendues sont des tâches dont l'ordonnancement consiste à décrire des régions d'exécution et des régions de suspension. Dans [Nel+15], Nelissen et al. invalident une analyse d'ordonnancement de tâches auto-suspendues. Pourtant, l'analyse originelle a été très utilisée et généralisée depuis sa publication. Les auteurs utilisent un contre exemple avec plusieurs scénarios pour montrer l'erreur.

Dans [Dav+07], Davis et al. montrent que l'analyse de réseaux temps réel CAN est défectueuse : la méthode pour ordonnancer les messages dans le réseau présente des erreurs. Ces erreurs sont apparues dans les preuve papier des calcul pour les pires cas de réponse.

Les travaux de Kemayo et al. [Kem+13a ; Kem+13b] montrent que l'analyse initiale des *trajectory approach* de Martin et Minet était trop optimiste et Bauer trop pessimiste. La méthode de sérialisation [Kem+13a] présente ce problème. Le calcul du pire dernier départ possible présente aussi ces caractéristiques [Kem+13b].

L'ensemble de ces travaux sert de motivation à l'ensemble des formalisations effectuées. En effet, la supervision d'un ordinateur dans la vérification est un moyen efficace pour éviter les erreurs dans les raisonnements logiques. La confiance repose alors, entre autres, sur l'outil informatique utilisé.

2.4.1 *Model checking* sur des propriétés temporelles

Nous allons citer dans cette partie les travaux concernant la vérification de propriétés temporelles sur les systèmes et réseaux temps réel avec le *model checking*. Le *model checking* consiste à vérifier des propriétés en explorant tous les comportements possibles d'un modèle. Cette méthode de vérification fait aussi appel à des outils et nous citerons, pour chaque vérification de propriétés temporelles, l'outil utilisé.

Adnan et al. dans [Adn+10] utilisent les automates temporisé avec l'outil UPPPAL [LPY97] pour déterminer des temps de traversée pire cas. Les auteurs utilisent pour cas d'étude un réseau AFDX et peuvent donc comparer leurs résultats avec le *Calcul réseau*. Dans cette comparaison, le *Calcul réseau* donne des résultats plus pessimistes. Une amélioration de l'approche est donnée par les mêmes auteurs dans [Adn+12]. Le réseau AFDX étudié dans cette évolution est composé de 32 *virtual links* (le nom donné aux canaux de communication en AFDX). Cependant, même avec cette version améliorée, une explosion combinatoire arrive dès que la taille du réseau est augmentée.

Dans [MA14], Mouradian et Augé-Blum étudient une méthode applicable sur des réseaux sans fils hybride entre le *Calcul réseau* et une vérification avec du *model checking*. Dans ces travaux, les auteurs utilisent une version du *Calcul réseau* pour modéliser et déterminer des contrats sur le réseau sans fils puis apportent un algorithme utilisant UPPAAL pour vérifier que chaque noeud du réseau est capable de rendre un service en dessous d'une certaine durée.

L'*Integrated Modular Avionics* (IMA) se base sur un réseau informatique. Ce réseau doit aussi répondre à des contraintes temporelles. Lauer et al. [Lau+10] utilise le *model checking* pour déterminer, entre autres, des délais bout en bout. Dans une telle étude, il faut prendre en compte les délais apportés par les éléments réseau mais aussi par les calculs effectués.

Dans les réseaux embarqués, *TTEthernet* est une application des *Time triggered* architecture. Dans une telle architecture, les entités fonctionnent et communiquent à l'aide d'une horloge commune. Des algorithmes permettent alors de synchroniser précisément ces horloges. Dans [SD11], Steiner et Dutertre amènent une automatisation sur la formalisation d'un tel algorithme.

2.4.2 Formalisation avec assistant de preuve de méthodes d'analyse

Nous allons maintenant présenter les formalisations des méthodes d'analyse de système temps réel. Cette formalisation est, comme dans notre cas, effectuée par un assistant de preuve.

Des premiers travaux menés par Wilding avec l'assistant *NQTHM* ont été présenté dans [Wil98]. Cet assistant de preuve est une version plus ancienne de *ACL2*, un assistant de preuve très largement utilisé par des organismes comme la *NASA* pour la vérification logicielle. Dans [Wil98], l'auteur présente une preuve formelle avec *NQTHM* d'un théorème sur l'ordonnançabilité d'un systèmes temps réel en *EDF*.

L'assistant de preuve *Prototype Verification System* (PVS) est un assistant de preuve en logique classique implémenté en Lisp [Owr+92]. En 2000, Dutertre réalise une formalisation d'une méthode d'analyse de système temps réel avec un protocole de gestion de ressources *Priority Ceiling Protocol* en utilisant PVS [Dut00]. Cette formalisation provient de travaux manuscrits réalisés par Sha, Rajkumar et Lehoczky [SRL90] que nous avons déjà cité dans ce Chapitre.

A l'aide de l'assistant de preuve Coq [Ber06], une formalisation d'une preuve papier est développée par De Rauglaudre dans [DR12]. Il s'agit d'une vérification formelle de conditions d'ordonnancabilité de tâches temps réel périodiques strictes.

Dans la continuité, le projet PROSA est un projet de Cerqueira, Stutz et Brandenburg en Coq autour de l'analyse d'ordonnancement de systèmes temps réel [CSB16 ; RC18]. Le principe est de donner un outil d'analyse d'ordonnancement réutilisable et facile à comprendre même pour un non connaisseur de Coq. Ces travaux ont déjà été repris de nombreuses fois. Nous pouvons citer [Fra+18] dans lequel Fradet et al. utilisent PROSA pour obtenir une analyse du pire cas typique (noté TWCA dans la littérature) sur des systèmes a priorité fixe avec préemption ou non, noté FPP ou FPNP respectivement, et par priorité de délai (EDF).

Guo et al. intègrent les travaux de PROSA dans un OS temps réel vérifié : *RT-CertiKos* [Guo+19]. Ces travaux permettent d'avoir un OS avec des notions temps réel vérifiées mais aussi avec des certifications de sémantique et compilation. Cette partie est assurée par des travaux en Coq autour du compilateur CompCert [Ler14] et du système d'exploitation CertiKOS [Gu+16].

Toujours avec *Coq*, Fradet et al. présentent CertiCAN : un outil pour certifier l'analyse de réseaux CAN [Fra+19]. Cet outil combine deux méthodes d'analyse de réseau CAN, une approximative mais rapide et l'autre précise mais lente.

Dans [Mab+13a ; Mab+13b], Mabille et al. utilisent *Isabelle*, un assistant de preuve proche de *Coq*, pour prouver et certifier des résultats et calculs du *Calcul réseau*. *Isabelle* permet de vérifier que les propriétés dans les preuves de la méthode sont appliquées correctement.

Chapitre 3

Introduction au *Calcul réseau*

Comme nous l'avons introduit au Chapitre 1, le *Calcul réseau* est une méthode d'analyse pour les réseaux temps réels. Cette théorie utilise des modélisations de quantités de données. Cette technique de modélisation a été initiée par les travaux de Cruz [Cru91] en 1991. Les quantités de données sont des messages ou des parties de messages. Le contenu de la donnée n'est pas pris en compte. À l'aide de cette modélisation, le *Calcul réseau* permet d'établir des bornes sur les temps de traversée des messages et sur la quantité de données qu'un élément du réseau doit pouvoir contenir.

Pour bien comprendre les notations, la prochaine définition englobe les éléments mathématiques minimums pour comprendre cette introduction.

Définition 3.1 (Ensembles $\mathbb{R}_+$, $\overline{\mathbb{R}}$ et $\mathcal{F}$) Nous utilisons $\mathbb{R}_+$ pour l'ensemble des réels positifs ou nuls et $\overline{\mathbb{R}}$ pour les réels étendues, c'est-à-dire $\mathbb{R} \cup \{+\infty, -\infty\}$. Enfin, nous définissons les fonctions $\mathcal{F}$ de $\mathbb{R}_+$ à $\overline{\mathbb{R}}$.

Fonctions cumulatives de données La modélisation de quantité de donnée s'effectue à l'aide de fonctions dans $\mathcal{F}$. Les fonctions cumulatives de données en *Calcul réseau* permettent de représenter la quantité cumulée de donnée à un endroit précis du réseau en fonction du temps. Ces courbes sont utilisées pour représenter un comportement réel du réseau. Nous donnons un exemple en Figure 3.1. Cette figure est une modélisation d'un point du réseau. À ce point-là, jusqu'à un temps t_1, aucun

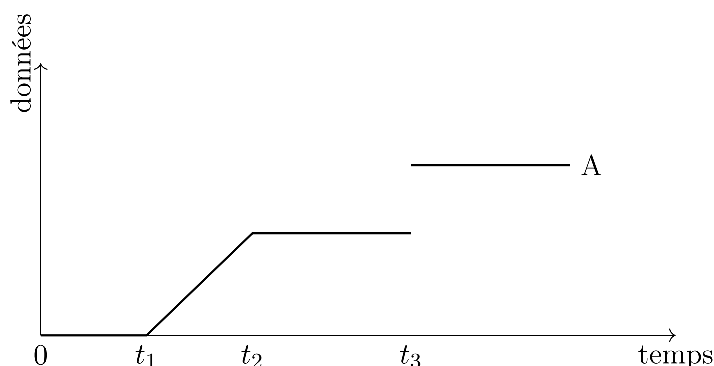


Figure 3.1 – Exemple de courbe cumulative de donnée : la fonction A représente la quantité de donnée passée par un point du réseau.

message ne passe. Entre t_1 et t_2, des données passent constamment, entre t_2 et t_3, aucune donnée ne passe et à t_3, plusieurs données passent instantanément.

Serveur En *Calcul réseau*, les éléments du réseau sont des serveurs. Un serveur est une relation entre des paires de courbes cumulatives de données A et D pour arrivée et départ. Pour illustrer cette définition, nous donnons en Figure 3.2 un exemple d'un serveur avec un buffer interne. Dans cet exemple, un buffer interne, c'est-à-dire une mémoire tampon, permet de stocker les informations arrivées via A avant de les transmettre sur la sortie D à la vitesse R. Donc, la courbe cumulée des départs D dépend du service que peut rendre le serveur. Dans cet exemple, le service est le débit R.

Avec les courbes cumulatives A et D, il est possible d'obtenir le délai maximal subi par un message dans un serveur. Le délai subi à un instant t est la durée entre sa date d'entrée et sa date de sortie. Prenons par exemple la Figure 3.3. Sur celle-ci, nous avons pris le cumul de donnée en entrée (bleu) et en sortie (rouge) d'un serveur. Le pire délai reçu par un message est alors la plus grande distance horizontale entre les courbes A et D.

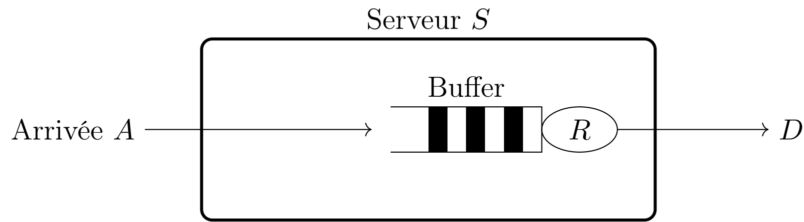


FIGURE 3.2 – Exemple d'un serveur S avec une arrivée A, un buffer, un débit R et un départ D. Les données entrantes arrivant en A sont stockés dans le *Buffer* avant d'être émise en D avec un débit R.

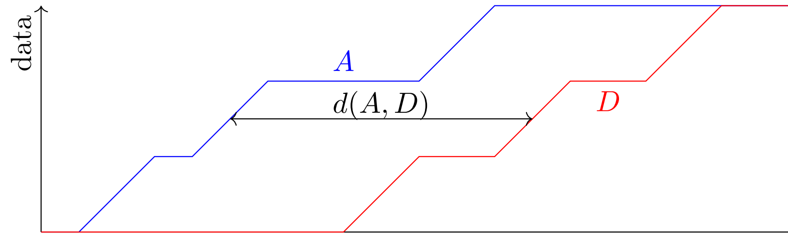


FIGURE 3.3 – Délai $d(A, D)$: la plus grande déviation entre A et D

Contraintes Une courbe cumulative de donnée représente un comportement réel du réseau. Ces comportements sont nombreux. Ils peuvent être englobés sous une contrainte qui rend leur manipulation plus facile.

Cette contrainte peut par exemple borner le cumul de donnée à un point : nous parlons alors de courbe d'arrivée maximale. Elle représente alors une borne maximale sur la quantité de donnée qui peut arriver en un point. Une courbe α est une courbe d'arrivée pour toute courbe cumulative A qui vérifie, pour tout t et $s \in \mathbb{R}_+$:

$$A(t + s) - A(t) \leqslant \alpha(s).$$

La traduction possible est que la quantité de donnée entre t et $t + s$ est plus petite que $\alpha(s)$. Un exemple de courbe d'arrivée pour une courbe cumulative A est montré en Figure 3.4.

Nous avons expliqué comment le *Calcul réseau* modélisait les données à travers le réseau. Le service que peut offrir un serveur est aussi contraint par ce que nous appelons une courbe de service notée β. Dans ce manuscrit, nous présenterons le service *(min, +)* et le service strict minimum. L'idée de ses deux contraintes est de donner une contrainte minimum du service que peut rendre un serveur à un flux entrant. Cette contrainte permet alors de déterminer une borne sur le départ de ce serveur. Nous donnons en Figure 3.5 un exemple de courbe de service.

Les contraintes de services *(min, +)* et d'arrivée sont définies formellement à l'aide de la convolution *(min, +)*, notée $*$. L'intuition à retenir pour les arrivées et les départs d'un serveur, est qu'ils sont

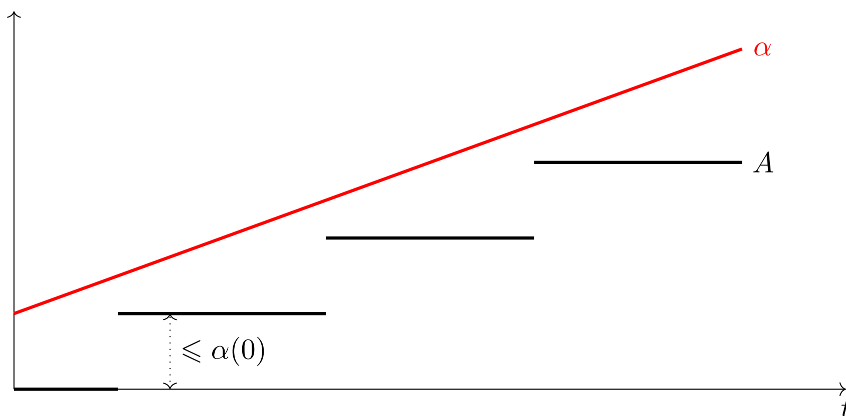


FIGURE 3.4 – Exemple courbe d'arrivée α pour une courbe d'arrivée A. Le point $\alpha(0)$ est supérieur au plus grand écart instantané de A pour envelopper la courbe A.

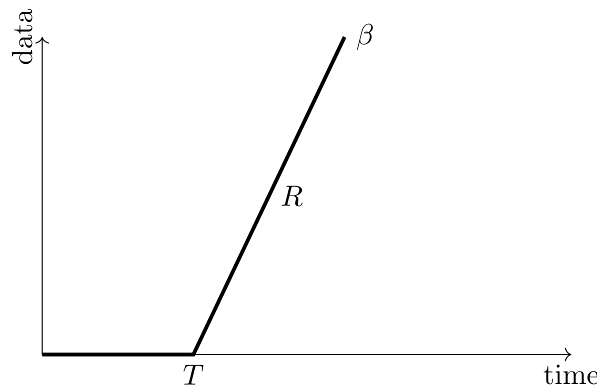


FIGURE 3.5 – Exemple de courbe de service : Rate-lantency. Le serveur peut ne traiter aucune donnée jusqu'à l'instant T. À partir de T, il traite au moins R donnée par période de temps.

respectivement contraints par une courbe d'arrivée α et une courbe de service β tels que

$$A \leqslant A * \alpha \quad \text{et} \quad A * \beta \leqslant D$$

Le produit de convolution *(min, +)* fait partie du *dioïde* des fonctions *(min, +)*. Un *dioïde* est une structure algébrique formée d'un ensemble et de deux lois de composition satisfaisant un ensemble de propriétés. Il est aussi appelé structure algébrique tropicale. Une telle association avec la structure algébrique et le produit de convolution est décrite dans l'ouvrage de LE BOUDEC et THIRAN [LBT01], en 2001. De cette manière, les courbes de contraintes en *Calcul réseau* peuvent être manipulées avec les propriétés du *dioïde*. Nous donnons dans le Chapitre 5 une définition des structures algébriques utilisées en *Calcul réseau* ainsi qu'aux fonctions et opérations associées.

Propagation Le *Calcul réseau* permet d'établir un contrat sur la sortie d'un serveur. Pour décrire un tel contrat, prenons un serveur avec une courbe de service β avec une arrivée contrainte par une courbe d'arrivée α. La sortie d'un serveur est contrainte par une courbe d'arrivée $\alpha \oslash \beta$ où $\oslash$ est l'opération de déconvolution du *dioïde* des fonctions *(min, +)*. Ainsi, il est possible de connaître aussi les contrats d'arrivée au fur et à mesure qu'un flux traverse le réseau. Nous donnons un exemple en figure 3.6.

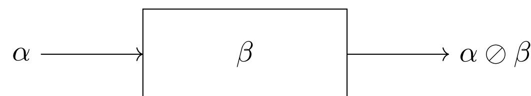


FIGURE 3.6 – Propagation des contraintes : exemple d'un flux contraint par la courbe d'arrivée α traversant deux serveurs de service β

Le *Calcul réseau* utilise cette modélisation pour décrire un serveur avec à gauche la modélisation des flux entrants et à droite les flux de sorties. Nous avons noté les contrats sur chaque élément : donc α pour l'unique courbe d'arrivée, β pour le serveur et enfin le contrat que donne le *Calcul réseau* en sortie du serveur.

Concaténation Une fois qu'il est possible de connaître le contrat de sortie d'un serveur, il est possible d'étendre ce contrat pour une entrée dans un autre serveur. Cette connexion de serveurs est une concaténation. Le *Calcul réseau* permet aussi d'étudier la concaténation de serveurs. Nous reprenons la modélisation de l'exemple précédent dans la Figure 3.7.

Le flux α traversant les deux serveurs de service β_1 et β_2 obtient, d'après le *Calcul réseau* et les propriétés du *dioïde* des fonctions *(min, +)*, le service $\beta_1 * \beta_2$. En reprenant la notion de délai, un message entrant dans ce réseau sera retardé d'au maximum $d(\alpha, \beta_1 * \beta_2)$. Le *Calcul réseau* permet aussi de montrer que ce contrat sur le retard est plus petit que le contrat calculé en utilisant la somme des délais, soit $d(\alpha, \beta_1)$ et $d(\alpha \oslash \beta_1, \beta_2)$.

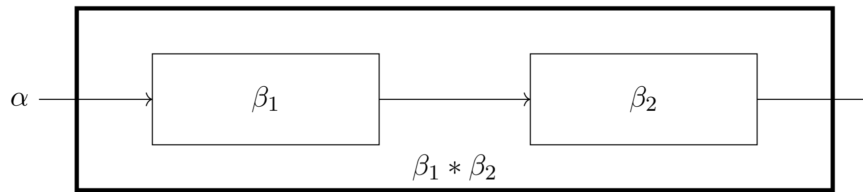
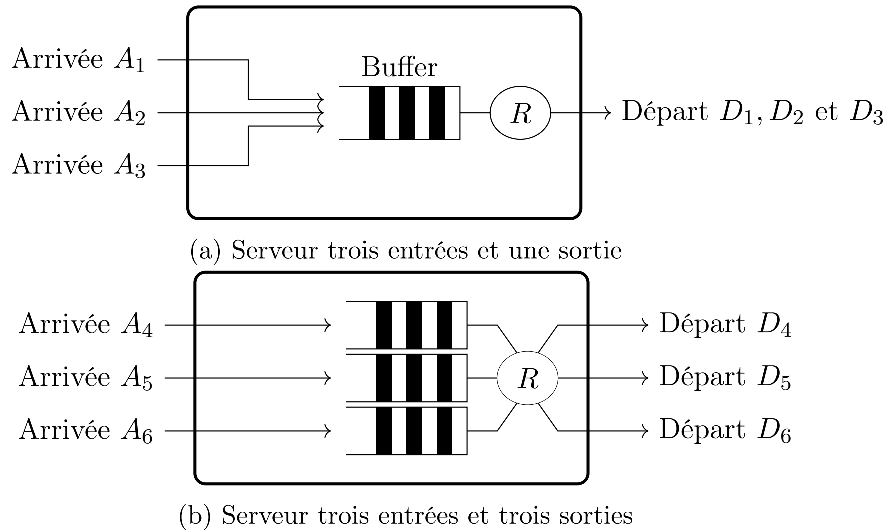


FIGURE 3.7 – Propagation des contraintes : exemple d'un flux contraint par la courbe d'arrivée α traversant deux serveurs de service β_1 et β_2.



(a) Serveur trois entrées et une sortie

(b) Serveur trois entrées et trois sorties

FIGURE 3.8 – Deux serveurs avec plusieurs entrées pour plusieurs sorties. Le nombre exact est inscrit au-dessous de chaque serveur. Le serveur trois entrées et une sortie (au-dessus) est une agrégation de trois flux entrants dans un unique buffer et celui-ci est vidé avec un débit R sur la sortie contenant les départs D_1, D_2 et D_3. Le serveur trois entrées et trois sorties (au-dessous) est un serveur contenant trois buffers remplis par chacun des flux entrants respectivement. Les buffers sont vidés sur chacune des sorties avec un débit R.

Plusieurs flux traversant un serveurs L'intérêt d'un serveur est de pouvoir traiter plusieurs entrées et sorties, de les associer ou les séparer. Il existe alors plusieurs configurations possibles : par exemple, deux flux remplissant un même buffer qui est ensuite émis sur une sortie ou une entrée remplissant plusieurs buffers qui sont ensuite transmis sur une sortie chacun. Nous en illustrons et commentons certains cas dans la Figure 3.8.

Mathématiquement, le *Calcul réseau* a plusieurs techniques pour ramener tous ces types de serveurs à une modélisation d'un serveur à une entrée et une sortie. Pour cela, le *Calcul réseau* dispose de notions d'agrégation de serveur (regroupement) et de serveur résiduel. En pratique, la puissance globale d'un serveur est connu. Ce service β est un service pour le serveur agrégé, il va servir l'ensemble des flux. En reprenant la Figure 3.8, ce service est donné par le débit R. Le principe est de déterminer un service pour chaque flux : par exemple pour le flux arrivant sur A_1 et sortant en D_1. Ce service résiduel β_i servira à déterminer les bornes pour chaque flux.

Dans le cas d'un buffer rempli par plusieurs flux, celui-ci va peut donner une priorité différente dans l'émission des informations arrivant dans ce buffer. Cet ordre est décrit par sa politique d'ordonnancement. Le *Calcul réseau* permet ensuite de déterminer le service résiduel. Nous donnons une intuition continuant la précédente figure en Figure 3.9.

Dans cet exemple, le serveur peut offrir un service β à trois flux le traversant : le flux d'entrée A_1 contraint par α_1 et de sortie D_1, le flux d'entrée A_2 avec respectivement α_2 et D_2 et le flux d'entrée A_3 avec respectivement α_3 et D_3. Le *Calcul réseau* permet alors d'établir un service offert à chacun de ces flux : donc β_1 pour A_1, β_2 pour A_2 et β_3 pour A_3. Ces courbes de service pour chaque flux correspondent à un service résiduel : chacun est dépendant des autres entrées. Dans ce cas précis et ne connaissant pas la politique d'ordonnancement du serveur, le *Calcul réseau* nous permet d'établir que :

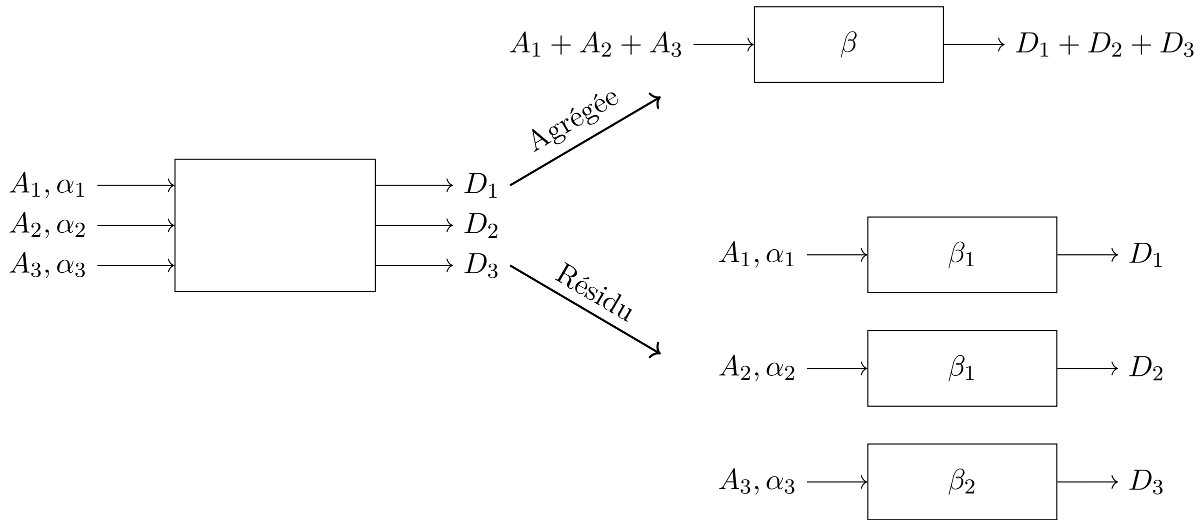


FIGURE 3.9 – Service résiduel pour le serveur 3 entrées et une sortie de la Figure 3.8

$\beta_1 = [\beta - (\alpha_2 + \alpha_3)]^+$, $\beta_2 = [\beta - (\alpha_1 + \alpha_3)]^+$ et $\beta_3 = [\beta - (\alpha_2 + \alpha_3)]^+$. Nous décrirons l'opération $[f]^+$ pour f une fonction de $\mathcal{F}$ au cours de ce manuscrit. L'intuition derrière cet opérateur est qu'il consiste à ne prendre que la partie positive et croissante de la fonction f.

Le service résiduel est dépendant des courbes d'arrivées des différents flux entrants et de la politique d'ordonnancement. Les travaux de LE BOUDEC et THIRAN [LBT01] présentent des résultats autour de la politique de *First in First out* (*FIFO*). Les travaux de BOUILLARD et JUNIER [BJ11] décrivent des bornes pour des serveurs avec une politique à priorité fixe entre les flux. Une autre approche se basant sur un modèle de *Packetisation* ont été mené par BOUILLARD, FARHI et GAUJAL [BFG11] pour permettre de mener une analyse avec le *Calcul réseau* sur une politique de service en *Round Robin*.

Outils et algorithmes Dans ce manuscrit, nous appelons *Outils* un programme informatique développé à partir d'un algorithme lui-même provenant d'une méthode d'analyse. En soi, l'outil permet d'obtenir des valeurs effectives à partir des valeurs du problème (taille messages, fréquence, débit serveur pour le réseau) et des résultats algébriques de la méthode ($\alpha \oslash \beta$ pour le *Calcul réseau*).

La méthode d'analyse que nous venons de présenter dispose de nombreux outils. Nous donnons une liste dans le Chapitre 2. Nous donnons dans ce même chapitre une liste des technologies sur lesquelles le *Calcul réseau* a pu être appliqué dans ce chapitre.

Pour conclure, nous avons dans cette partie présentée la méthode qui va nous permettre d'établir des temps de traversée. Cette théorie nous permet d'obtenir, à partir du *dioïde* des fonctions *(min, +)*, des expressions algébriques. À partir de celles-ci, les outils et programme informatique permettent de calculer des valeurs concrètes. Ces valeurs concrètes sont ensuite utilisées dans l'étude des réseaux embarqués.

BOUILLARD, BOYER et LE CORRONC réunissent dans [BBLC18] de nombreuses définitions, notions et résultats autour de la théorie du *Calcul réseau*. Les auteurs y détaillent suffisamment de résultats pour pouvoir être appliqué dans des cas réels. Ainsi, cet ouvrage présentant le *Calcul réseau* déterministe servira de base à ce manuscrit pour les notions.

Chapitre 4

L'assistant de preuve Coq

L'objectif de ce chapitre est d'apporter à un lecteur novice en Coq une explication sur le fonctionnement de ce logiciel : afficher et écrire une définition, utiliser l'environnement de preuve pour rédiger une preuve... Le but de cette présentation est de montrer comment nous utilisons Coq pour formaliser les notions et propriétés du reste du manuscrit.

Tout d'abord, Coq est un logiciel libre. Il appartient à la classe des assistants de preuve ou *Interactive Theorem Provers* (ITP), introduite au cours de l'introduction de ce manuscrit. La particularité de Coq est qu'il est fondé sur le calcul des constructions. Chaque objet/fonction est associé à une preuve construite par déduction dans une logique intuitioniste. Cette logique, à la différence de la logique classique, n'admet pas par exemple la règle du tiers exclus (pour une proposition P, nous n'avons pas $P \vee \neg P$).

Comme tout logiciel, Coq dispose de plusieurs versions et vient avec un ensemble de librairies. Dans ce chapitre ainsi que dans le reste du manuscrit, le code Coq est affiché avec la version 8.13. Nous utilisons, pour les librairies disponible sur Opam :

```
Mathcomp             => version 1.12
Mathcomp Analysis    => version 0.3.9
Hierarchy Builder    => version 1.0.0
CoqEAL               => version 1.0.5
```

et pour la librairie disponible sur `https://github.com/math-comp/dioid` :

```
Mathcomp Dioid => version master
```

Pour installer les librairies, il faut exécuter les commandes suivantes :

- `opam repo add coq-released https://coq.inria.fr/opam/released`
- `opam repo add coq-extra-dev https://coq.inria.fr/opam/extra-dev`
- `opam install coq.8.13.1 coq-mathcomp-algebra.1.12.0`
  `coq-mathcomp-analysis.0.3.9 coq-hierarchy-builder.1.0.0`
  `coq-coqeal.1.0.5 coq-mathcomp-dioid.dev`

Dans ce chapitre, nous allons dans la Section 4.1 dérouler un exemple d'une preuve Coq. Pour comprendre la correspondance avec une preuve papier, nous donnerons d'abord le raisonnement mathématique papier ou LaTeX. Puis, dans la Section 4.2 nous présenterons les librairies utilisés dans le reste du manuscrit.

Enfin, pour présenter un script Coq, nous utiliserons systématiquement l'environnement suivant :

```
Script coq (* Commentaires du Code Coq *)
```

avec comme indiqué, des commentaires entre les balises `(* *)`. Nous présenterons les autres environnement un peu plus spécifiques au fur et à mesure de ce chapitre.

4.1 Présentation de Coq

Pour cette présentation de l'assistant, nous allons prendre des exemples simples et représentatif. Dans cette section, nous allons donc tout d'abord expliquer quelles sont les définitions que nous souhaitons formaliser, écrites en LaTeX. Puis, nous développerons en Coq, pas à pas, la formalisation en Coq de ces définitions, avec du `code` Coq.

Coq utilise le paradigme de programmation fonctionnel. Prenons deux fonctions `f` et `g` : la fonction `g` prend un entier en entrée et la fonction `f` prend deux paramètres : la sortie de `g` et un entier. L'appel de la fonction `f`, avec comme premier argument le résultat de `g` appliqué à 1 et comme second argument 2, se note :

```
f (g 1) 2
```

Ce qui correspond à la syntaxe suivante, plus commune dans de nombreux langages de programmation :

```
f(g(1), 2)
```

Remarque 4.1 (*Type des fonctions*) Nous dirons alors que `g` est de type $\mathbb{N} \to A$, où A est un Type quelconque et que `f` est de type $A \to \mathbb{N} \to B$. La notation utilisée en Coq est alors :

```
g : nat -> A
f : A -> nat -> B
```

où la notation `nat` représente $\mathbb{N}$.

La programmation Coq se fait avec une liste de commandes. Nous donnons ici un ensemble des commandes qui sont utiles à la lecture et la compréhension de ce manuscrit. D'autres commandes peuvent être trouvées dans la documentation.

Définitions LaTeX

Nous souhaitons construire en Coq l'addition sur l'ensemble $\overline{\mathbb{R}} \triangleq \mathbb{R} \cup \{+\infty, -\infty\}$ comme utilisé dans le chapitre précédent. L'addition est définie par :

Définition 4.1 (Addition dans $\overline{\mathbb{R}}$)

$$\forall x, y \in \overline{\mathbb{R}}, x + y = \begin{cases} x + y & \text{si } x, y \in \mathbb{R} \\ +\infty & \text{si } x = +\infty \text{ ou } y = +\infty \\ -\infty & \text{sinon.} \end{cases}$$

Remarque 4.2 (*Addition de $+\infty$ et $-\infty$*) L'addition de $+\infty$ et $-\infty$ est souvent indéfinie. Pour correspondre avec le dioïde *(min, +)*, nous avons $+\infty + -\infty = +\infty$. En effet, dans le *dioïde (min, +)*, l'élément neutre $+\infty$ de la loi additive min est absorbant pour la loi multiplicative + [BBLC18].

Nous allons prouver la commutativité de cet opérateur.

Lemme 4.1 (L'addition sur $\overline{\mathbb{R}}$ est commutative) Pour tout $a, b \in \overline{\mathbb{R}}$, nous avons : $a + b = b + a$.

DÉMONSTRATION Par étude de cas sur a et b : si a et b sont réels, alors $a + b = b + a$. Si $a = +\infty$ alors, quelque soit b, nous avons $a + b = +\infty$. Si enfin $a = -\infty$, alors, si $b = +\infty$, nous avons $a + b = +\infty$, sinon nous avons $a + b = -\infty$. □

Code Coq

Maintenant que le LaTeX est donné, nous pouvons écrire le script. Comme tout programme informatique, il faut premièrement introduire des librairies avec la commande `Require Import`.

```
From mathcomp Require Import all_ssreflect ssralg ssrnum.
From mathcomp.analysis Require Import ereal.
```

Chaque commande Coq doit terminer par un point. En premier, nous prenons tout le contenu de `ssreflect` : une extension de Coq. Puis nous prenons le contenu de `ssralg` et `ssrnum` : utile pour décrire des structures algébrique et manipuler des structures numériques respectivement. Une description de ces librairies est disponible dans l'ouvrage de MAHBOUBI et TASSI [MT21]. Il contient notamment une introduction détaillée à l'utilisation de `ssreflect`.

Ensuite, nous prenons le module `ereal`, pour *extended real* de la librairie `MathComp Analysis`. Cette librairie permet de faire de l'analyse classique : un article de AFFELDT, COHEN et ROUHLING [ACR18] explique sa construction et son fonctionnement. Le module `ereal` sert à étendre un type numérique avec $+\infty$ et $-\infty$.

Pour mieux utiliser une librairie, Coq intègre de nombreuse commandes pour afficher les informations d'un objet. L'une d'entre elle est `Locate`. Suivi d'une chaîne de caractère, Coq donnera les informations qu'il a pu trouver sur cette chaîne qu'il a trouvé dans sa base de donnée. Nous écrivons alors, après une recherche dans les explications du fichier `ereal.v`, la commande :

```
Locate "\bar".
```

La faculté d'un assistant de preuve est d'aider l'utilisateur à rédiger une preuve. Donc, dès l'exécution d'une commande, Coq renvoie un messages en réponse expliquant les erreurs ou renvoyant les informations demandées par l'utilisateur.

```
∗ Reponses ∗
Notation "'\bar' R" := (extended R) : type_scope (default interpretation)
```

Une `Notation` est une expression pour réécrire plus lisiblement et parfois plus facilement une expression. Ici, Coq nous indique ici qu'une `Notation` est crée, dans le module `ereal` de la librairie `MathComp Analysis`, pour associer un ensemble numérique `R` à la fonction `extended` avec la notation `\bar R`.

Nous avons donc une fonction `extended`. La commande `Print` fait partie des commandes d'affichages de Coq. Elle affiche la définition, complète d'un objet Coq. Donc, pour comprendre `extended`, nous écrivons :

```
Print extended.
```

Tout comme `Locate`, la commande `Print` renvoie un message :

```
∗ Reponses ∗
Inductive extended (R : Type) : Type := EFin : R → \bar R | EPInf : \bar R | ENInf : \bar R
```

Le mot clé `Inductive` indique que `extended` est définie par cas (un *type somme*). Il est suivi du nom de la définition, ici `extended`. Elle comprend un paramètre `R` de type `Type` (un type générique). Nous avons ensuite les trois cas de la définition séparés par des barres verticales :

- le premier est `EFin`, pour *Extended Finite* : une valeur de `R` injectée dans `\bar R`.
- `EPInf` et `ENInf` pour *Extended Positive* et *Negative Infinite* : respectivement $+\infty$ et $-\infty$

Ces trois objets sont des fonctions à part entière. Ils forment à eux trois, les constructeurs de la fonction `extended` : ainsi, lorsque nous voudrons faire une analyse par cas d'un `extended`, nous obtiendrons ces trois éléments de `extended`.

Coq est un langage fortement typé : il est donc intéressant de connaître le détail du type d'une fonction. Dans ce but, nous utilisons la commande `About` : elle renvoie toutes les informations sur un objet Coq. Il peut s'agir d'informations comme les arguments à passer à une fonction, le type en sortie de fonction ou la manière utilisée pour définir l'objet (`Inductive` pour `er`).

```
About EFin.
```

```
* Reponses *

EFin : ∀ [R : Type], R → \bar R

Arguments EFin [R]%type_scope _
Expands to: Constructor mathcomp.analysis.ereal.EFin
```

Coq nous renvoie les informations du type du destructeurs `EFin`. Le message nous dit : pour tout type `R`, pour un élément de type `R`, `EFin` renvoie un objet de type `\bar R`.

L'information présente ensuite sur la deuxième ligne concerne les arguments implicite de `EFin`. Le premier est implicite, le type `[R]`, nous n'avons pas besoin de le préciser à chaque appel. C'est le cas généralement pour une information redondante car elle est présente dans les autres arguments. Ici, le type `R` sera le type de l'élément, Coq placera alors tout seul l'argument `R`.

Remarque 4.3 (*Donner les arguments implicite*) Si nous voulons donner tout de mêmes les arguments implicite, il faut écrire `@` devant la fonction. Cela peut être utile quand un type est ambigu ou dans les cas où Coq n'arrive pas à reconnaître le type d'un objet.

Dans le module `ereal`, d'autres notations sont présentes. Elles sont :

```
Notation "+∞ " := (@EPInf _) : ereal_scope.
Notation "−∞ " := (@ENInf _) : ereal_scope.
Notation "x %:E" := (@EFin _ x) (at level 0) : ereal_scope.

Declare Scope ereal_scope.
Delimit Scope ereal_scope with E.

Local Open Scope ereal_scope.
```

Lors de l'appel d'une fonction, il est possible de faire deviner à Coq cet argument. Pour cela, il faut noter `_` à la place de l'argument. Ainsi, $+\infty$ et $-\infty$ sont les termes pour `EPInf` et `ENInf` respectivement, quels que soit le type.

Pour `EFin`, nous utilisons la notation `%:E`. Afin d'améliorer la lisibilité, la notation introduite ici oblige Coq à trouver seul le premier argument de `EFin`. Cette obligation est donnée avec le `@` comme expliqué dans la remarque 4.3. Coq pourra systématiquement inférer cet argument puisque qu'il s'agit du type de `x`.

La commande `Declare Scope` sert à définir un `Scope`. Un `Scope` permet de forcer l'interprétation en un certain sens. En effet, avec l'inférence de type, il est bien souvent inutile de spécifier tout les paramètres d'une fonction puisque Coq les trouve seul. Cependant, l'utilisateur veut parfois spécifier un type de paramètres bien précis (cf Exemple 4.1). Donc, pour éviter toutes ambiguïtés de types, un mécanisme est mis en place avec le `Scope` pour indiquer à Coq la notation précise. Par convention, un type est associé à une lettre et il faut encadrer avec des parenthèses et un `%` la notation à laquelle on souhaite appliquer le `Scope`.

Exemple 4.1. Exemple `Scope` sur `R` et sur `F`

Prenons par exemple deux valeur a et $b \in \mathbb{R}$ et f et $g \in \mathcal{F}$. Lorsque l'on note : $a + b$, cela revient à noter une addition sur deux réels alors que $f + g$ revient à noter une addition de deux fonctions.

En Coq, la différence avec les deux notations se montre avec le `Scope` : `R` pour les réels et `F` pour les fonctions. Ainsi,

Suite de l'exemple 4.1

```
Check (a + b)%R.
```

signifie qu'il faut interpréter la notation + suivant le Scope R et

```
Check (f + g)%F.
```

signifie qu'il faut interpréter la notation + suivant le Scope F.
Pour connaître dans quel Scope une notation peut intervenir, nous faisons Locate "x" qui renvoie, dans cet exemple uniquement :

```
* Reponses *

Notation "x + y" := (GRing.add x y) : ring_scope
  (default interpretation)
Notation "x + y" := (F_plus x y) : F_scope
```

La fonction GRing.add est la fonction d'addition que nous utilisons pour l'addition de réels. La fonction F_plus sera présentée dans le Chapitre 5.

Le Scope des réels étendus est associé à la lettre, ici E. Cela arrivera notamment dans ce manuscrit et ainsi nous pourrons spécifier précisément le Scope de tel ou tel terme.

Enfin, nous ouvrons le Scope avec Local Open Scope pour ne pas avoir à préciser %E à chaque appel de Notation du Scope ereal_scope. Concrètement, cela revient à écrire $+\infty$ et au lieu de $+\infty$ %E.

Utilisons maintenant ssrnum pour supposer l'ensemble $\mathbb{R}$. Nous utilisons :

```
Section Presentation.

Context {R : numDomainType}.
```

Nous donnons d'abord une Section avant de décrire le reste de cette présentation. Ainsi, pour avoir une variable x dans l'ensemble $\overline{\mathbb{R}}$, il nous suffit de faire x : \bar R. Nous utilisons cette notation pour définir l'addition comme dans la Définition 4.1.

```
Definition adde (x y : \bar R) := match x, y with
  | x%:E , y%:E  => (x + y)%R%:E
  | +∞ , _ => +∞
  | _ ,  +∞ => +∞
  | _ ,  _ => −∞
 end.
```

Le mot clé Definition est utilisé pour définir une fonction basique et fonctionne comme Inductive : le nom est ici adde. Les deux paramètres sont x et y de type \bar R, donc $\overline{\mathbb{R}}$. La définition utilise ensuite un match pour décrire tous les cas de x et y simultanément. Coq permet cette étude de cas car \bar R est un type inductif. Il n'est pas possible d'oublier un cas en Coq.

Dans un match, les différents cas n'ont pas la même priorité. Le premier cas est prioritaire, cela signifie que la condition du premier cas sera vérifiée avant toutes les autres. Ensuite, si la première condition n'est pas remplie, la seconde condition sera vérifiée et ainsi jusqu'au dernier cas, qui est alors un cas par défaut.

Le premier cas de x et y est le cas des valeurs réelles : x%:E et y%:E sont les valeurs de x et y sur R respectivement. Dans ce cas là, la sortie de la fonction est alors la somme de x et y sur $\mathbb{R}$ mais ramené sur $\overline{\mathbb{R}}$, d'où le Scope %R autour de l'addition puis le Scope %E après.

Les deux cas suivants sont les cas où soit x soit y sont $+\infty$. Dans ces cas là, le résultat ne dépend

pas de la valeur de `y` ou `x` respectivement. C'est pour cela que nous notons `_`. Notons tout de même que dans le cas décrit par :

```
| _     ,   +∞ ⇒ +∞
```

la valeur de `x` ne peut pas être un réel ou $+\infty$, car ces cas ont déjà été traités dans les deux cas précédent. Ainsi, dans le dernier cas, `x` ou `y` sont forcement $-\infty$.

> **Remarque 4.4** (*Suite de la Remarque 4.2*) La Définition 4.1 de l'addition est différente de celle présente dans toutes les librairies Coq sur les réels étendus. Dans la librairie Coquelicot développée par BOLDO, LELAY et MELQUIOND dans [BLM15], la somme de $+\infty$ et $-\infty$ donne 0. Cette définition permet d'obtenir des propriétés avec l'opposé mais cette addition n'est pas associative. Dans la librairie `MathComp Analysis`, l'addition de $+\infty$ et $-\infty$ donne $-\infty$ pour avoir une algèbre de dioïde avec max et + comme opérateur respectivement d'addition $\oplus$ et de multiplication $\otimes$.

Nous ajoutons maintenant des `Notation`.

```
Notation "+%E" := adde.
Notation "x + y" := (adde x y) : ereal_scope.
```

Coq permet aussi de calculer le résultat de fonctions. Sur la fonction `adde` définie au dessus, nous pouvons par exemple évaluer la valeur de $-\infty + +\infty$ avec la commande :

```
Eval compute in −∞ + +∞ .
```

```
* Reponses *

  = +∞
  : \bar R
```

Nous retrouvons bien le résultat décrit dans la Remarque 4.2. Calculer des valeurs permettra de simplifier les cas simples dans l'écriture des preuves.

Maintenant que la définition est donnée, nous pouvons prouver le Lemme 4.1. Pour écrire ce Lemme, nous allons faire appel à une fonction présente dans `ssreflect` : `commutative` :

```
Print commutative.
```

```
* Reponses *

commutative =
fun (S T : Type) (op : S → S → T) ⇒ ∀ x y : S, op x y = op y x
     : ∀ S T : Type, (S → S → T) → Prop
```

Donc, cette fonction prend en entrée deux types `S` et `T` et un opérateur `op` de type `S → S → T`. La propriété renvoyée par la fonction est bien la propriété de commutativité de `op`.

Pour mieux comprendre, il peut être utile d'utiliser la commande `Check`, qui permet de demander le type d'une expression.

```
Check commutative +%E.
```

Nous avons ici appliqué `commutative` à notre fonction d'addition `+%E`. La réponse obtenu est :

∗ Reponses ∗

```
commutative +%E
  : Prop
```

Notons ici que les argument `S` et `T` sont implicites. Coq les devine grâce au type + `%E` : `S` et `T` sont de Type `\bar R` puisque `adde` : `\bar R` → `\bar R` → `\bar R`.

Nous pouvons alors commencer à énoncer le Lemme avec la commande `Lemma`. Cette commande attend une `Prop` après les :. Le nom donné ci dessous est `addeC`.

∗ Goals ∗

```
Lemma addeC : commutative +%E.
Proof.
```

```
R : numDomainType
______________________________
commutative +%E
```

La preuve sera décrite à gauche, après le mot clé `Proof`. Sur la droite, nous affichons le `Goal` donné par Coq. Celui ci est interactif et évoluera au fur et à mesure que la preuve évoluera.

Nous faisons appel à des tactiques pour résoudre le `Goal`. La première d'entre elles est `move=>` qui permet d'introduire les variables de la preuve, c'est à dire les éléments a et b.

∗ Goals ∗

```
Lemma addeC : commutative +%E.
Proof.
move=> a b.
```

```
R : numDomainType
a, b : \bar R
______________________________
a + b = b + a
```

Une fois introduite, les variables apparaissent en haut du but à prouver. Le type de `a` et `b` est explicité à cet endroit et nous retrouvons bien le type Inductif `\bar R` défini plus tôt.

En reprenant la preuve LaTeXdu Lemme 4.1, nous faisons une étude de cas sur les variables `a` puis `b`. Cette étude se fait avec la tactique `case`: suivie du nom de la variable. Pour appliquer la seconde étude de cas à chaque cas de la première étude, nous allons placer un ; après `case: a`. Ainsi, le `case: b` sera appliqué à chaque sous cas découlant de `case: a`.

∗ Goals ∗

```
Lemma addeC : commutative +%E.
Proof.
move=> a b.
case: a; case : b.
```

```
R : numDomainType
______________________________
∀ r r0 : R, r0%:E + r%:E = r%:E + r0%:E

subgoal 2 (ID 6314) is:
 ∀ r : R, r%:E + +∞ = +∞ + r%:E
subgoal 3 (ID 6315) is:
 ∀ r : R, r%:E + −∞ = −∞ + r%:E
subgoal 4 (ID 6326) is:
 ∀ r : R, +∞ + r%:E = r%:E + +∞
```

∗ Goals ∗

```
subgoal 5 (ID 6327) is:
 +∞ + +∞ = +∞ + +∞
subgoal 6 (ID 6328) is:
 +∞ + −∞ = −∞ + +∞
subgoal 7 (ID 6339) is:
 ∀ r : R, −∞ + r%:E = r%:E + −∞
subgoal 8 (ID 6340) is:
 −∞ + +∞ = +∞ + −∞
subgoal 9 (ID 6341) is:
 −∞ + −∞ = −∞ + −∞
```

Coq donne les 9 cas qu'il reste à prouver (3 possibilités pour `a` et pareillement pour `b`). L'intérêt ici est de demander à Coq de calculer chaque cas trivial pour réduire le nombre de buts à prouver.

Nous ajoutons ainsi à la commande précédente `//=` qui est une tactique `ssreflect` pour calculer puis résoudre les cas triviaux. Cela donne :

∗ Goals ∗

```
Lemma addeC : commutative +%E.
Proof.
move⇒ a b.
case: a; case : b ⇒ //=.
```

```
R : numDomainType
______________________________
∀ r r0 : R, (r0 + r)%R%:E
                    = (r + r0)%R%:E
```

Nous nous retrouvons avec le cas uniquement réel, les autres ont été résolus automatiquement par Coq. Pour terminer la preuve, il reste alors à prouver ce dernier but.

Pour cela, il nous faut introduire `r` et `r0` en sachant cette fois ci que les valeurs sont dans $\mathbb{R}$. Nous leurs donnerons le nom `a` et `b` respectivement. Puis nous pourrons réécrire le lemme correspondant à la commutativité de l'addition de deux nombres réels.

Pour utiliser un tel lemme, il faut généralement connaître son nom. Pour éviter de rechercher dans la librairie, lemme après lemme, la propriété qui nous intéresse, Coq peut lui même rechercher dans les librairies chargées au milieu d'une preuve. Ainsi, en tapant :

```
Search commutative +%R.
```

∗ Reponses ∗

```
addrC : ∀ V : zmodType, commutative +%R
```

Nous allons réécrire ce Lemme avec la commande `rewrite`. Cette commande peut être utilisée avec plusieurs lemmes, donnant ainsi une suite de réécritures du but.

∗ Goals ∗

```
Lemma addeC : commutative +%E.
Proof.
move⇒ a b.
case: a; case : b ⇒ //=a b.
rewrite addrC.
```

```
R : numDomainType
b, a : R
______________________________
 (b + a)%R%:E = (b + a)%R%:E
```

Le but obtenu est trivial : une égalité symétrique. Pour terminer la preuve, nous pouvons utiliser la commande `by` qui doit se placer avant la dernière commande.

Cependant, nous allons d'abord simplifier la preuve que nous avons écrite. Il peut exister plusieurs manières d'écrire une preuve Coq. La façon décrite depuis le début de cette preuve est assez longue et

verbeuse. Un développeur en Coq expérimenté aura tendance à comprimer la preuve en faisant ressortir les étapes principales de la preuve.

Par exemple, nous avons fait `move=>` suivi de `case`: avec la même variable. En `ssreflect`, nous pouvons utiliser `move=> []` pour directement faire l'analyse de cas. De plus, il sera possible d'introduire des variables avec cette commande en différenciant les cas avec des barres verticales. Dans le cas d'une variables de type `\bar R`, nous devons introduire une variable dans le premier cas mais pas dans le deuxième et le troisième. Alors, pour introduire `a` dans uniquement le premier cas, nous faisons : `move=> [a| ||]`, et idem pour `b`.

```
* Goals *
Lemma addeC : commutative +%E.
Proof. by move=> [a| ||] [b| ||] //=; rewrite addrC. Qed.
```

La commande `Qed` permet de conclure la preuve. Nous concluons aussi la section avec :

```
End Presentation.
```

4.2 Librairies

Avant de citer les librairies, nous allons expliquer un détail syntaxique largement utilisé dans ce manuscrit. Lors de l'importation des librairies, Coq charge des modules. Dans l'idée, cela correspond à un fichier Coq en particulier. Un module peut contenir d'autres modules. Pour y accéder, il faut utiliser un `.` comme dans une programmation objet. Par exemple, si nous avons un module `moduleA` contenant `moduleDansA`, accéder à ce second module se ferait avec `moduleA.moduleDansA`.

Nous avons commencé par citer des librairies pour décrire l'exemple dans la section précédente. Nous allons ici détailler un peu plus en détail les librairies utiles au reste du manuscrit.

Pour commencer, nous avons cité le livre explicatif de `MathComp` [MT21] au début de ce Chapitre. Les preuves de ce manuscrit sont rédigées en grande partie à l'aide de tactique de `ssreflect` car elles sont plus facile à utiliser que les tactiques Coq basiques.

Exemple 4.2. Module dans `ssreflect`

Pour reprendre les propos énnoncé en début de cette section, nous utiliserons souvent une fonction `max` déclaré dans le fichier `order`. Celui-ci contient un module `Order` contenant la définition de `max` donc pour récupérer cette défini ton (et généralement les propriétés développées autour), il nous faut écrire `Order.max`.

Par la suite, les même auteurs ont donné une introduction aux Structures Canoniques en Coq [MT13]. Coq utilise l'inférence de type pour comparer les types avec sa base de donnée. Les structures canoniques servent à étendre cette base de donnée. Concrètement, cela permet à Coq de reconnaître et créer des liens dans les types ajoutés par un utilisateur.

Dans l'exemple précédent, les `Canonical Structure` permettent à Coq de comprendre que `\bar R` est bien un type possédant un opérateur d'égalité et disposant d'un ordre.

Avec les structures canoniques, `ssrflect` permet d'utiliser des *big* opérations qui permettent de généraliser des opérations binaires sur des opérations d'ensembles. BERTOT et al. donnent en [Ber+08] une explication sur l'utilisation. Pour un opérateur, afin d'obtenir de nombreuses propriétés si l'on souhaite effectuer cette opération sur un ensemble, il faut déclarer une structure canonique de *monoid* commutatif (cf. Définition 5.1). Cela revient à montrer que l'opérateur est associatif, avec un élément neutre et commutatif.

Pour la construction de structures algébriques, nous utiliserons des librairies sur les structures algébriques de `MathComp`. L'une d'entre elle est celle exposée dans l'exemple ci dessus, `ssralg`. Une librairie plus récente est sortie au cours de la réalisation de cette thèse, la librairie `Hierarchy Builder`

développée par COHEN, SAKAGUCHI et TASSI [CST20]. Elle permet de créer des structures algébriques directement à partir de la hiérarchie de celle ci et de s'abstraire des déclarations précise de structures canoniques.

Exemple 4.3. Exemple de hierarchy Builder

Nous souhaitons par exemple construire la structure algébrique de *semi-groupe* correspondant à la définition 2.1.1 de MINOUX et GONDRAN [MG08] : un *semi-groupe* est un ensemble E muni d'une loi binaire associative. Nous avons comme pour la propriété de commutativité, une fonction de `MathComp` qui permet de décrire l'associativité d'une loi :

```
* Reponses *

associative =
    fun (S : Type) (op : S → S → S) ⇒ ∀ x y z : S, op x (op y z) = op (op x y) z
```

La fonction `associative` prend un type `S` et une loi binaire `op : S → S → S` et renvoie la propriété que la loi est associative. Comme pour `commutative`, le paramètre `S` sera inféré automatiquement avec le paramètre `op`. Pour décrire un *semi-groupe*, nous faisons :

```
HB.mixin Record semi_groupe_of_type R := {
    add : R → R → R;          (* Une loi *)
    addA : associative add; (* La propriete que cette loi est associative *)
  }.
```

La commande `HB.mixin` est ajouté par la librairie `Hierarchy Builder` [CST20]. Elle permet de créer un constructeur d'instance nommé `mixin` pour la structure de `semi-groupe`. La commande `Record` agit comme la commande `Definition` avec la différence que `Record` est une description éléments par éléments de l'objet défini.
Pour créer la structure en elle-même, nous utilisons la commande `HB.instance`, qui prend le `mixin` précédemment définie est donne la structure de `semi-groupe` :

```
HB.structure Definition SemiGroupe := { R of semi_groupe_of_type R }.
```

Nous pouvons réutiliser cette structure pour, par exemple, déclarer un *semi-groupe commutatif*, avec l'instruction suivante :

```
HB.mixin Record ComSemiGroupe_ofSemigroupe R of SemiGroupe R := {
  muldC : ∀ a b : R, add a b = add b a ;
}.
```

Dans cette commande, nous supposons un Type `R` muni d'un `SemiGroupe`. Alors, un *semi-groupe commutatif* est un `SemiGroupe` avec la propriété `muldC` : propriété de commutativité que nous avons explicité (il aurait été possible d'utiliser `commutative`).

En introduction à ce chapitre, nous avons listé aussi la librairie `MathComp Dioid`. Cette librairie fait partie des contributions de ce manuscrit. Elle sera détaillée dans le Chapitre 5.

Nous aurons besoin dans ce manuscrit de construire des programmes Coq pour vérifier des valeurs concrètes. Pour cela, nous allons devoir raffiner les propriétés prouvées : produire un programme de calcul à partir de spécifications. Une librairie effectue déjà cette transition, `CoqEAL`. Dans la partie 3 de la thèse de ROUHLING [Rou19], une utilisation de cette librairie est largement décrite.

Chapitre 5

Formalisation en Coq du *dioïde* des fonction *(min, +)*

Sommaire

5.1 Théorie des *dioïdes* 30
5.1.1 *Semi-anneau*, *dioïde* et relation d'ordre 30
5.1.2 Un *dioïde* complet à partir d'un treillis complet 37
5.1.2.1 Treillis complet 37
5.1.2.2 *Dioïde* complet 40
5.1.3 Théorie pour l'étoile de Kleene 42
5.1.4 Théorie de la résiduation 44
5.1.5 Sous-structures algébriques 45
5.2 Le *dioïde* des fonctions *(min, +)* 46
5.2.1 L'instance $(\overline{\mathbb{R}}, \min, +)$ 46
5.2.2 Instance d'un sous ensemble de $\overline{\mathbb{R}}$ 48
5.2.3 Instances $(\mathcal{F}, \min, *)$: le *dioïde* des fonctions *(min, +)* 48
5.2.4 Le sous ensemble des fonctions positives est un dioïde complet 51
5.2.5 Le sous ensemble des fonctions croissantes est un dioïde complet 52
5.2.6 Propriétés et lien dans les différentes instances de *dioïde complet* 54
5.3 Conclusion 55

L'objectif de ce chapitre est de formaliser la structure mathématiques utilisée dans le *Calcul réseau*. Pour cela, nous allons devoir développer les définitions et exprimer les lemmes nécessaires à cette construction. Ce niveau de détail est impliqué par l'utilisation d'un assistant de preuve.

Cette formalisation va utiliser l'assistant de preuve Coq. Nous avons déjà introduit cet outil au Chapitre 4 et aucune autre connaissances en Coq n'est nécessaire pour comprendre ce chapitre. Notre développement se base sur la librairie `MathComp` [MT21] et la librairie plus récente `Hierarchy Builder` [CST20].

La structure algébrique que nous allons formaliser est le *dioïde* des fonctions *(min, +)*. Pour cela, nous allons d'abord formaliser la base nécessaire mathématique : la structure algébrique des *dioïdes* et *dioïde complet*. Il s'agit de structures déjà présentées par Minoux et Gondran dans [MG08] et repris par Bouillard, Boyer et Le Corronc [BBLC18].

Une fois cette base mathématiques développée, nous associerons des instances à cette structure. Cette association a pour but de faire profiter à l'instance de l'ensemble des propriétés mathématiques de la structure des *dioïdes*. De plus, le *dioïde* des fonctions *(min, +)* est une instance et ainsi nous aurons créé la base mathématiques du *Calcul réseau*.

Comme nous l'avons dit précédemment, l'ensemble de la formalisation a été produit en Coq. Afin d'améliorer la lisibilité, nous ne montrerons pas les preuves ou seulement une partie d'entre elles. La formalisation se trouve dans les fichiers Coq que nous rappelons en chaque début de section et sous section. Chaque définition, lemmes ou proposition est alors le nom d'une fonction Coq que affichons avec ce `style`.

La première section concerne la formalisation des structures algébriques de *dioïde* et *dioïde complet*. Dans celle ci, nous donnons l'ensemble des propriétés et opérations apportées par la structure. Dans la seconde section, nous faisons l'association avec plusieurs instances et nous donnons la traduction des opérations apportées par les structures algébriques.

Nous allons dans ce chapitre décrire des notions et définitions qui ont été formalisé en Coq. Ainsi, pour chaque propriété mathématique, nous donnerons au moins le nom Coq de l'élément dans lequel

cette propriété est formalisée. Quand le nom de l'élément en Coq n'est pas mentionné, alors il s'agit d'un élément qui ne peut pas être décris avec un seul élément. Les éléments Coq que nous décrirons sont compris dans des fichiers Coq eux mêmes compris dans des répertoires. Dans ce chapitre, nous aurons besoin de deux répertoires. Nous mentionnerons le répertoire à chaque début de partie ainsi qu'un lien vers lequel le code peut être regardé.

5.1 Théorie des *dioïdes*

Le *Calcul réseau* utilise une structure algébrique : le *dioïde* des fonctions *min,+*. En se basant sur la définition donnée par Minoux et Gondran [MG08], l'ouvrage de Bouillard, Boyer et Le Corronc [BBLC18] définissent cette la structure algébrique des *dioïdes*. Cet ouvrage formalise sur papier et développe de nombreuse définitions et propriétés mathématiques de la structure algébrique de dioïde.

Nous choisissons la formalisation de ce dernier ouvrage.

L'ensemble du dévelopement est disponible sur `github`. En effet, la branche de cette contribution a pu être ajoutée à une partie du projet `MathComp`. Il est disponible à l'adresse :

`https://github.com/math-comp/dioid`.

Chaque fichier de cette section, avec l'extension `.v`, est donc disponible dans ce répertoire.

5.1.1 *Semi-anneau*, *dioïde* et relation d'ordre

Nous allons dans cette partie décrire les éléments formalisés en Coq dans le fichier `dioid.v`. Nous avons ici importé les librairies citées en section 4.2 hormis la librairie sur les *dioïdes* puisque nous en faisons une description ici.

Tout d'abord, un *dioïde* est une structure algébrique. Donc il se compose d'un ensemble, de lois et d'éléments respectant des règles. Dans le cas d'un *dioïde*, sa structure se base sur celle d'un *semi-anneau*. Pour définir un *semi-anneau*, nous avons tout d'abord besoin de la structure de *monoïde*. Nous rappelons ici sa définition.

Définition 5.1 (Monoïde) Soit $\mathcal{D}$ un ensemble, $\oplus : \mathcal{D} \times \mathcal{D} \to D$ un opérateur et $\bar{0} \in \mathcal{D}$ un élément neutre. Nous appelons $(\mathcal{D}, \oplus)$ un *monoïde* si :

— $\oplus$ est associative, c'est à dire : $\forall a, b, c \in \mathcal{D}, (a \oplus b) \oplus c = a \oplus (b \oplus c)$,

— et $\bar{0}$ est un élément neutre de $\oplus$, c'est à dire : $\forall a \in \mathcal{D}, a \oplus \bar{0} = \bar{0} \oplus a = a$.

Remarque 5.1 (*Notation pour l'élément neutre*) Il est souvent utilisé ϵ dans la littérature (dans [BBLC18 ; MG08] par exemple) pour décrire l'élément neutre de l'addition. Nous préférons utiliser $\bar{0}$ qui est plus proche de la notation usuelle du 0.

Une telle structure se retrouve dans la théorie des langages, avec un ensemble de mots comme ensemble, la loi de concaténation de mots et le mot vide comme élément neutre. Si la loi est commutative, nous avons alors une autre structure algébrique définie ci-dessous.

Définition 5.2 (Monoïde commutatif) Soit $(\mathcal{D}, \oplus)$ un *monoïde*. Nous appelons $(\mathcal{D}, \oplus)$ un *monoïde commutatif* si $\oplus$ est commutatif, c'est à dire : $\forall a, b \in \mathcal{D}, a \oplus b = b \oplus a$.

Exemple 5.1. Monoïde commutatif $(\overline{\mathbb{R}}, \max)$

Nous pouvons associer l'ensemble $\overline{\mathbb{R}}$ et la loi max à la structure algébrique de *monoïde commutatif*. Il est clair que la loi max est associative, a pour élément neutre $-\infty$ et est commutative. Nous disons alors que $(\overline{\mathbb{R}}, \max)$ est une instance de *monoïde commutatif*.

Les structures algébriques des Définitions 5.1 et 5.2 n'ont pas nécessité une nouvelle formalisation en Coq. Ces structures algébriques existent déjà dans `MathComp` avec la formalisation donnée par Bertot et al. [Ber+08], dans le fichier `bigop` sous le nom `Monoid.law` et `Monoid.com_law`. En utilisant ces deux structures algébriques, nous pouvons définir un *semi-anneau*.

Définition 5.3 (Semi-anneau) Un ensemble $\mathcal{D}$ avec deux lois binaire $\oplus$ et $\otimes : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ et deux éléments $\bar{0}$ et $\bar{1} \in \mathcal{D}$ est un *semi-anneau* si :

— $(\mathcal{D}, \oplus)$ est un *monoïde commutatif* avec $\bar{0}$ comme élément neutre

— $(\mathcal{D}, \otimes)$ est un *monoïde* avec $\bar{1}$ comme élément neutre,

— $\otimes$ est distributive par rapport à $\oplus$, c'est à dire que pour tout a, b et $c \in \mathcal{D}$, nous avons :

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \quad \text{et} \quad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$$

— $\bar{0}$ est absorbant pour $\otimes$, c'est à dire : $\forall a \in \mathcal{D}, a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$.

Nous disons alors que $(\mathcal{D}, \oplus, \otimes)$ est un *semi-anneau* avec $\bar{1}$ pour élément unitaire et $\bar{0}$ pour élément zéro.

Remarque 5.2 (*Notation pour* $\otimes$) Dans les preuves, nous noterons parfois ab pour $a \otimes b$.

Nous donnons dans l'exemple suivant une instance de semi-anneau avec les mêmes opérations que dans l'exemple 5.1. Nous y décrivons une opération définie dans un Chapitre précédent, l'addition de valeurs dans $\overline{\mathbb{R}}$, avec une différence étant donné l'instance qui est présentée dans cet exemple. [1]

Exemple 5.2. Semi-anneau $(\overline{\mathbb{R}}, \max, +)$

Avec l'exemple 5.1, nous savons que $(\overline{\mathbb{R}}, \max)$ est déjà un *monoïde commutatif*. De la même manière, nous savons que $(\overline{\mathbb{R}}, +)$ est un *monoïde* avec pour élément neutre 0. Nous avons, par ailleurs, pour tout a, b et $c \in \overline{\mathbb{R}}$, $a + \max(b, c) = \max(a + b, a + c)$ et $-\infty$ absorbant pour l'addition en définissant l'addition sur $\overline{\mathbb{R}}$ telle que $+\infty + -\infty = -\infty$ [1]. Donc, $(\overline{\mathbb{R}}, \max, +)$ est un *semi-anneau*.

L'objectif des prochains paragraphes est de formaliser en Coq la structure de *semi-anneau*. En plus des fonctions présentées dans le Chapitre 4 (`commutative` à la page 24 et `associative` à la page 28), nous allons utiliser les fonctions `left_id` et `right_id` qui renvoient la propriété qu'un élément est neutre par rapport à une opération, à gauche et à droite respectivement. En Coq, nous avons après un `Print` des deux fonctions :

```
* Reponses *

left_id = fun (S T : Type) (e : S) (op : S → T → T) ⇒ ∀ x : T, op e x = x
right_id = fun (S T : Type) (e : S) (op : T → S → T) ⇒ ∀ x : T, op x e = x
```

La fonction `left_distributive` est une fonction donnant la propriété qu'un premier opérateur est distributif à gauche par rapport à un second opérateur :

```
* Reponses *

left_distributive = fun (S T : Type) (op : S → T → S) (add : S → S → S) ⇒
                          ∀ (x y : S) (z : T), op (add x y) z = add (op x z) (op y z)
```

1. Le résultat est différent de celui montré en Définition 4.1 car dans celle ci, nous étions dans le semi-anneau $(\overline{\mathbb{R}}, \min, +)$.

Similairement, `right_distributive` donne comme propriété : `op x (add y z) = add (op x y) (op x z)`. Enfin, les fonctions `left_zero` et `right_zero` renvoient la propriété d'absorption par un élément, à gauche et à droite respectivement. Pour `left_zero`, nous avons :

```
* Reponses *

left_zero = fun (S T : Type) (z : S) (op : S → T → S) ⇒ ∀ x : T, op z x = z
```

La formalisation en Coq d'un *semi-anneau* a été faite grâce aux librairies `Hierarchy Builder` [CST20] et `MathComp`. Un exemple de description d'une structure algébrique a déjà été donné en Section 4.2. La première étape consiste à lister les propriétés spécifiques d'un *semi-anneau* :

```
HB.mixin Record SemiRing_of_WrapChoice R of WrapChoice R := {
  zero : R;         (* Élément neutre *) one : R;          (* Élément neutre *)
  add : R → R → R; (* Loi additive *)    mul : R → R → R; (* Loi multiplicative *)
```

Nous supposons ici un ensemble `R` qui est un `WrapChoice`. Il s'agit d'une structure créée dans un fichier que nous avons ajouté dans notre développement :`HB_wrappers.v`. En effet, nous verrons au fur et à mesure de ce chapitre que nous aurons besoin de structures Coq de liaison avec des librairies existantes. Ici, `WrapChoice` permet d'avoir une connexion aux structures `eqType` et `choiceType`, utilisés respectivement pour avoir une égalité et pour avoir une fonction de choix dans la structure. Nous décrivons ensuite les éléments `zero`, `one`, `add` et `mul` correspondant respectivement à $\bar{0}$, $\bar{1}$, $\oplus$ et $\otimes$.

Dans le même `Record`, nous donnons ensuite les propriétés du *semi-anneau*. Les premières sont :

```
  (* Propriétés sur la loi additive (monoide commutatif) *)
  adddA : associative add;   adddC : commutative add;    add0d : left_id zero add;
  (* Propriétés sur la loi multiplicative (monoide) *)
  muldA : associative mul;   mul1d : left_id one mul;    muld1 : right_id one mul;
```

Plutôt que de supposer un *monoïde*, nous donnons directement les propriétés sur $\oplus$ et $\otimes$. A noter qu'il n'est pas nécessaire de donner la propriété de `right_id` pour la loi additive : nous pouvons la retrouver avec `addC` et `add0d` (cf. Lemme 5.1).

Enfin, nous décrivons les dernières propriétés de distributivité et d'absorption de la Définition 5.3 avec :

```
  (* la loi multiplicative est distributive sur la loi additive *)
  muldDl : left_distributive mul add;        muldDr : right_distributive mul add;
  (* zero est absorbant pour la loi multiplicative *)
  mul0d : left_zero zero mul;                muld0 : right_zero zero mul;
}.
```

qui concluent la définition. Pour créer une instance de *semi-anneau*, il faut être capable de donner les éléments `zero`, `one`, `add` et `mul` associés aux propriétés `adddA`, `adddC`, `add0d`, `muldA` ...

Nous définissons la structure de semi anneau à l'aide de `HB.structure` comme la structure contenant `WrapChoice` et le mixin `SemiRing_of_WrapChoice` qu'on vient de définir.

```
HB.structure Definition SemiRing := { R of WrapChoice R & SemiRing_of_WrapChoice R }.
```

Donc un `SemiRing` est un ensemble `R` muni des propriétés de `WrapChoice` et `SemiRing_of_WrapChoice`.

Comme il a été mentionné plus haut, nous pouvons retrouver la propriété d'absorption à droite par $\oplus$.

Lemme 5.1 (`addd0`) Soit $(\mathcal{D}, \oplus, \otimes)$ un *semi-anneau*, pour tout $x \in \mathcal{D}$ nous avons $x \oplus \bar{0} = x$.

En Coq, la formalisation de ce lemme s'écrit :

```
Variables R : SemiRing.type.

Lemma addd0 : right_id (@zero R) add.
```

Cette propriété pourra être utilisée sous tout objet ou instance qui est un *semi-anneau*. Nous montrons une utilisation dans l'Exemple 5.4.

La prochaine définition est une extension de la structure algébrique de *semi-anneau*.

Définition 5.4 (Semi-anneau commutatif) Un *semi-anneau* $(\mathcal{D}, \oplus, \otimes)$ est un *semi-anneau commutatif* si la seconde loi $\otimes$ est commutative.

Pour la formalisation, nous pouvons reprendre la structure de `SemiRing` dans le mixage de propriété. Ainsi, nous n'avons pas à donner les propriétés de *semi-anneau*.

```
HB.mixin Record ComSemiRing_of_SemiRing R of SemiRing R := {
  muldC : @commutative R _ mul;
}.
```

Nous supposons donc un `SemiRing` avec l'ensemble `R`. Le mixeur ajoute ensuite la propriété `muldC` pour la commutativité. Ici, il est nécessaire de donner le premier argument de `commutative` car l'opération `mul` n'est pas spécifiée sur un type en particulier, elle est juste une opération pour un `SemiRing` mais aucun type n'y est associé. Coq ne peut pas inférer le paramètre `R` : nous utilisons la syntaxe `@` pour donner explicitement tous les paramètres. Le deuxième argument est quand à lui inféré automatiquement.

Bien évidement, le Lemme 5.1 que nous avons montré sur le `SemiRing` est utilisable sur un *semi-anneau commutatif*. Nous avons ensuite la déclaration de la structure :

```
HB.structure Definition ComSemiRing := { R of SemiRing R & ComSemiRing_of_SemiRing R }.
```

Puis nous pouvons donner de nouvelles propriétés. Celles-ci seront utilisables sur toutes les instances de `ComSemiRing`.

Lemme 5.2 (`muldAC, muldCA, muldACA`) Soit $(\mathcal{D}, \oplus, \otimes)$ un *semi-anneau commutatif*. Nous avons, pour tout x, y et $z \in \mathcal{D}$:

$$(x \otimes y) \otimes z = (x \otimes z) \otimes y$$
$$x \otimes (y \otimes z) = y \otimes (x \otimes z)$$
$$(x \otimes y) \otimes (z \otimes t) = (x \otimes z) \otimes (y \otimes t)$$

Nous utilisons les fonctions `right_commutative`, `left_commutative` et `interchange` de `MathComp` pour décrire le Lemme 5.2 : elles sont exactement la définition des équations de ce lemme. Par exemple, pour `right_commutative` :

```
* Reponses *

right_commutative =
fun (S T : Type) (op : S → T → S) ⇒ ∀ (x : S) (y z : T), op (op x y) z = op (op x z) y
```

La prochaine définition utilise un *semi-anneau* pour définir la structure algébrique d'un *dioïde*.

Définition 5.5 (Dioïde) Un *semi-anneau* $(\mathcal{D}, \oplus, \otimes)$ est un *dioïde* si la première loi $\oplus$ est idempotente, c'est à dire si pour tout $a \in \mathcal{D}$ la relation $a \oplus a = a$ est respectée.

Avec un *dioïde*, nous pouvons avoir un *dioïde commutatif*.

Définition 5.6 (Dioïde commutatif) Un *dioïde* $(\mathcal{D}, \oplus, \otimes)$ est un *dioïde commutatif* si la loi $\otimes$ est commutative.

Définition 5.7 (Relation d'ordre dans le *dioïde*) Nous pouvons associer à tout *dioïde* $(\mathcal{D}, \oplus, \otimes)$ la relation d'ordre $\preccurlyeq$ définie par, pour a et $b \in \mathcal{D}$,

$$a \preccurlyeq b \overset{\Delta}{\iff} a \oplus b = b.$$

Démonstration (Définition 5.7) Montrons que $\preccurlyeq$ est une relation d'ordre, c'est à dire qu'elle est :

— réflexive, c'est à dire : $a \preccurlyeq a$,

— antisymétrique, c'est à dire que si $a \preccurlyeq b$ et $b \preccurlyeq a$ alors $a = b$,

— transitive, c'est à dire si $a \preccurlyeq b$ et $b \preccurlyeq c$ alors $a \preccurlyeq c$.

La relation $\preccurlyeq$ est réflexive parce que la loi $\oplus$ est idempotente.

Si pour a et b nous avons $a \preccurlyeq b$ et $b \preccurlyeq a$, alors $a \oplus b = b$ et $b \oplus a = a$ donc $a \oplus b = a$ par commutativité et $b = a$.

Enfin, Si pour a et b nous avons $a \preccurlyeq b$ et $b \preccurlyeq c$, alors $a \oplus b = b$ et $b \oplus c = c$. Donc, $a \oplus c = a \oplus (b \oplus c)$ et, par associativité, nous avons $a \oplus c = (a \oplus b) \oplus c = b \oplus c = c$. □

Exemple 5.3. Dioïde $(\overline{\mathbb{R}}, \max, +)$

En poursuivant l'exemple 5.2, $(\overline{\mathbb{R}}, \max, +)$ est un *dioïde* car la loi max est bien idenpotente. C'est aussi un *dioïde commutatif*. La relation d'ordre $\preccurlyeq$ est pour ce *dioïde* la relation $\leqslant$ usuelle sur $\overline{\mathbb{R}}$.

Pour la formalisation d'un *dioïde*, prenons premièrement la fonction `idempotent` de `MathComp` qui retourne la propriété d'idempotence rappelé à la Définition 5.5 :

∗ Reponses ∗

```
idempotent = fun (S : Type) (op : S → S → S) ⇒ ∀ x : S, op x x = x
```

Nous ajoutons ensuite une connexion aux structures d'ordre de `MathComp` : nous étendons `WrapChoice` avec la structure d'ordre de `MathComp` formalisée dans `order`. Nous définissons ainsi la structure `WrapPOrder` qui dispose d'une égalité, d'une fonction de choix et d'un ordre. Avec `WrapPOrder` et `SemiRing` , nous sommes capables de formaliser les Définitions 5.5 et 5.7 ensemble.

Avec `Hierarchy Builder`, nous écrivons le code suivant :

```
HB.mixin Record Dioid_of_SemiRing_and_WrapPOrder D of SemiRing D & WrapPOrder D := {
  adddd : @idempotent D add;
  le_def : ∀ (a b : D),
      (Order.POrder.le wrap_porderMixin a b) = (Equality.op wrap_eqMixin (add a b) b);
}.
```

Nous utilisons le même mixage de propriétés que pour `Semiring`, avec ici l'extension avec `WrapPOrder` décrite plus haut. Ensuite, nous retrouvons la propriété d'idempotence de la loi additive, nommée `adddd` , et nous associons directement la relation d'ordre au *dioïde* par `le_def`. La définition de cette relation est très explicite car :

— `Order.POrder.le` a besoin de la connexion à `order`,

— `Equality.op` a besoin de la connexion à `eqType`.

mais les deux termes de l'égalité ne sont ni plus ni moins que :

— (`Order.POrder.le wrap_porderMixin a b`) pour $a \preccurlyeq b$

— (`Equality.op wrap_eqMixin (add a b)b`) pour $a \oplus b = b$.

Nous avons fait le choix ici d'associer directement la relation d'ordre avec le *dioïde*. De cette manière, lors de la déclaration d'une instance de *dioïde*, l'ordre n'est pas dissocié de celui-ci et grâce aux Structures Canoniques de Coq, il n'y a pas de différence entre l'ordre de l'instance et l'ordre du *dioïde*. Cette solution existe déjà et a été traitée par AFFELDT et al. dans [Aff+20]. Les auteurs appellent cette technique *forgetful inheritance*

La déclaration de la structure est alors :

```
HB.structure Definition Dioid :=
  { D of SemiRing D & WrapPOrder D & Dioid_of_SemiRing_and_WrapPOrder D }.
```

où nous formalisons qu'un `Dioid` est un `SemiRing`, un `WrapPOrder` et le mixage de propriétés que nous venons d'énoncer. De même que pour un *semi-anneau commutatif*, nous avons pour un *dioïde commutatif* :

```
HB.structure Definition ComDioid := { D of Dioid D & ComSemiRing_of_SemiRing D }.
```

Remarque 5.3 (*Totalité de la relation d'ordre du dioïde*) La relation d'ordre $\preccurlyeq$ d'une instance de *dioïde* n'a pas besoin d'être totale. Il se trouve que c'est tout de même le cas dans le *dioïde* $(\overline{\mathbb{R}}, max, +)$ présenté dans l'exemple 5.3.

Comme pour le *semi-anneau commutatif*, les propriétés données ici sont réutilisables pour chaque instance de *dioïde*. Nous avons aussi d'autres propriétés applicables sur un *dioïde*.

Lemme 5.3 (`le0d`) Pour un *dioïde* D et pour tout $a \in \mathcal{D}$, nous avons $\bar{0} \preccurlyeq a$.

DÉMONSTRATION (LEMME 5.3) Par définition de $\preccurlyeq$, nous avons $\bar{0} \preccurlyeq a \iff \bar{0} \oplus a = a$ et donc par absorption de $\bar{0}$ par $\oplus$, la preuve est conclue. □

Lemme 5.4 (`led_add2r, led_add2l`) Pour un *dioïde* D et pour tout a, b et $c \in \mathcal{D}$ tel que $a \preccurlyeq b$, nous avons : $a \oplus c \preccurlyeq b \oplus c$ et $c \oplus a \preccurlyeq c \oplus b$.

Lemme 5.5 (Monotonie de $\oplus$ `led_add`) Pour tout *dioïde* $\mathcal{D}$ et pour a, b, c et $d \in \mathcal{D}$, nous avons

$$a \preccurlyeq c \implies b \preccurlyeq d \implies a \oplus b \preccurlyeq c \oplus d.$$

Lemme 5.6 (`led_addl, led_addr`) Pour un *dioïde* D et a et $b \in \mathcal{D}$ nous avons : $a \preccurlyeq a \oplus b$ et $a \preccurlyeq b \oplus a$.

Lemme 5.7 (`led_mul2r, led_mul2l`) Pour un *dioïde* D et pour tout a, b et $c \in \mathcal{D}$ tel que $a \preccurlyeq b$, nous avons : $a \otimes c \preccurlyeq b \otimes c$ et $c \otimes a \preccurlyeq c \otimes b$.

Lemme 5.8 (Monotonie de $\oplus$ `led_mul`) Pour tout *dioïde* $\mathcal{D}$ et pour a, b, c et $d \in \mathcal{D}$, nous avons

$$a \preccurlyeq c \implies b \preccurlyeq d \implies a \otimes b \preccurlyeq c \otimes d.$$

Les lemmes 5.8 et 5.5 sont traduis en Coq à l'aide des lemmes 5.4, 5.6 et 5.7. De plus, ces derniers seront réutilisables sans passer par les contraintes amenés par les généralisations des lemmes 5.8 et 5.5.

Pour permettre une meilleure lisibilité dans la suite de ce Chapitre, nous donnons quelques notations qui sont :

```
Notation "0" := zero : dioid_scope.
Notation "1" := one : dioid_scope.
Notation "+ %D" := (@add _) : dioid_scope.
Notation "*%D" := (@mul _) : dioid_scope.
Infix "+ " := (@add _) : dioid_scope.
Infix "*" := (@mul _) : dioid_scope.
```

Le `dioide_scope` sera utilisé avec une lettre, comme pour les `Scope` présenté dans le Chapitre 4. Pour celui ci, nous utiliserons `%D`.

Dans l'exemple 5.4, nous montrons une application du Lemme 5.1 sur l'instance qui a servi d'exemple au cours de cette section, $(\overline{\mathbb{R}}, \max, +)$. Comme les autres Lemmes, il sera aussi applicable sur toutes les instances.

Exemple 5.4. Utilisation d'une propriété de *semi-anneau* sur l'instance $(\overline{\mathbb{R}}, \max, +)$

Dans cet exemple, nous allons montrer qu'il est possible d'utiliser une propriété d'un *semi-anneau* sur une instance de *semi-anneau*. Nous supposons dans cet exemple que la preuve pour montrer que $(\overline{\mathbb{R}}, \max, +)$ est une instance de *semi-anneau* a été développée en Coq.
Nous avons donc une première propriété :

```
Lemma Rbar_dioid_addE (x y : Rbar) : (x + y)%D = Order.max x y.
```

qui permet de changer l'opération d'addition du *dioïde*, $\oplus$ qui est noté `+` avec le `Scope` `%D`, en première opération de l'instance $(\overline{\mathbb{R}}, \max, +)$, max. La preuve de cette réécriture est triviale lorsque la déclaration d'instance a été donnée. Ensuite, prenons par exemple le `Goal` suivant :

∗ Goals ∗	

`Goal ∀ a, (a ⩽ Order.max a 0%D)%E.`
`Proof.`
`move⇒ a.`
	`a : {ereal R}`
	`————————————`
	`(a ⩽ Order.max a 0%D)%E`

qui peut se traduire par $\forall a, a \leqslant \max(a, \bar{0})$. L'objectif ici est d'utiliser le Lemme 5.1. Donc, nous devons d'abord modifier le max en $\oplus$ à l'aide du Lemme donné au dessus.

∗ Goals ∗	

`Goal ∀ a, (a ⩽ Order.max a 0%D)%E.`
`Proof.`
`move⇒ a.`
`rewrite -Rbar_dioid_addE.`
	`a : {ereal R}`
	`————————————`
	`(a ⩽ Builders_6.Super.add a 0%D)%E`

où la fonction `Builders_6.Super.add` correspond à la loi additive d'un *semi-anneau*. Elle prend cette notation avec la création du constructeur d'instance que nous avons ajouté à l'aide de `Hierarchy Builder`.
Ensuite nous pouvons réécrire le Lemme 5.1.

∗ Goals ∗	

`Goal ∀ a, (a ⩽ Order.max a 0%D)%E.`
`Proof.`
`move⇒ a.`
`rewrite -Rbar_dioidE.`
`rewrite add0.`
	`a : {ereal R}`
	`————————————`
	`(a ⩽ a)%E`

Suite de l'exemple 5.4

Le Goal est alors conclu puisque qu'il s'agit exactement de la propriété de réflexivité, donnée par `le_refl` :

* Goals *

```
Goal ∀ a, (a ⩽ Order.max a 0%D)%E.
Proof.
move⇒ a.
rewrite -Rbar_dioidE.
rewrite addd0.
exact: le_refl.
Qed.
```

Cet exemple conclut sur la formalisation de la structure algébrique d'un *dioïde* en Coq. Pour la suite de ce chapitre et en suivant [BBLC18], nous allons chercher à ajouter à la structure de *dioïde* un élément permettant d'obtenir un *dioïde complet*.

5.1.2 Un *dioïde* complet à partir d'un treillis complet

Pour suivre la définition de structure de [BBLC18], nous devons formaliser une notion de somme infinie. Elle fera office de borne supérieure pour le *dioïde* puisqu'elle est liée à la loi additive $\oplus$ du *dioïde*. Elle fait partie de la définition d'un *dioïde complet* que nous définirons une fois cette borne définie.

La formalisation d'une telle opération ne peut se faire directement. En effet, définir une opération sur une infinité de points n'est pas possible et nous devons donner quelques étapes de plus que dans [BBLC18].

5.1.2.1 Treillis complet

Il faut dans un premier temps formaliser la présence d'une borne sur notre structure. Une telle structure existe déjà et s'appelle un *treillis complet*. Un *treillis* est une autre structure algébrique qui devient complet lorsqu'il est borné. Nous allons donc formaliser cette définition et y associer des propriétés pour pouvoir ensuite l'utiliser dans la définition d'un *dioïde complet*. La formalisation de cette partie est disponible dans le fichier `complete_lattice.v`.

Définition 5.8 (Treillis complet) Nous appelons *treillis complet* un ensemble $\mathcal{T}$ muni d'une relation d'ordre $\preccurlyeq$ qui possède un opérateur d'ensemble $\bigoplus : \mathcal{P}(\mathcal{T}) \to \mathcal{T}$, tel que, pour tout $S \subseteq \mathcal{T}$ l'élément $\bigoplus S$ est une borne supérieure de S, c'est à dire que $\bigoplus S$ est un majorant de S et $\bigoplus S$ est le plus petit des majorants

$$\forall x \in S, x \preccurlyeq \bigoplus S \quad \wedge \quad \forall y \in S, (\forall x \in S, x \preccurlyeq y) \implies \bigoplus S \preccurlyeq y$$

Comme nous l'avons expliqué avant cette définition, cette borne est liée à l'addition du *dioïde* d'où le choix de la notation $\bigoplus$. Pour comprendre ce lien, considérons que $\oplus$ est un maximum entre deux valeurs, $\bigoplus$ est alors la plus grande de toutes ces valeurs. Par dualité, cette borne devient la plus petite des valeurs si $\oplus$ est un minimum.

Remarque 5.4 (*Abus de Notation*) Nous noterons souvent $\bigoplus_{i=0}^{+\infty} E_i$ pour $\bigoplus \{E_i \mid i \in \mathbb{N}\}$.

Pour la traduction en Coq, nous avons d'abord formalisé la notion de borne, et donc l'opération $\bigoplus$ mais pour n'importe quelle structure. Celle ci est décrite ci dessous.

```
Definition set_f_is_lub T (eM : Equality.mixin_of T)
                          (oM : Order.POrder.mixin_of eM) (set_f : set T → T) :=
  (∀ (S : set T) x, S x → Order.POrder.le oM x (set_f S)) ∧
  (∀ (S : set T) y, (∀ x, S x → Order.POrder.le oM x y) → Order.POrder.le oM (set_f S) y).
```

La fonction `set_f_is_lub` prend en entrée un Type `T`, correspondant à $\mathcal{T}$ dans la Définition 5.8, deux constructeur `eM` et `oM` pour vérifier que `T` est bien équipé d'un critère d'égalité et d'une relation d'ordre. La dernière entrée, `set_f`, est la borne supérieure. La fonction `Order.POrder.le` est la relation d'ordre pour un ensemble ordonnée, ici `oM` pour chaque appel.

Le retour de la fonction est la définition de la borne supérieure : pour une sous partie `S` de `T`,

— `set_f` est plus grand élément que n'importe quel élément de `S` c'est à dire que pour tout `x` dans `S`, `x` $\preccurlyeq$ `set_f S`

— et `set_f` est le plus petit des majorants de `S`, c'est à dire que pour tout `y` dans `T`, si pour tout `x` dans `S`, `x` $\preccurlyeq$ `y` alors `set_f S` $\preccurlyeq$ `y`.

Pour formaliser la structure algébrique d'un *treillis complet*, nous utilisons aussi `Hierarchy Builder` pour donner un mixeur de propriétés :

```
HB.mixin Record CompleteLattice_of_WrapLattice T of WrapLattice T := {
  set_join : set T → T;
  set_join_is_lub : set_f_is_lub wrap_porderMixin set_join;
}.
```

Nous avons une connexion avec les treillis de `MathComp` défini dans la structure `WrapLattice`. Cette connexion est une extension de `WrapPOrder` défini dans la Section précédente (aussi défini dans le fichier `HB_wrappers.v`). Dans cette définition, nous appelons donc l'opérateur d'ensemble du trellis `set_join` et lui associons la propriété de plus petit majorant dans `set_join_is_lub`.

Comme dans la Section précédente, nous donnons la définition de la structure d'un *treillis complet*.

```
HB.structure Definition CompleteLattice :=
  { T of WrapLattice T & CompleteLattice_of_WrapLattice T }.
```

Nous allons maintenant ajouter des propriétés et des éléments à un treillis complet, comme nous l'avons fait pour le *semi-anneau* et le *dioïde*.

Lemme 5.9 `set_join_le_incl` Pour tout $\mathcal{T}$ un treillis complet et S et $S' \subseteq \mathcal{T}$, nous avons

$$S \subseteq S' \implies \bigoplus S \preccurlyeq \bigoplus S'.$$

Lemme 5.10 (`set_join_unique`) Pour $\mathcal{T}$ un treillis complet et $S \subseteq \mathcal{T}$, l'élément $\bigoplus S$ est l'unique plus grand majorant de S, c'est à dire que pour tout $a \in \mathcal{T}$, nous avons :

$$(\forall x \in S, x \preccurlyeq a) \implies (\forall y \in S, (\forall x \in S, x \preccurlyeq y) \implies a \preccurlyeq y) \implies a = \bigoplus S$$

Lemme 5.11 (`set_join_set1`) Pour $\mathcal{T}$ un treillis complet et $x \in \mathcal{T}$, nous avons $\bigoplus \{x\} = x$.

Des éléments particuliers peuvent être définis sur cette structure. Nous avons en premier un élément majorant `top` et un élément minorant `bottom`.

Définition 5.9 (Top et bottom) Nous dénotons les éléments *top* et *bottom* par $\top$ et $\bot$ et nous les définissons par, pour tout treillis complet $\mathcal{T}$:

$$\top = \bigoplus \mathcal{T} \quad \text{et} \quad \bot = \bigoplus \emptyset$$

Pour la formalisation en Coq, nous pouvons utiliser les éléments données par `MathComp Analysis` qui sont : `set0` et `setT` correspondant respectivement à l'ensemble vide et l'ensemble plein.

Il vient directement par cette définition les deux propriétés suivantes :

Lemme 5.12 (`bottom_minimum, top_maximum`) Pour $\mathcal{T}$ un treillis complet et pour tout $x \in \mathcal{T}$, nous avons :

$$x \preccurlyeq \top \quad \wedge \quad \bot \preccurlyeq x.$$

Nous souhaitons manipuler des ensembles et unifier des ensembles dans les prochaines propriétés. Dans ce chapitre, l'union de deux ensembles A et B, notée $A \cup B$, est l'ensemble des x tel que $x \in A$ ou $x \in B$. Pour la formalisation, nous prenons la fonction `setU` disponible dans `MathComp Analysis`. Elle viens avec une notation qui est :

```
* Reponses *
Notation "A `|` B" := (setU A B) : classical_set_scope
```

Prendre une fonction existante pour décrire un opérateur nous permet d'obtenir toute la théorie qui a déjà été développée autour de cet opérateur.

Remarque 5.5 (*Notation '|'*) Dans les librairies importées et dans le développement que nous présentons, la notation '|' est aussi utilisée pour la fonction `Order.join`. Cette fonction vient de la connexion à la structure algébrique d'un treillis.

```
* Reponses *
Notation "x `|` y" := (Order.join x y) : order_scope
```

Elle est alors ambiguë avec la notation introduite pour la fonction `setU`.

Lemme 5.13 (`set_joinUl, set_joinUr`) Soit $\mathcal{T}$ un treillis complet et S et $S' \subseteq \mathcal{T}$. Pour tout $y \in \mathcal{T}$, si $y \in S$ alors nous avons $y \preccurlyeq \bigoplus S \cup S'$ et si $y \in S'$ alors nous avons $y \preccurlyeq \bigoplus S \cup S'$.

Démonstration (Lemme 5.13) Si $y \in S$, alors nous avons $y \in S \cup S'$ et donc, par définition de $\bigoplus$ de la définition 5.8, nous avons $y \preccurlyeq \bigoplus \{S \cup S'\}$. Nous avons la même conclusion par commutativité de l'union (`setUC` de `MathComp Analysis`) quand $y \in S'$. □

Lemme 5.14 (`set_joinU_ge_l, set_joinU_ge_r`) Pour $\mathcal{T}$ un treillis complet et S et $S' \subseteq \mathcal{T}$, nous avons :

$$\bigoplus S \preccurlyeq \bigoplus S \cup S' \quad \wedge \quad \bigoplus S' \preccurlyeq \bigoplus S \cup S'$$

Démonstration (Lemme 5.14) Avec le lemme 5.13, nous savons que pour tout $x \in S, x \preccurlyeq \bigoplus \{S \cup S'\}$. Avec la définition 5.8, cela revient donc à dire que $\bigoplus S \preccurlyeq \bigoplus \{S \cup S'\}$. Nous obtenons la même conclusion avec $x \in S', x \preccurlyeq \bigoplus \{S \cup S'\}$ du lemme 5.13. □

Lemme 5.15 (`set_join_set1`) Pour tout treillis complet $\mathcal{T}$ et $x \in \mathcal{T}$, nous avons $\bigoplus \{x\} = x$.

Démonstration (Lemme 5.15) Montrons que $x \preccurlyeq \bigoplus \{x\}$ et $\bigoplus \{x\} \preccurlyeq x$. Pour le premier cas, nous avons forcément $x \in \{x\}$ donc la Définition 5.8 conclut. Dans le deuxième cas, par réflexivité de $\preccurlyeq$, nous savons que pour tout $x_0 \in \{x\}, x_0 \preccurlyeq x$ donc la Définition 5.8 conclut. □

Lemme 5.16 (`set_joinU_le`) Pour un treillis complet $\mathcal{T}$ et S et $S' \subseteq \mathcal{T}$, si pour tout $x \in \mathcal{T}$, on a $\bigoplus S \preccurlyeq x$ et $\bigoplus S' \preccurlyeq x$ alors nous avons $\bigoplus S \cup S' \preccurlyeq x$.

5.1.2.2 *Dioïde* complet

La structure algébrique de *treillis complet* nous permet de formaliser une borne supérieure sur un ensemble pour un *dioïde*. Cette spécification est l'objectif de la prochaine définition.

Définition 5.10 (Dioïde complet) Un *dioïde* $(\mathcal{D}, \oplus, \otimes)$ est un *dioïde complet* si c'est un treillis complet avec comme borne supérieure $\bigoplus$ tel que, pour tout $A \subseteq \mathcal{D}$ et pour tout $b \in \mathcal{D}$,

$$\begin{aligned} \left(\bigoplus A\right) \otimes b &= \bigoplus \{a \otimes b \mid a \in A\} \\ b \otimes \left(\bigoplus A\right) &= \bigoplus \{b \otimes a \mid a \in A\} \end{aligned}$$

Comme pour le *semi-anneau*, nous pouvons aussi définir un *dioïde complet commutatif* à partir de cette dernière structure et un *semi-anneau commutatif*.

Définition 5.11 (Dioïde complet commutatif) Si $(\mathcal{D}, \oplus, \otimes)$ est un *dioïde complet* et un *semi-anneau commutatif*, alors $(\mathcal{D}, \oplus, \otimes)$ est un *dioïde complet commutatif*.

Exemple 5.5. Dioïde complet $(\overline{\mathbb{R}}, \max, +)$

Pour poursuivre l'exemple 5.3, le *dioïde* $(\overline{\mathbb{R}}, \max, +)$ est un *dioïde complet* avec l'opérateur d'ensemble sup car, pour tout $A \subseteq \overline{\mathbb{R}}$,

$$\forall x \in A, x \leqslant \sup\{a \mid a \in A\} \quad \wedge \quad \forall y \in A, (\forall x \in A, x \leqslant y) \implies \sup\{a \mid a \in A\} \leqslant y$$

donc sup respecte les conditions de la Définition 5.8 et pour tout $b \in \overline{\mathbb{R}}$,

$$(\sup\{a \mid a \in A\}) + b = \sup\{a + b \mid a \in A\} \quad \wedge \quad b + (\sup\{a \mid a \in A\}) = \sup\{b + a \mid a \in A\}$$

donc sup respecte les conditions de la Définition 5.10.

La formalisation d'un *dioïde complet* se trouve dans le fichier `complete_dioid.v`. Nous supposons donc ici que le fichier `complete_lattice.v` a été importé correctement.

Comme pour les autres structures, un premier mixage de propriétés :

```
HB.mixin Record CompleteDioid_of_Dioid_and_CompleteLattice D
        of Dioid D & CompleteLattice D := {
  set_mulDl : ∀ (a : D) (B : set D), a * set_join B = set_join [set a * x | x in B];
  set_mulDr : ∀ (a : D) (B : set D), set_join B * a = set_join [set x * a | x in B];
}.
```

où nous retrouvons les deux propriétés sur l'opérateur `set_join` avec la notation : `[set a * x | x in B]` qui indique l'ensemble des `a * x` tel que `x in B`.

```
HB.structure Definition CompleteDioid :=
  { D of Dioid D & CompleteLattice D & CompleteDioid_of_Dioid_and_CompleteLattice D }.
```

Nous définissons ensuite une nouvelle fonction pour utiliser `set_join`.

```
Section CompleteDioidTheory.
Variables D : CompleteDioid.type.

Definition set_add : set D → D := set_join.
```

De cette manière, nous avons la fonction `set_add` pour l'opérateur d'ensemble $\bigoplus$ et celle-ci ne fonctionne que sur $\mathcal{P}(\mathcal{D})$ pour $\mathcal{D}$ un *dioïde complet*.

Avant de déclarer des instances ou de définir de nouveaux opérateurs sur cette structure, nous allons donner des propriétés générales sur les *dioïde complet* que nous utiliserons ensuite sur les instances.

Le prochain lemme porte sur l'union et l'opérateur d'ensemble du *dioïde complet*.

Lemme 5.17 (`set_addDl`) Nous avons, pour tout $a \in \mathcal{D}$ et $B \subseteq \mathcal{D}$,

$$a \oplus \left(\bigoplus B \right) = \bigoplus \{a\} \cup B.$$

Lemme 5.18 (`add_def`) Nous avons, pour tout a et $b \in \mathcal{D}$ $a \oplus b = \bigoplus \{a, b\}$.

Nous donnons deux propriétés sur *top* et *bottom* effectives pour un *dioïde complet*. La seconde propriété est séparée en deux `Lemma` Coq.

Lemme 5.19 (`bottom_zero`) Pour un *dioïde complet* $\mathcal{D}$, nous avons $\bot = \bar{0}$.

Démonstration (Lemme 5.19) Montrer l'égalité $\bot = \bar{0}$ revient à montrer que $\bot \preccurlyeq \bar{0}$ et $\bar{0} \preccurlyeq \bot$. Le premier cas est vérifié par le Lemme 5.12 et le second avec le Lemme 5.3. □

Lemme 5.20 (`add_dtop, add_topd`) L'élément $\top$ est absorbant à gauche et à droite pour la loi $\oplus$.

Les lemmes suivants sont des égalités pour un *dioïde complet* $\mathcal{D}$ et pour une suite $F : \mathbb{N} \to \mathcal{D}$. Nous noterons les éléments F_0, F_1, etc dans $\mathcal{D}$ les valeurs prises par cette suite.

Lemme 5.21 (`set_add_0, set_add_1`) Pour un *dioïde complet* $\mathcal{D}$ et $F : \mathbb{N} \to \mathcal{D}$, nous avons :

$$\bigoplus \{F_i \mid i < 0\} = \bar{0} \quad \wedge \quad \bigoplus \{F_i \mid i < 1\} = F_0$$

Lemme 5.22 (`set_add_S`) Pour un *dioïde complet* $\mathcal{D}$ et $F : \mathbb{N} \to \mathcal{D}$, nous avons :

$$\forall k \in \mathbb{N}, \bigoplus \{F_i \mid i \leqslant k\} = \bigoplus \{F_i \mid i < k\} \oplus F_k$$

Les prochains lemmes vont servir à manipuler des suites avec la relation d'ordre du *dioïde*.

Lemme 5.23 (`set_add_led`) Pour un *dioïde complet* $\mathcal{D}$ et $F : \mathbb{N} \to \mathcal{D}$, nous avons :

$$\forall k \in \mathbb{N}, \bigoplus \{F_i \mid i < k\} \preccurlyeq \bigoplus \{F_i \mid i \in \mathbb{N}\}$$

Lemme 5.24 (`set_add_lim_nat`) Pour un *dioïde complet* $\mathcal{D}$ avec $F : \mathbb{N} \to \mathcal{D}$ et $l \in \mathcal{D}$, nous avons :

$$\left(\forall k \in \mathbb{N}, \bigoplus \{F_i \mid i < k\} \preccurlyeq l \right) \implies \bigoplus_{i=0}^{+\infty} F_i \preccurlyeq l.$$

Lemme 5.25 (`set_add_led_set`) Pour un *dioïde complet* $\mathcal{D}$ avec F et $F' : \mathbb{N} \to \mathcal{D}$, nous avons :

$$\left(\forall k \in \mathbb{N}, \bigoplus \{F_i \mid i < k\} \preccurlyeq \bigoplus \{F'_i \mid i < k\} \right) \implies \bigoplus_{i=0}^{+\infty} F_i \preccurlyeq \bigoplus_{i=0}^{+\infty} F'_i$$

Ces propriétés concluent la partie sur le *dioïde complet*. Nous allons maintenant définir de nouveaux opérateurs sur cette structure algébrique.

5.1.3 Théorie pour l'étoile de Kleene

Nous considérons maintenant que la structure de *dioïde complet* est créée. En plus de l'opérateur d'ensemble, nous décrivons ici l'opérateur de Kleene, aussi connu sous le nom d'étoile de Kleene. Le développement de cette partie est dans le fichier `complete_dioid.v`.

Définition 5.12 (Puissance dans un *dioïde complet*) Pour un *dioïde complet* $(\mathcal{D}, \oplus, \otimes)$, nous définissons $a^0 = \bar{1}$ et pour tout $i \in \mathbb{N}, a^{i+1} = a \otimes a^i$.

```
Definition exp a n := iterop n *%D a 1.
```

La Définition 5.12 admet la prochaine propriété pour un *dioïde complet*.

Lemme 5.26 (`expSr`) Pour $\mathcal{D}$ un *dioïde complet*, $a \in \mathcal{D}$ et $i \in \mathbb{N}$, nous avons $a^{i+1} = a^i \otimes a$.

> **Remarque 5.6** (*expSr*) La propriété énoncée dans le Lemme 5.26 est vérifiée alors que la loi $\otimes$, dans le *dioïde complet*, n'est pas commutative.

Définition 5.13 (Étoile de Kleene) Pour un *dioïde complet* $(\mathcal{D}, \oplus, \otimes)$, l'*étoile de kleene* est l'opération telle que, pour $a \in \mathcal{D}$

$$a^\star \triangleq \bigoplus_{i=0}^{+\infty} a^i$$

De plus, nous notons $a^+ \triangleq \bigoplus_{i=1}^{+\infty} a^i$.

Traduit en Coq par :

```
Definition op_kleene a := set_add [set of exp a].
Definition op_plus a := set_add [set of fun i => exp a i.+1].
```

L'*étoile de kleene* est un opérateur pour un *dioïde complet*. Donc, cet opérateur sera utilisable sur toutes les instances de *dioïde complet*. Il en est de même pour les propriétés que nous présentons dans le reste de cette section. Nous considérons que nous disposons d'un *dioïde complet* $\mathcal{D}$ pour le reste de cette sous section.

Lemme 5.27 (Lien entre a^+ et $a^\star$, `kleeneSR`, `plusSR`) Pour tout $a \in \mathcal{D}$, nous avons :

$$a^\star = \bar{1} \oplus a^+ \quad \wedge \quad a^+ = a \otimes a^\star.$$

Démonstration (Lemme 5.27) D'après la définition, nous avons :

$$a^\star = \bigoplus_{i=0}^{+\infty} a^i = \bigoplus \{a^i | i \in \mathbb{N}\} = a^0 \oplus \bigoplus \{a^i | i > 1\} = \bar{1} \oplus \bigoplus_{i=1}^{+\infty} a^i.$$

De plus, $a \otimes \left(\bigoplus_{i=0}^{+\infty} a^i\right) = \left(\bigoplus_{i=0}^{+\infty} a \otimes a^i\right)$ par distributivité dans le *dioïde complet*, donc nous avons bien $a \otimes a^\star = \left(\bigoplus_{i=1}^{+\infty} a^i\right) = a^+$. □

Dans le prochain lemme, nous donnons le nom des `Lemma` Coq associé dans la liste des propriétés.

Lemme 5.28 Pour tout $a \in \mathcal{D}$, nous avons $a^+ \preccurlyeq a^\star$ (`plus_le_kleene`), $a \preccurlyeq a^+$ (`le_plus`) et $a \preccurlyeq a^*$ (`le_kleene`).

Lemme 5.29 (`kleene_monotony`) Pour a et $b \in \mathcal{D}$, nous avons $a \preccurlyeq b \implies a^\star \preccurlyeq b^\star$.

Lemme 5.30 Pour tout $a \in \mathcal{D}$, nous avons $(a \oplus \bar{1})^\star = a^\star$ (`kleene_1r`), $a^\star \otimes a^\star = a^\star$ (`kleene_sqr`), $(a^\star)^\star = a^\star$ (`kleene_kleene`) et pour $i \in \mathbb{N}$ tel que $(\bar{1} \leqslant i)$, $(a^\star)^i = a^\star$(`kleene_exp`).

Proposition 5.1 (`kleene_star_eq, kleene_star_least`) Pour tout a et $b \in \mathcal{D}$, le terme $a^\star b$ est la plus petite solution à l'équation :

$$x = ax \oplus b.$$

DÉMONSTRATION (PROPOSITION 5.1) Montrons d'abord que $a^\star b$ est solution de l'équation $x = ax \oplus b$. Nous avons donc :

$$\begin{aligned} a(a^\star b) \oplus b &= aa^\star b \oplus b \\ &= a^+ b \oplus b \\ &= (a^+ \oplus \bar{1})\ b = a^\star b \end{aligned}$$

Montrons enfin que $a^\star b$ est la plus petite solution, donc que :

$$\forall x \in \mathcal{D}, x = ax \oplus b \implies a^\star b \preccurlyeq x$$

Par distributivité de la multiplication dans un *dioïde complet*, cela revient à montrer que $\left(\bigoplus_{i=0}^{+\infty} a^i b\right) \preccurlyeq x$. D'après le Lemme 5.24 cela revient à montrer que : $\forall k, \bigoplus \left\{a^i b \mid i < k\right\} \preccurlyeq x$. Par récurrence sur $k \in \mathbb{N}$, nous savons que le cas initial est vrai car $\bigoplus \left\{a^i b | i < 0\right\} = \bot = \bar{0}$ avec la définition 5.9 et $\bar{0} \preccurlyeq x$ avec le lemme 5.3.

Pour démontrer le cas d'induction, nous allons d'abord montrer que, pour tout $n \in \mathbb{N}$, nous avons $a^n b \preccurlyeq x$. Par récurrence sur n, le cas initial est vérifié cette fois ci par définition de x : $b \preccurlyeq x = ax \oplus b$ et grâce à la monotonie de $\oplus$ (Lemme 5.5). Pour le cas d'induction, nous savons que : $a^n b \preccurlyeq x$ et que $\bar{0} \preccurlyeq b$ donc :

$$\begin{aligned} a^n b \preccurlyeq x &\implies aa^n b \preccurlyeq ax \\ &\implies a^{n+1} b \preccurlyeq ax \preccurlyeq ax \oplus b = x \\ &\implies a^{n+1} b \preccurlyeq x \end{aligned}$$

Sachant maintenant que pour tout n, on a $a^n b \preccurlyeq x$, il vient :

$$\bigoplus \left\{a^i b | i < k\right\} \preccurlyeq x \implies \bigoplus \left\{a^i b | i < k\right\} \oplus a^k b \preccurlyeq x \oplus x \implies \bigoplus \left\{a^i b | i < k+1\right\} \preccurlyeq x$$

qui vérifie la récurrence sur k et conclue la preuve. □

Lemme 5.31 (`kleene_add_mul`) Soit $(\mathcal{D}, \oplus, \otimes)$ un *dioïde complet commutatif*. Pour tout a et $b \in \mathcal{D}$, nous avons la relation $(a \oplus b)^\star = a^\star \otimes b^\star$

> **Remarque 5.7** (*Lemme `kleene_add_mul`*) Ce Lemme correspond à une généralisation de la Proposition 2.6 de [BBLC18] qui est seulement prouvée pour une instance particulière de *dioïde complet commutatif*. Nous avons donc généralisé la propriété pour tout *dioïde complet commutatif*. Cette propriété est probablement fausse sur les autres structure de *dioïde* que nous avons présenté, i.e sans la propriété de commutativité de la loi multiplicative.

5.1.4 Théorie de la résiduation

Afin d'obtenir une pseudo inverse pour la loi multiplicative, nous introduisons un opérateur de résiduation.

Définition 5.14 (Résiduation) Pour tout *dioïde complet* $(\mathcal{D}, \oplus, \otimes)$, nous appelons opération de résiduation l'opération $\oslash$ tel que, pour a et $b \in \mathcal{D}$:

$$a \oslash b \triangleq \bigoplus \{c \mid c \otimes b \preccurlyeq a\}$$

La traduction en Coq de cette opération se fait avec :

```
Definition div a b := set_add [set c | c * b ⩽ a].
```

Exemple 5.6. Résiduation sur $(\overline{\mathbb{R}}, \max, +)$

Sur $(\overline{\mathbb{R}}, \max, +)$, l'opération de résiduation, pour a et $b \in \overline{\mathbb{R}}$, est $a \oslash b = \sup \{c \mid c + b \leqslant a\} = \sup \{c \mid c \leqslant a - b\}$ donc cela correspond à

$$a \oslash b = a - b$$

où la soustraction sur $\overline{\mathbb{R}}$ est définie de manière usuelle, avec $+\infty - +\infty = +\infty$ et $-\infty - -\infty = +\infty$.

Nous donnons dans le reste de cette sous section des propriétés et équivalences de l'opération de résiduation avec l'ordre et la multiplication. Nous supposerons avoir un *dioïde complet* $(\mathcal{D}, \oplus, \otimes)$ dans le reste de cette partie.

Lemme 5.32 (`div_mul_le`, `mul_div_le`) Nous avons pour tout a et $b \in \mathcal{D}$:

$$(a \oslash b) \otimes b \preccurlyeq a \quad \wedge \quad a \preccurlyeq (a \otimes b) \oslash b.$$

Lemme 5.33 (`mul_div_equiv`) La résiduation est liée avec l'opérateur $\otimes$ du *dioïde*, pour tout a, b et x des éléments de $\mathcal{D}$, par la relation :

$$x \otimes a \preccurlyeq b \Leftrightarrow x \preccurlyeq b \oslash a.$$

Lemme 5.34 (`led_divl`, `led_divr`, `led_div`) Soit a, b, c et $d \in \mathcal{D}$. Si $a \preccurlyeq b$ et $c \preccurlyeq d$, alors nous avons

$$a \oslash c \preccurlyeq b \oslash c \quad \wedge \quad c \oslash b \preccurlyeq c \oslash a \quad \wedge \quad a \oslash d \preccurlyeq b \oslash c.$$

Lemme 5.35 (`addd_div_le`) Pour tout a, b et $c \in \mathcal{D}$ nous avons : $(a \oslash c) \oplus (b \oslash c) \preccurlyeq (a \oplus b) \oslash c$.

Lemme 5.36 (`div_mul`, `mul_divA`) Pour tout a, b et $c \in \mathcal{D}$ nous avons

$$a \oslash (b \otimes c) = a \oslash c \oslash b \quad \wedge \quad a \otimes (b \oslash c) \preccurlyeq (a \otimes b) \oslash c.$$

Nous avons ensuite des équivalences avec l'opérateur de Kleene.

Lemme 5.37 (`kleene_div_equiv`) Pour tout $a \in \mathcal{D}$ nous avons : $a = a^\star \iff a = a \oslash a$.

Lemme 5.38 (`mul_kleene`, `div_kleene`) Pour tout a et $b \in \mathcal{D}$ nous avons :

$$a \otimes b^\star = (a \otimes b^\star) \oslash b^\star \quad \wedge \quad a \oslash b^\star = (a \oslash b^\star) \otimes b^\star.$$

Remarque 5.8 (*Généralisations sur un dioïde complet*) Les propriétés apportées par le Lemme 5.37 et le Lemme 5.38 sont des généralisations de la Proposition 2.9. Comme pour la Remarque 5.7, les propriétés originales sont décrites sur une instance de *dioïde complet*, nous les avons formalisé sur la structure de *dioïde complet*.

Enfin, nous donnons une propriété avec le *top* du *treillis complet.*

Lemme 5.39 (`div_top`) Pour tout $x \in \mathcal{D}$, nous avons : $\top \oslash x = \top$.

Ces lemmes concluent les définitions des structures algébriques et des opérateurs associés. Avant d'associer des instances à ces structures, nous allons ajouter des conditions de stabilité à ces structures.

5.1.5 Sous-structures algébriques

Une fois les structures algébriques développées avec toutes les propriétés que nous avons citées, nous ajoutons la possibilité d'étendre les instances vers des instances proches. Les ensembles de ses nouvelles instances seront des sous ensemble d'une instance plus générale. Donc, quand nous avons une instance $\mathcal{D}$ pour un *dioïde complet*, nous souhaitons qu'il soit facile d'avoir une instance de *dioïde complet* sur un sous-ensemble de $\mathcal{D}$ (avec les mêmes lois).

Les opérations de cette nouvelle instance resteront les mêmes que l'instance originale, mais appliqués sur un nouvel ensemble. Il faut donc que ces opérations fonctionnent de la même manière que dans l'ensemble original. Cela revient à dire qu'il faut que les opérations soient stables pour ce nouvel ensemble. Nous avons donc besoin dans un premier temps de formaliser une condition de stabilité pour ces opérations.

La prochaine définition pose un prédicat pour qu'un *semi-anneau* soit stable.

Définition 5.15 (Clôture d'un *semi-anneau*) Soit $(\mathcal{D}, \oplus, \otimes, \bar{0}, \bar{1})$ un *semi-anneau* et $\mathcal{D}' \subseteq \mathcal{D}$. L'ensemble $\mathcal{D}'$ est clôt pour un *semi-anneau* signifie que nous avons :

$$\bar{0} \in \mathcal{D}' \quad \wedge \quad \bar{1} \in \mathcal{D}' \quad \wedge \quad \forall x, y \in \mathcal{D}', x \oplus y \in \mathcal{D}' \quad \wedge \quad \forall x, y \in \mathcal{D}', x \otimes y \in \mathcal{D}'.$$

Enfin nous avons la définition du prédicat pour un *dioïde complet*. Elle pose surtout une condition sur l'opérateur d'ensemble.

Définition 5.16 (Clôture d'un *dioïde complet*) Soit $\mathcal{D}$ un *dioïde complet* avec la borne supérieure $\bigoplus$ et $\mathcal{D}' \subseteq \mathcal{D}$. L'ensemble $\mathcal{D}'$ est clôt pour un *dioïde complet* signifie que nous avons :

$$\forall B \subseteq \mathcal{D}', \bigoplus B \in \mathcal{D}'.$$

Lemme 5.40 (Extension avec la Définition 5.15) Soit $\mathcal{D}$ un ensemble et $\mathcal{D}' \subseteq \mathcal{D}$. Soit $\oplus, \otimes$ des opérations binaires dans $\mathcal{D}$ et $\bigoplus$ une opération d'ensemble dans $\mathcal{D}$. Si $(\mathcal{D}, \oplus, \otimes)$ est un *semi-anneau*, si $\mathcal{D}'$ est clôt au sens de la Définition 5.15 alors $\mathcal{D}'$ est un *semi-anneau*. C'est aussi le cas pour un *semi-anneau commutatif*, un *dioïde* et un *dioïde complet*.

Lemme 5.41 (Extension avec la Définition 5.16) Soit $\mathcal{D}$ un ensemble et $\mathcal{D}' \subseteq \mathcal{D}$. Soit $\oplus, \otimes$ des opérations binaires dans $\mathcal{D}$ et $\bigoplus$ une opération d'ensemble dans $\mathcal{D}$. Si $(\mathcal{D}, \oplus, \otimes)$ est un *dioïde complet*, si $\mathcal{D}'$ est clôt au sens de la Définition 5.16 alors $\mathcal{D}'$ est un *dioïde complet*. C'est aussi le cas pour un *dioïde complet commutatif*.

Pour l'implémentation en Coq de cette partie, nous reprenons le mécanisme de `ssralg.v` de `MathComp`. Nous définissons des instances canoniques pour les définitions 5.15 et 5.16 et pour les lemmes 5.40 et 5.41. Ensuite, nous avons ajouter des notations pour déclarer des sous instances de *semi-anneau*, *semi-anneau commutatif*, *dioïde* et *dioïde complet* dans le fichier `dioid.v` et *dioïde complet* et *dioïde complet commutatif* dans le fichier `complete_dioid.v`. Par exemple, la notation ajoutée pour déclarer une nouvelle instance de *semi-anneau* est :

```
Notation "[ 'SemiRing' 'of' U 'by' <: ]" := (semiRingMixin (Phant U))
  (at level 0, format "[ 'SemiRing' 'of' U 'by' <: ]") : form_scope.
```

Nous l'utiliserons dans la prochaine section lors de l'instanciation de structure de *semi-anneau* à partir d'une instance de *semi-anneau*. Cela fera l'objet de sous partie au cours de cette section dans lesquelles nous ferons référence à cette partie.

5.2 Le *dioïde* des fonctions *(min, +)*

Maintenant que les structures algébriques sont définies, nous allons associer des instances à ses structures. Cette partie vise à pouvoir utiliser les propriétés de chaque structure sur une instance, c'est à dire, dans le cas d'un *semi-anneau* par exemple, un ensemble muni de deux lois.

Comme dans la partie précédente, nous allons donner des définitions d'éléments que nous avons formalisé en Coq. Pour pouvoir suivre ce développement, il est nécessaire de compiler la librairie *dioïde* (donné dans le chapitre précédent) comme nous l'expliquons dans l'annexe A (page 111). Le code qui sera décrit dans cette partie provient d'un autre répertoire, plus spécifique au *Calcul réseau*, accessible via :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien`

De plus, tout le code de cette partie est contenue dans le fichier `RminStruct.v`, qui est accessible directement sur le lien suivant :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/blob/phd-lucien/RminStruct.v`

5.2.1 L'instance $(\overline{\mathbb{R}}, \min, +)$

Nous souhaitons associer l'instance qui a pour ensemble $\overline{\mathbb{R}}$ et les opérations min et + avec un *dioïde complet* afin de disposer de cette instance dans le développement des preuves du *Calcul réseau*. Dans la Section 5.1, nous avons décris dans les Exemples 5.2, 5.3 et 5.5 comment $(\overline{\mathbb{R}}, \max, +)$ est un *semi-anneau*, *dioïde* et *dioïde complet* respectivement. Pour comprendre le lien entre les deux instances, nous donnons l'exemple suivant.

Exemple 5.7. Dual de $(\overline{\mathbb{R}}, \max, +)$: $(\overline{\mathbb{R}}, \min, +)$

Par dualité, nous avons aussi un *semi-anneau*, *dioïde* et *dioïde complet* avec $(\overline{\mathbb{R}}, \min, +)$:

— La relation d'ordre $\preccurlyeq$ est inversé, nous avons ici $\geqslant$ (au lieu de $\leqslant$) pour avoir :

$$a \geqslant b \Leftrightarrow \min(a, b) = b$$

— L'opérateur d'ensemble $\bigoplus$ est, pour $A \subseteq \overline{\mathbb{R}}$ l'opération $\inf_{a \in A} a$.

L'addition ici n'est pas exactement la même entre les instances $(\overline{\mathbb{R}}, \min, +)$ et $(\overline{\mathbb{R}}, \max, +)$ comme déjà mentionné dans les Remarque 4.2 et 4.4. Dans l'instance $(\overline{\mathbb{R}}, \min, +)$, la somme de $+\infty$ et $-\infty$ donne $+\infty$ (cf. Définition 4.1).

Pour une traduction en Coq, il faut dans un premier temps déclarer que $\overline{\mathbb{R}}$ est bien une instance de `WrapChoice`, `WrapPOrder` et `WrapLattice`. Le type $\overline{\mathbb{R}}$ est traduit en Coq par (`\bar R`), où `\bar` est une notation pour la définition de la fonction `extended`. Elle est définie dans `MathComp Analysis` par dans le fichier `ereal.v` :

```
* Reponses *

Inductive extended (R : Type) : Type := EFin : R → \bar R | EPInf : \bar R | ENInf : \bar R
```

Cette fonction correspond à la fonction introduite et commentée dans le Chapitre 4. Les opérations

nécessaire à la déclaration de l'instance ont déjà été définies dans d'autres librairies ou du fichier `ereal.v` présent dans les sources. Nous commençons par la loi additive de l'instance. L'opération `Order.min` est donnée par le fichier `order` de `MathComp`. Elle est définie par :

* Reponses *

```
Order.min = fun (disp : unit) (T : porderType disp) (x y : T) ⇒ if (x < y)%O then x else y
```

Dans cette définition, il faut comprendre que l'ensemble `T` auquel `x` et `y` appartiennent est muni d'un ordre.

Pour la loi multiplicative, nous reprenons la Définition 4.1 (page 20) de l'addition de réel étendu afin d'avoir $+\infty$ comme élément $\bar{0}$ pour le *semi-anneau* et absorbant pour le min. (cf. Remarque 4.2). Nous rappelons ici sa définition.

Définition 5.17 (Addition dans $\overline{\mathbb{R}}$, `adde`)

$$\forall x, y \in \overline{\mathbb{R}}, x + y = \begin{cases} x + y & \text{si } x, y \in \mathbb{R} \\ +\infty & \text{si } x = +\infty \text{ ou } y = +\infty \\ -\infty & \text{sinon.} \end{cases}$$

La définition de `adde` se trouve dans le fichier `ereal.v`. Nous donnons des propriétés similaires à l'addition étendue donnée dans `MathComp Analysis` hormis quelques propriétés comme celle donnée dans la réponse suivante :

* Reponses *

```
addooe:
  ∀ {R : numDomainType} [x : \bar R], x != +∞ → −∞ + x = −∞
analysis.ereal.addooe:
  ∀ {R : numDomainType} [x : \bar R], x != −∞ → analysis.ereal.adde +∞ x = +∞
```

Le lemme `addooe` provient de notre fichier `ereal.v` alors que le lemme `analysis.ereal.addooe` provient de `MathComp Analysis`. Notre propriété est forcément différente car il sera impossible de conclure avec l'hypothèse `x != −∞` et donc nous devons l'adapter.

Afin de correspondre à la théorie des *dioïdes*, il nous faut munir notre instance de *dioïde* d'une relation d'ordre respectant la Définition 5.7. Nous prenons la relation $\geqslant$ puisque pour tout a et $b \in \overline{\mathbb{R}}$, la relation $a \geqslant b$ est équivalente à $\min(a, b) = b$. Ainsi, nous avons explicitement la relation :

$$a \preccurlyeq b \iff a \geqslant b$$

Enfin, pour l'opération d'ensemble $\bigoplus$, nous prenons l'opération inf : c'est le plus grand minorant pour $\overline{\mathbb{R}}$. Pour cet opérateur, nous reprenons directement celui développé dans `MathComp Analysis` : `ereal_inf` et obtenons les propriétés déjà connues sur cette fonction.

Nous avons ainsi la prochaine proposition qui montre que $(\overline{\mathbb{R}}, \min, +)$ est un *dioïde complet commutatif*. Cette proposition se trouve dans le fichier `RminStruct.v`.

Proposition 5.2 ($(\overline{\mathbb{R}}, \min, +)$ est un *dioïde complet commutatif*) L'instance $(\overline{\mathbb{R}}, \min, +)$ est un *dioïde complet commutatif* avec l'opérateur d'ensemble inf, l'élément zéro $+\infty$ et l'élément unitaire 0.

Démonstration (Proposition 5.2) Nous donnons le nom des propriétés Coq associées. Elles sont incluses dans le module `RbarDioid`.

Il suffit de montrer que l'opération min est associative (`meetA`), avec l'élément neutre $+\infty$ (`add0l`), commutative (`meetC`) et idempotente (`minxx`). De plus, l'opération $+$ est associative (`addeA`), l'élément neutre est 0 (`add0e`) et commutative `addeC`. Nous savons que l'addition est distributive à gauche par rapport au min (`mulDl`), et que $+\infty$ est absorbant pour l'addition (`mul0l`). Enfin, il suffit d'avoir la relation d'ordre (`le_def`) et de montrer que l'addition est distributive à gauche par rapport à inf avec le résultat déjà obtenu sur l'instance $(\overline{\mathbb{R}}, \min, +)$ (`set_mulDl`). □

La preuve associée à cette proposition est formalisée dans le fichier `RminStruct.v`. En particulier, nous y avons décrit les lemmes manquant pour la preuve. Nous avons ensuite donné la déclaration de la proposition dans une déclaration d'instance en utilisant `Hierarchy Builder`.

Pour terminer, nous ajoutons des lemmes de réécriture des opérations du *dioïde*. Ces lemmes servent à changer le min en $\oplus$ et $+$ en $\otimes$ et ainsi obtenir toutes les propriétés du *dioïde complet commutatif* (cf. Exemple 5.4).

5.2.2 Instance d'un sous ensemble de $\overline{\mathbb{R}}$

Nous voulons déclarer l'instance avec pour ensemble $\overline{\mathbb{R}}_+ \triangleq \{x \mid x \in \overline{\mathbb{R}} \wedge 0 \leqslant x\}$ muni des mêmes opérations que l'instance $(\overline{\mathbb{R}}, \min, +)$. L'ensemble $\overline{\mathbb{R}}_+$ est crée dans le fichier `nngenum.v`. Dans le fichier `RminStruct.v`, le lecteur pourra donc trouver la notation :

```
Notation "'barR+ '" := {nonnege R}.
```

qui nous permet d'avoir le type $\overline{\mathbb{R}}_+$ avec la notation `barR+` . Le nom des définitions et lemmes qui lui sont associées portent pourtant le nom de `nnRbar` pour *non-negative Rbar* puisque nous ne pouvons pas utiliser de signe dans un nom *Coq*.

Pour déclarer l'instance, nous allons utiliser les prédicats montrés en 5.1.5. Nous avons ainsi deux lemmes à prouver.

Lemme 5.42 (`nnRbar_semiring_closed`) L'ensemble $\overline{\mathbb{R}}_+$ est clos pour un *semi-anneau* au sens de la Définition 5.15.

Lemme 5.43 (`nnRbar_set_join_closed`) L'ensemble $\overline{\mathbb{R}}_+$ est clos pour un *dioïde complet* au sens de la Définition 5.16.

Nous pouvons alors démontrer la proposition suivante. Celle ci permet d'équiper $(\overline{\mathbb{R}}_+, \min, +)$ avec une structure de *dioïde complet commutatif*, comme la Proposition 5.2.

Proposition 5.3 ($(\overline{\mathbb{R}}_+, \min, +)$ est un *dioïde complet commutatif*) L'instance $(\overline{\mathbb{R}}_+, \min, +)$ est un *dioïde complet commutatif* avec l'opérateur d'ensemble inf, avec l'élément zéro $+\infty$ et l'élément unitaire 0.

Démonstration (Proposition 5.3) En utilisant les lemmes 5.42 et 5.43 ainsi que la Proposition 5.2. □

Le développement en Coq de cette partie réutilise donc les définitions données dans les fonctions `nnRbar_semiring_closed` et `nnRbar_set_join_closed`. Nous utilisons aussi la commande `HB.instance` de `Hierarchy Builder`.

5.2.3 Instances $(\mathcal{F}, \min, *)$: le *dioïde* des fonctions *(min, +)*

Nous allons maintenant déclarer l'instance utilisée dans le *Calcul réseau*. Celle ci utilise donc les fonctions $\mathcal{F} : \mathbb{R}_+ \to \overline{\mathbb{R}}$, déclarées dans la Définition 3.1 au cours de l'introduction au *Calcul réseau*. En *Coq*, la définition de cet ensemble est très ressemblante à la définition mathématique. Nous avons dans le fichier `RminStruct.v` :

```
Definition F := R+ -> \bar R.
```

Nous équipons cet ensemble d'un `eqType` et d'un `choiceType`, pour les mêmes raisons que le *semi-anneau* dans la section précédente, donc avoir une égalité et une fonction de choix.

Nous pouvons associer une relation d'ordre à notre ensemble.

Définition 5.1 (Relation d'ordre dans $\mathcal{F}$, `leF`) Nous définissons la relation d'ordre de $\mathcal{F}$ tel que, pour tout f et $g \in \mathcal{F}$: $f \leqslant g \triangleq t \mapsto f(t) \leqslant g(t)$

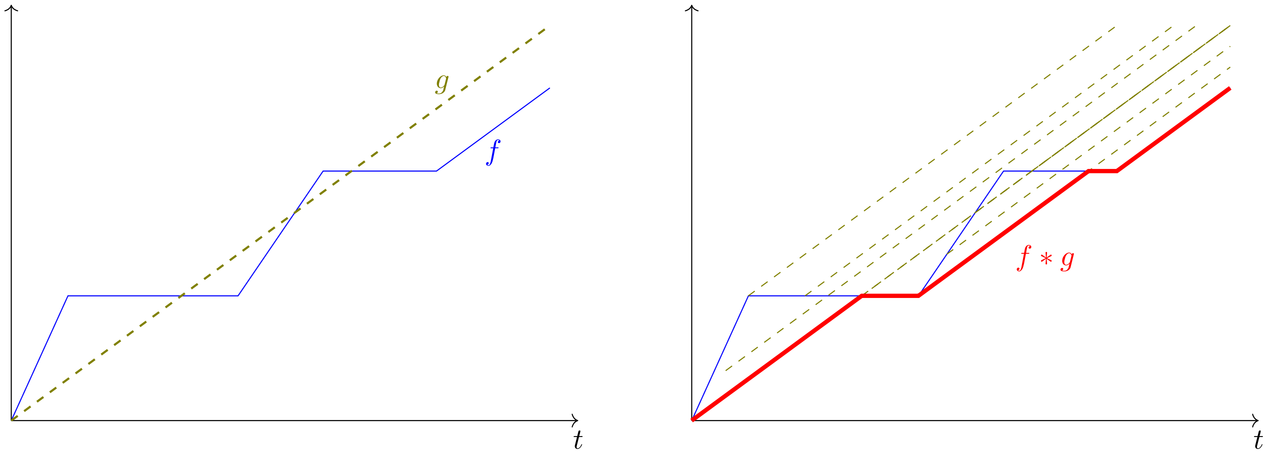


FIGURE 5.1 – Exemple de convolution entre f et $g \in \mathcal{F}$. Pour obtenir la convolution entre deux $\mathcal{F}$ comme sur la Figure 5.1, il faut faire glisser la fonction g sur la fonction f et prendre le minimum du résultat.

DÉMONSTRATION (DÉFINITION 5.1) C'est bien une relation reflexive (`leF_refl`), antisymétrique (`leF_anti`) et transitive (`leF_trans`). □

L'ordre défini dans la Définition 5.1 n'est pas total. Ce n'est pas un problème pour le *dioïde* (cf. Remarque 5.3). Avec cette relation d'ordre, nous pouvons déclarer $\mathcal{F}$ comme une instance de `porder` (*partial_ order*) (avec notre `WrapPOrder`) et ainsi utiliser les fonctions. Nous allons justement utiliser la fonction `Order.min` donnée pour l'instance $(\overline{\mathbb{R}}, \min, +)$ en 5.2.1. Nous allons appliquer point à point cette définition pour définir le minimum entre deux fonctions. Le code Coq correspondant est le suivant.

```
Definition F_min (f f' : F) : F := fun t ⇒ Order.min (f t) (f' t).
```

Pendant l'introduction au *Calcul réseau*, nous avons évoqué le produit de convolution entre deux fonctions. Cette opération est définie dans la prochaine définition.

Définition 5.2 (Produit de convolution) Pour $f, g \in F$, le produit de convolution entre f et g est défini par :

$$(f * g) \triangleq t \mapsto \inf_{\substack{u,v \geqslant 0 \\ u+v=t}} \{f(u) + g(v)\}$$

Nous donnons une intuition du produit de convolution entre deux fonctions en Figure 5.1. La traduction en Coq de cette opération est la suivante :

```
Definition F_min_conv (f g : F) : F :=
  fun t ⇒ ereal_inf [set (f uv.1 + g uv.2)%E
                    | uv in [set uv : R+ * R+ | uv.1%:nngnum + uv.2%:nngnum = t%:nngnum]].
```

Dans cette définition, nous reprenons la fonction `ereal_inf` pour l'inf. La notation [`set` ...] permet de décrire un ensemble. Le terme `uv` est une paire : nous accédons au premier (resp. second) élément avec .1, (resp. .2) et représente `u` (resp. `v`). De plus, `%E` permet de donner l'addition `adde`, définie plus tôt. Enfin, `%:nngnum` est une notation d'une fonction qui permet ici de passer un `R+` en `R`.

Nous allons donner trois définitions. Elles correspondent respectivement aux éléments zéro et unitaire du *dioïde* pour $\mathcal{F}$ et à l'opérateur d'ensemble du *dioïde complet* pour $\mathcal{F}$.

Définition 5.3 (`F_zero`) Nous définissons $f_{\bar{0}}$ la fonction $f_{\bar{0}} \triangleq t \mapsto +\infty$.

Définition 5.4 (`F_one`) Nous définissons $f_{\bar{1}}$ la fonction $f_{\bar{1}} \triangleq t \mapsto \begin{cases} 0 \text{ si } t = 0 \\ +\infty \text{ sinon} \end{cases}$.

Définition 5.5 (`F_inf`) Soit $F \subseteq \mathcal{F}$, nous définissons la fonction inf pour F comme :

$$\inf\{F\} \triangleq t \mapsto \inf\{f(t) \mid f \in F\}.$$

Proposition 5.4 (($\mathcal{F}$, min, $*$) est un *dioïde complet commutatif*) L'instance $(\mathcal{F}, \min, *)$ est un *dioïde complet commutatif* avec l'opérateur d'ensemble inf, avec l'élément zéro $f_{\bar{0}}$ et l'élément unitaire $f_{\bar{1}}$.

Démonstration (Proposition 5.4) Nous donnons le nom des propriétés Coq associées. Elles sont incluses dans le module `FDioid`.

Il suffit de montrer que l'opération min pour des fonctions est associative (`F_minA`), avec l'élément neutre $f_{\bar{0}}$ (`add0l`), commutative (`F_minC`) et idempotente (`F_minxx`). De plus, l'opération $*$ est associative (`F_min_convA`), l'élément neutre est $f_{\bar{1}}$ (`F_min_conv_1_l`) et commutative `F_min_convC`. Il suffit de montrer que la convolution est distributive à gauche par rapport au min (`mulDl`), et que $f_{\bar{0}}$ est absorbant pour la convolution (`mul0l`). Enfin, il suffit d'avoir la relation d'ordre (`le_def`) et de montrer que l'addition est distributive à gauche par rapport à inf (`set_mulDl`). □

Avec cette proposition, l'ensemble $\mathcal{F}$ obtient les opérations du *dioïde complet*, donc l'opération de résiduation et l'étoile de kleene. Celles-ci viennent avec l'ensemble des propriétés démontrées au cours de la description de la structure.

Remarque 5.9 (*Déconvolution*) L'opération de résiduation du *dioïde* est appelée dans cette instance la déconvolution. Nous n'allons donc pas le définir mais donner une notation à cette opération pour correspondre avec la définition de la résiduation (cf. Définition 5.14). Par exemple, pour $f, g \in \mathcal{F}$, la déconvolution de f et g est égale à :

$$\inf\{h \in \mathcal{F} | f \leqslant h * g\}.$$

Nous utiliserons la notation $x \oslash_D y$ pour la déconvolution de x et $y \in D$.

Comme pour les instances sur $\overline{\mathbb{R}}$ et $\overline{\mathbb{R}}_+$, nous avons décrit des propriétés pour traduire les opérations et éléments de $\mathcal{F}$ en opération et éléments du *dioïde complet*. Ces propriétés sont rassemblées dans la propriété Coq `F_dioidE`.

L'opération de convolution est définie dans la Définition 2.7 dans [BBLC18] avec une notation différente. Dans celle ci, pour $f * g$ comme défini dans la Définition 5.2, il est fait abstraction de la variable v qui est remplacé par $t - u$ dans l'ensemble utilisé dans inf. Cette écriture du produit est équivalente.

Lemme 5.44 (`alt_def_F_min_conv`) Soit f et $g \in \mathcal{F}$ et $t \in \mathbb{R}_+$, nous avons

$$(f * g)(t) = \inf\{f(u) + g(t-u) \mid 0 \leqslant u \leqslant t\}.$$

Nous préférons utiliser en Coq la définition 5.2 puisque la soustraction $t - u$ doit être un $\mathbb{R}_+$. Or, pour cela, il faut être capable de montrer que $t - u \geqslant 0$. Ce but est vrai sans arrêt mais il a tendance à alourdir les preuves.

Nous avons aussi le prochain lemme qui permet de déterminer le point de départ de la convolution de deux fonctions.

Lemme 5.45 (`F_min_conv0`) Pour tout f et $g \in \mathcal{F}$, nous avons $(f * g)(0) = f(0) + g(0)$.

Nous définissons ensuite une addition point à point pour l'ensemble $\mathcal{F}$ puis une fonction neutre pour cette opération.

Définition 5.6 (`F_plus`) Pour f et $g \in \mathcal{F}$, nous définissons $f + g \triangleq t \mapsto f(t) + g(t)$.

Définition 5.7 (`F_0`) Nous définissons la fonction $f_0 \triangleq t \mapsto 0$.

L'addition de fonctions $\mathcal{F}$ est une opération sur laquelle nous pouvons déclarer une instance de *monoïde commutatif* comme défini dans la Définition 5.2. Cette déclaration nécessite de montrer quelques propriétés, notamment l'existence d'un élément neutre : la fonction f_0 définie en Définition 5.7. Ainsi, nous pouvons utiliser les travaux autour des opérations d'ensembles de Coq [Ber+08].

Nous avons introduit dans le développement en Coq de nombreuses notations pour améliorer la lisibilité.

```
Infix "+ " := F_plus : F_scope.
Infix "*" := F_min_conv : F_scope.
Notation "f / g" := ((div f%F g%F) : F) : F_scope.
```

Pour rappel, la fonction `div` correspond à l'opération de résiduation du *dioïde complet*. Les fonctions $\mathcal{F}$ sont associées au `Scope` `F_scope` délimité par `%F`. Respectivement, ces notations sont pour l'addition, le produit de convolution et la déconvolution. Cette dernière ne fait pas l'objet d'une définition (cf Remarque 5.9).

5.2.4 Le sous ensemble des fonctions positives est un dioïde complet

Avec les prédicats de clôture de 5.1.5, nous allons déclarer comme pour $\overline{\mathbb{R}}_+$ des instances de *dioïde complet commutatif* sur des sous ensemble de $\mathcal{F}$. En effet, ces fonctions plus restrictives permettent d'obtenir des résultats en *Calcul réseau* en raison de leurs propriétés. La première sous instance concerne l'ensemble des fonctions $\mathcal{F}$ positives. Nous le formalisons dans la prochaine définition.

Définition 5.8 (`Fplus`) Nous définissons l'ensemble $\mathcal{F}^+ \triangleq \{f \in \mathcal{F} \mid \forall t, f(t) \geqslant 0\}$.

Les deux prochains lemmes sont les preuves que les opérations du *semi-anneau* et du *dioïde complet* sont clôt dans l'ensemble $\mathcal{F}^+$.

Lemme 5.46 (`Fplus_semiring_closed`) $\mathcal{F}^+$ est clos pour un *semi-anneau* au sens de la Définition 5.15.

Lemme 5.47 (`Fplus_set_join_closed`) $\mathcal{F}^+$ est clos pour un *dioïde complet* au sens de la Définition 5.16.

Et enfin nous pouvons déclarer une instance de *dioïde complet commutatif* sur $\mathcal{F}^+$.

Proposition 5.5 (($\mathcal{F}^+$, min, $*$) est un *dioïde complet commutatif*) L'instance $(\mathcal{F}^+, \min, *)$ est un *dioïde complet commutatif* avec l'opérateur d'ensemble inf, avec l'élément zéro $f_{\bar{0}}$ et l'élément unitaire $f_{\bar{1}}$.

Démonstration (Proposition 5.5) La preuve consiste à utiliser les lemmes 5.46 et 5.47 ainsi que la Proposition 5.4. □

Comme pour l'instance $(\mathcal{F}, \min, *)$, nous obtenons ici les opérations supplémentaires du *dioïde* (déconvolution, kleene,..) sans avoir à les redéfinir. Comme dans les autres instances, la traduction des éléments et opérations dans $\mathcal{F}^+$ vers le *dioïde* sont inclues dans un seul lemme, ici `Fplus_dioideE`. De plus, il est possible alors d'ajouter de nouvelles propriétés concernant les opérations héritées du *dioïde*. C'est le but du prochain lemme.

Lemme 5.48 (`Fplus_divE`) Pour tout f et $g \in \mathcal{F}^+$, nous avons, pour tout t :

$$f \oslash_{\mathcal{F}^+} g = t \mapsto \max\left(0, \sup\left\{f(t+u) - g(u) \mid u \in \mathbb{R}_+\right\}\right)$$

Nous ajoutons ensuite la définition d'une addition pour $\mathcal{F}^+$, comme nous l'avons fait dans la Définition 5.6. A la différence de cette précédente relation, il faut ici montrer que l'addition de deux fonctions dans $\mathcal{F}^+$ est une fonction dans $\mathcal{F}^+$.

Définition 5.9 (`Fplus_plus`) Nous définissons l'addition par

$$\forall f, g \in \mathcal{F}^+, (f+g) \triangleq t \mapsto f(t) + g(t)$$

qui est une fonction dans $\mathcal{F}^+$.

La traduction de cette définition en Coq utilise un `Program Definition`

```
Program Definition Fplus_plus (f g : Fplus) : Fplus := Build_Fplus (f + g)%F _.
```

La commande `Program Definition` permet de donner une définition à trous en demandant à Coq de créer automatiquement le but à prouver correspondant à chaque trou. Ce but est l'élément manquant à la fonction `Build_Fplus`, puisque celle ci prend deux arguments : la valeur d'une fonction et la preuve que celle ci est une fonction positive. Nous ne donnons pas directement cette preuve, d'où le `_` à la place de cet argument. Le but ainsi généré est :

```
Program Definition Fplus_plus (f g : Fplus) : Fplus :=
  Build_Fplus (f + g)%F _.
Next Obligation.
```

```
* Goals *
f, g : Fplus
______________________________
(f + g)%F \is a nonNegativeF
```

La commande `Next Obligation` permet de commencer la preuve. Le `Goal` ici généré par Coq demande donc de montrer que la somme de $f+g$ est positive. La preuve est triviale et n'est pas montrée ici.

Nous définissons ensuite un élément neutre à cette fonction d'addition.

Définition 5.10 (`Fplus_0`) Nous définissons la fonction $f_0^+ \in \mathcal{F}^+$ comme $f_0^+ \triangleq t \mapsto 0$.

Afin d'utiliser les propriétés données par les opérations d'ensemble de [Ber+08] (cf. Section 4.2), nous devons avoir un *monoïde commutatif*. Dans le prochain lemme, nous déclarons une instance de *monoïde commutatif* avec la loi définie dans la définition précédente, l'addition.

Lemme 5.49 (`Fplus_comoid`) L'ensemble $\mathcal{F}^+$ avec l'addition définie en 5.9 est un *monoïde commutatif* avec la fonction f_0^+ (cf. Définition 5.10) comme élément neutre.

5.2.5 Le sous ensemble des fonctions croissantes est un dioïde complet

Cette seconde instance concerne les fonctions dans $\mathcal{F}^+$ qui sont croissantes. Pour les définir, il nous faut d'abord un critère pour distinguer une fonction croissante.

Définition 5.11 (Clôture croissante) Pour une fonction $f \in \mathcal{F}$, nous appelons clôture croissante de f la fonction $t \mapsto \sup\{f(u) \mid u \leqslant t\}$. Nous noterons cette fonction $f_\uparrow$.

Pour mieux comprendre la Définition 5.11, nous donnons un exemple en Figure 5.2.

La prochaine propriété montre comment nous pouvons caractériser les fonctions croissantes avec la clôture croissante.

Lemme 5.50 (`non_decr_closureP`) Pour tout $f \in \mathcal{F}$,

$$(\forall x, y \in \mathbb{R}_+, x \leqslant y \implies f(x) \leqslant f(y)) \iff f_\uparrow \leqslant f$$

Nous avons ensuite d'autres propriétés sur la clôture croissante.

Lemme 5.51 (`le_non_decr_closure`) Pour tout $f \in \mathcal{F}$, nous avons $f \leqslant f_\uparrow$.

Lemme 5.52 (`non_decr_closure_non_decr`) Pour tout $f \in \mathcal{F}$ et pour tout x et $y \in \mathbb{R}_+$ tel que $x \leqslant y$, nous avons $f_\uparrow(x) \leqslant f_\uparrow(y)$.

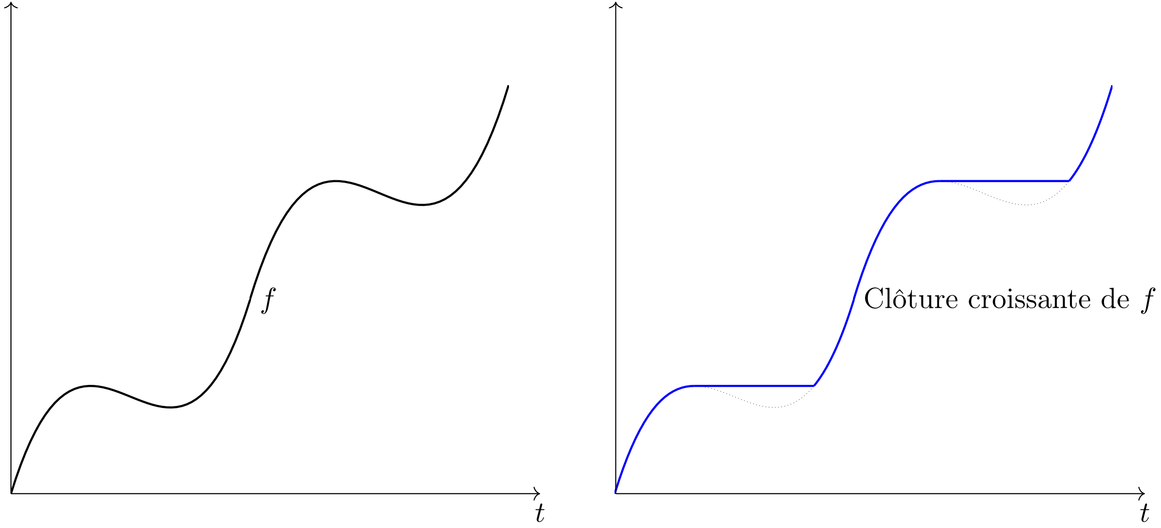


FIGURE 5.2 – Clôture croissante d'une fonction f : tracé en bleu

Lemme 5.53 (`non_decr_closure_F_min_conv_F_0`) Pour tout $f \in \mathcal{F}$, nous avons $f_\uparrow = -(-f * f_0)$.

Ce dernier lemme est utilisé dans une preuve spécifique du Chapitre 7. Avec la Définition 5.11, nous sommes capables de donner la définition des fonctions $\mathcal{F}^+$ croissantes.

Définition 5.12 (`Fup`) Nous définissons l'ensemble $\mathcal{F}^\uparrow \triangleq \{f \in \mathcal{F}^+ \mid f_\uparrow \leqslant f\}$.

L'ensemble $\mathcal{F}^\uparrow$ représente les fonctions croissantes. Nous le savons notamment grâce au Lemme 5.50 qui permet de réécrire le critère de clôture croissante dans la définition même d'une fonction croissante.

Nous avons ensuite la clôture avec le *semi-anneau* et le *dioïde complet*.

Lemme 5.54 (`Fup_semiring_closed`) $\mathcal{F}^\uparrow$ est clos pour un *semi-anneau* au sens de la Définition 5.15

Lemme 5.55 (`Fup_set_join_closed`) $\mathcal{F}^\uparrow$ est clos pour un *dioïde complet* au sens de la Définition 5.16.

Et enfin nous pouvons déclarer une instance de *dioïde complet commutatif* sur $\mathcal{F}^\uparrow$.

Proposition 5.6 (($\mathcal{F}^\uparrow$, min, $*$) est un *dioïde complet commutatif*) L'instance $(\mathcal{F}^\uparrow, \min, *)$ est un *dioïde complet commutatif* avec l'opérateur d'ensemble inf, avec l'élément zéro $f_{\bar{0}}$ et l'élément unitaire $f_{\bar{1}}$.

DÉMONSTRATION (PROPOSITION 5.6) La preuve consiste à utiliser les lemmes 5.54 et 5.55 ainsi que la Proposition 5.5. □

L'instance $(\mathcal{F}^\uparrow, \min, *)$ obtient maintenant les propriétés prouvées dans la théorie de la structure de *dioïde complet* et les opérateurs. Nous ajoutons aussi une traduction des opérations vers le *dioïde* dans `Fup_dioidE` ainsi que quelques définitions, notamment l'addition.

Définition 5.13 (`Fup_plus`) Pour tout f et $g \in \mathcal{F}^\uparrow$, nous définissons $f + g \triangleq t \mapsto f(t) + g(t)$ qui est une fonction dans $\mathcal{F}^\uparrow$.

L'implémentation en Coq reprend la fonction `Fplus_plus` pour ne pas avoir à prouver à nouveau que la fonction obtenue est positives. Il faut tout de même montrer qu'elle appartient à $\mathcal{F}^\uparrow$, et donc montrer que, pour tout t, $(f(t) + g(t))_\uparrow \leqslant f(t) + g(t)$. Nous donnons aussi un élément neutre.

Définition 5.14 (`Fup_0`) Nous définissons la fonction $f_0^\uparrow \in \mathcal{F}^\uparrow$ comme $f_0^\uparrow \triangleq t \mapsto 0$.

Cette définition est semblable à la Définition 5.10 de `Fplus_0`. Nous avons aussi un *monoïde commutatif*, comme pour $\mathcal{F}^+$ et l'addition. Nous pourrons ainsi utiliser l'opération `Fup_plus` sur des ensembles avec les propriétés des `big_operator` [Ber+08] avec l'élément $f_0^\uparrow$ comme élément neutre.

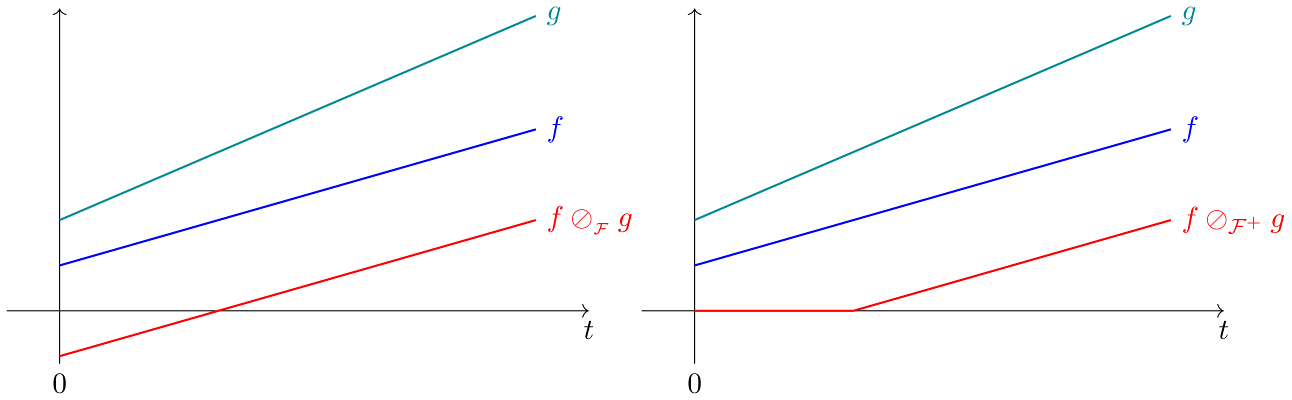


FIGURE 5.3 – Déconvolution entre les fonction f et $g \in \mathcal{F}$. À gauche, nous avons $f \oslash_{\mathcal{F}} g$ et à droite nous avons $f \oslash_{\mathcal{F}^+} g$. La fonction à droite ne peut admettre de valeur négative par définition. La déconvolution d'une telle fonction est calculé avec le Lemme 5.48.

Lemme 5.56 (`Fup_comoid`) L'ensemble $\mathcal{F}^{\uparrow}$ avec l'addition est un *monoïde commutatif* avec comme élément neutre $f_0^{\uparrow}$.

Nous avons établi dans cette sous partie une instance de *dioïde complet* avec l'ensemble $\mathcal{F}^{\uparrow}$. Comme pour les instances $\mathcal{F}$ et $\mathcal{F}^+$, une opération de déconvolution, noté $\oslash_{\mathcal{F}^{\uparrow}}$ a été ajouté lors de la création l'instance $\mathcal{F}^{\uparrow}$. Nous verrons dans la prochaine partie les liens entre les déconvolution des différentes instances.

5.2.6 Propriétés et lien dans les différentes instances de *dioïde complet*

Nous allons décrire plusieurs propriétés sur les instances de *dioïde* que nous avons déclaré dans ce chapitre. Une première propriété concerne la déconvolution et un point important qui sera réutilisé dans ce manuscrit.

Nous avons expliqué plus haut dans ce chapitre que chaque instance disposait d'une déconvolution. La prochaine propriété permet de lier les déconvolutions de $\mathcal{F}^+$ et $\mathcal{F}^{\uparrow}$.

Lemme 5.57 (`Fup_divE`) Pour tout f et $g \in \mathcal{F}^{\uparrow}$, nous avons :

$$f \oslash_{\mathcal{F}^{\uparrow}} g = f \oslash_{\mathcal{F}^+} g$$

Ce lemme montre donc que les déconvolutions sur $\mathcal{F}^+$ et $\mathcal{F}^{\uparrow}$ sont identiques. Ce n'est pas le cas pour la déconvolution sur $\mathcal{F}$. Nous n'avons pas de lien entre $\oslash_{\mathcal{F}}$ et $\oslash_{\mathcal{F}^+}$. Cependant, nous pouvons supposer que pour deux fonctions f et $g \in \mathcal{F}^+$, $f \oslash_{\mathcal{F}} g \leqslant f \oslash_{\mathcal{F}^+} g$. Nous donnons un exemple d'un tel cas dans la Figure 5.3.

Nous ajoutons ensuite des définitions d'éléments utilisés en *Calcul réseau* sur cet ensemble de fonctions. En effet, nous verrons que les définitions du *Calcul réseau* utilisent souvent les fonctions de l'ensemble $\mathcal{F}^{\uparrow}$, notamment pour les contraintes. Nous avons donc les deux prochaines définitions.

Définition 5.15 (`Fup_shift_l`) Nous définissons pour $f \in \mathcal{F}^{\uparrow}$ et $d \in \mathbb{R}_+$ la fonction $F_{shift}(f, d) \in \mathcal{F}^{\uparrow}$ comme $t \mapsto f(t + d)$.

Définition 5.16 (Pseudo inverse, `pseudo_inv_inf_nnR`**)** Soit $f \in \mathcal{F}^{\uparrow}$. Nous définissons la *pseudo inverse* de f comme la fonction qui à tout $x \in \overline{\mathbb{R}}_+$ associe

$$f^{-1}(x) \triangleq \inf \{t \mid x \leqslant f(t)\}.$$

La dernière définition peut être réécrite avec le prochain lemme.

Lemme 5.58 (`alt_def_pseudo_inv_inf_nnR`) Pour tout $f \in \mathcal{F}^{\uparrow}$ et $x \in \overline{\mathbb{R}}_+$, nous avons :

$$f^{-1}(x) = \max (0, \sup \{t \mid f(t) < x\}).$$

Ce lemme est une réécriture du Lemme 3.2 de [BBLC18]. Dans ce dernier, il n'est pas fait mention de $\max(0, \cdot)$ nécessaire pour gérer le cas où l'ensemble $\{t \mid f(t) < x\}$ est vide.

Remarque 5.10 ($\mathcal{F}_0$ *n'est pas un dioïde*) En *Calcul réseau*, les comportements sont souvent représentés par des fonctions de l'ensemble $\mathcal{F}^{\uparrow}$ démarrant à 0. Formellement, elles sont définies par :

$$\mathcal{F}_0 \triangleq \left\{ f \in \mathcal{F}^{\uparrow} \mid f(0) = 0 \right\}$$

(traduit en Coq avec le type `F0up`). Cet ensemble est stable pour toutes les opérations du *dioïde*. Cependant il ne contient pas l'élément $f_{\bar{0}} \in \mathcal{F}^{\uparrow}$. En effet, $f_{\bar{0}}$ vaut $+\infty$ pour tout t donc n'appartient pas à $\mathcal{F}_0$ et il n'est pas possible de trouver un autre élément absorbant pour la convolution. Nous ne pouvons en fait pas construire d'instance de *dioïde* sur cet ensemble puisque nous ne pouvons pas donner d'éléments absorbant pour la loi multiplicative.

5.3 Conclusion

Dans ce chapitre, nous avons commencé par formaliser les structures algébriques de *semi-anneau*, *dioïde*, *treillis complet* et *dioïde complet*. Nous avons ainsi pu donner les définitions formelles d'opérations : l'opération de kleene et l'opération de résiduation. Ces formalisations ont pris environ 2000 lignes de développement en Coq.

Ensuite, nous avons construit plusieurs instances de ces structures. Ces instances sont axées pour être utilisées dans la suite du développement de ce manuscrit puisque elles se basent sur le modèle mathématiques du *Calcul réseau*. Nous avons ainsi formalisé les instances sur les ensembles $\overline{\mathbb{R}}$ et $\overline{\mathbb{R}}_+$ munis des opérations min et $+$ puis sur les ensembles $\mathcal{F}$, $\mathcal{F}^+$ et $\mathcal{F}^{\uparrow}$ munis des opérations min et $*$ (c.f., Définition 5.2). La construction de ces instances et les quelques opérations et lemmes annexes décrits dans ce chapitre ont nécessité environ 1000 lignes de Coq. Ce code peut être retrouvé dans le répertoire :

`https://github.com/math-comp/dioid`.

pour la Section 5.1 et dans le fichier :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/blob/phd-lucien/RminStruct.v`

pour la Section 5.2.

Par rapport à la référence qui nous a servi de base ([BBLC18]), certaines parties sur le *dioïde* $(\mathcal{F}, \min, *)$ ont été démontrées dans le cadre plus général de tout *dioïde*. Nous avons noté ces généralisations dans la Remarque 5.7 et dans la Remarque 5.8. De plus, nous avons du formaliser la structure de *treillis complet* pour disposer d'une somme infinie. Celles ci étaient utilisées sans être formellement définies. Enfin, nous avons levé des ambiguïtés sur les instances $\mathcal{F}^+$ et $\mathcal{F}^{\uparrow}$ et sur les résiduations de ces instances. Notamment sur le fait que l'opération $\oslash_{\mathcal{F}^+}$ soit équivalente à $\oslash_{\mathcal{F}^{\uparrow}}$ mais que $\oslash_{\mathcal{F}} \neq \oslash_{\mathcal{F}^+}$.

Le prochain chapitre va consister à utiliser ces résultats pour formaliser une partie des définitions et propriétés données dans la même référence que ce chapitre.

Chapitre 6

Formalisation en Coq des définitions et théorèmes du *Calcul réseau*

Sommaire

6.1 Formalisation des comportements réels **58**
6.1.1 Courbes cumulatives de données 58
6.1.1.1 Présentation et définition 58
6.1.1.2 Addition vers la librairie `bigop` 60
6.1.2 Serveurs 61
6.1.3 Déviations horizontales et verticales 63
6.1.4 Délai et *backlog* 64
6.2 Formalisation des courbes de contraintes **65**
6.2.1 Fonctions usuelles 66
6.2.2 Courbes d'arrivée 67
6.2.3 Courbes de service 69
6.2.3.1 Période de backlog 69
6.2.3.2 Service strict minimum 70
6.2.3.3 Service maximal et gabarit 70
6.3 Formalisation des bornes **71**
6.3.1 Borner le délai et le *backlog* 71
6.3.2 Propagation des contraintes 72
6.3.3 Ordonnancement FIFO 73
6.3.4 Ordonnancement avec priorité statique et préemption 74
6.3.4.1 Définitions et borne sur le délai et *backlog* 75
6.4 Cas d'étude **76**
6.4.1 Preuve papier 77
6.4.2 Preuve Coq 82
6.5 Conclusion **86**

Comme cela a été présenté, le *Calcul réseau* est une méthode d'analyse de réseau embarqué. Elle permet de déterminer des bornes sur des temps de traversée ou des quantités de données maximales contenues par certains points du réseau. Ces bornes sont établies à l'aide de définitions et notions mathématiques. Nous avons vu dans le chapitre précédent la structure algébrique utilisée dans cette théorie, le *dioïde* des fonctions *(min, +)*. Dans l'idée de formaliser le *Calcul réseau*, nous avons logiquement commencé par formaliser cette partie. L'objectif est maintenant de formaliser le *Calcul réseau* afin d'augmenter le niveau de confiance dans cette méthode.

La méthode du *Calcul réseau* consiste à borner, à l'aide de contrats, les comportements réels du réseau. Dans une première section donc, l'idée sera de formaliser ces comportements réels afin de pouvoir établir les contrats. Ces contrats sont des courbes de contraintes et seront formalisés dans une deuxième section. Ensuite, nous utiliserons ces contrats pour formaliser les théorèmes et les preuves associés aux propriétés établies sur les bornes. Cette formalisation sera un gain en confiance dans la validité de ces bornes. Enfin, nous utiliserons les formalisations précédentes sur un cas d'étude. Ce cas d'étude sera un réseau informatique qui est un échantillon des réseaux utilisant le *Calcul réseau* comme méthode d'analyse.

Nous considérons dans ce chapitre que le lecteur est familier avec les notations Coq que nous avons présentées dans les chapitres 4 et 5. Dans ce chapitre, nous donnerons de temps à autres les traductions Coq des énoncés mathématiques que nous décrivons. Toutefois, chaque propriété possède

sa formalisation en Coq : il est fait mention, comme depuis le début de ce manuscrit, du nom Coq associé à la propriété. De plus, nous donnerons pour chaque début de partie le nom du fichier Coq dans lequel les propriétés sont contenues. Ces fichiers peuvent être obtenus à l'adresse suivante :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien`

Il faut au préalable suivre les étapes de compilation des librairies expliqué en Annexe A.

6.1 Formalisation des comportements réels

Nous présentons dans cette première section notre formalisation en Coq des modèles de comportement réel utilisés en *Calcul réseau*. Nous aurons ainsi plusieurs sous parties qui vont successivement présenter les comportements réel suivant : les courbes cumulatives de données, les modèles de serveurs, les déviations horizontales et verticales et enfin les délais et *backlog*.

6.1.1 Courbes cumulatives de données

6.1.1.1 Présentation et définition

Les courbes cumulatives de données sont des fonctions de $\mathbb{R}_+$ dans $\mathbb{R}_+$, croissantes, continues à gauche et qui démarrent en zéro. Elles permettent de modéliser la quantité de données cumulées depuis le démarrage du temps jusqu'à l'instant courant observée à un point donné du réseau. Cette modélisation est donc une représentation d'un comportement réel à un point précis dans le réseau. Cette section porte sur le développement nécessaire pour définir en Coq les courbes cumulatives de données. Les traductions Coq peuvent se trouver dans le fichier `cumulatives_curves.v`.

Dans un premier temps, nous raffinons la Définition 5.11, présentée pour définir l'ensemble $\mathcal{F}^{\uparrow}$.

Définition 6.1 (Clôture croissante à gauche, `left_cont_closure`) Pour $f \in \mathcal{F}$, nous appelons la *clôture croissante à gauche* de f la fonction $g : \mathbb{R}_+ \to \overline{\mathbb{R}}$ définie par

$$g \triangleq t \mapsto \max\left(0, \sup\left\{f(u) \mid u < t\right\}\right).$$

La traduction en Coq donne :

```
Definition left_cont_closure (f : F) : F :=
  fun t ⇒ Order.max 0%:E (ereal_sup [set f u | u in [set u | u < t]]).
```

Une telle définition n'est pas présente dans les présentations classiques du *Calcul réseau*, qui utilisent elles plutôt la clôture croissante (cf. Définition 5.11). Nous donnons en Figure 6.1 un exemple illustrant la différence entre les deux définitions.

Cette nouvelle définition nous permet, comme pour les fonctions de l'ensemble $\mathcal{F}^{\uparrow}$ dans la Définition 5.12, d'obtenir un critère sur la croissance des fonctions. En effet, les courbes cumulatives modélisent une quantité de données au cours du temps, elles sont donc des fonctions croissantes. Ces fonctions cumulatives ont deux autres propriétés : elles démarrent en zéro et sont continues à gauche. Nous allons voir dans les prochains lemmes que la Définition 6.1 permet d'obtenir aussi ces deux propriétés.

Lemme 6.1 (`left_cont_closure0`, `left_cont_closure_ge0`) Pour tout $f \in \mathcal{F}$ et pour g la clôture croissante à gauche de f, nous avons :

$$g(0) = 0 \quad \text{et } 0 \leqslant g(t).$$

Le Lemme 6.1 montre que la clôture croissante à gauche de f démarre bien en zéro et que cette même fonction est une fonction positive. Pour le prochain lemme, nous allons supposer que la fonction est positive.

Lemme 6.2 (`left_cont_ereal_sup`) Pour tout $f \in \mathcal{F}^+$ et t tel que $0 < t$, la clôture croissante stricte de f, noté g, peut se réécrire avec :

$$g(t) = \sup \{f(u) \mid u < t\} .$$

Notons que le $\sup \emptyset = -\infty$. La prochaine définition donne la définition de l'ensemble des fonctions représentant une quantité de donnée cumulée à un instant t pour un endroit précis du réseau.

Définition 6.2 (Courbe cumulative de données, `flow_cc`) Nous appelons les *courbes cumulatives de données* l'ensemble des fonctions $f \in \mathcal{F}^{\uparrow}$ tel que f soit inférieur ou égal à sa clôture croissante stricte et tel que f n'admet pas de valeur infinie. L'ensemble de ces fonctions est noté $\mathcal{C}$.

> **Remarque 6.1** (*Correspondance avec la définition de [BBLC18]*) Nous avons expliqué plus haut que les fonctions cumulatives de données étaient croissantes, démarraient en 0, étaient continues à gauche et s'exprimaient dans $\mathbb{R}_+$. Notre définition est une version plus concise de la Définition 1.1 de [BBLC18]. L'ensemble $\mathcal{C}$ est forcément croissant vu que toute courbe $\mathcal{C} \subseteq \mathcal{F}^{\uparrow}$ et les autres propriétés seront l'objet de lemmes.

La traduction de cette définition en Coq est :

```
Record flow_cc := {
    flow_cc_val :> Fup;
    _ : (flow_cc_val ⩽ left_cont_closure flow_cc_val :> F)%O
        && pselect (∃ f : R+ → R+ , ∀ t, flow_cc_val t = (f t)%:nngnum%:E)
}.
```

Le `Record` décrit trois propriétés : un `flow_cc` est une fonction `flow_cc_val` de type `Fup` (nom donné à l'ensemble $\mathcal{F}^{\uparrow}$).

Cette fonction a comme propriété d'être inférieure à `left_cont_closure flow_cc_val` qui est la traduction de la fonction clôture croissante à gauche de f. Nous utilisons sur cette fonction la relation d'ordre de l'ensemble `F` en utilisant la `Coercion :> F`. Coq ajoute alors automatiquement les fonctions qui transforment un type `Fup` en type `F`.

Nous avons conjoint, avec un *ET* logique noté &&, cette propriété avec une autre propriété. Celle ci est le résultat de la fonction `pselect`. Cette fonction provient de `MathComp`. Elle permet de donner

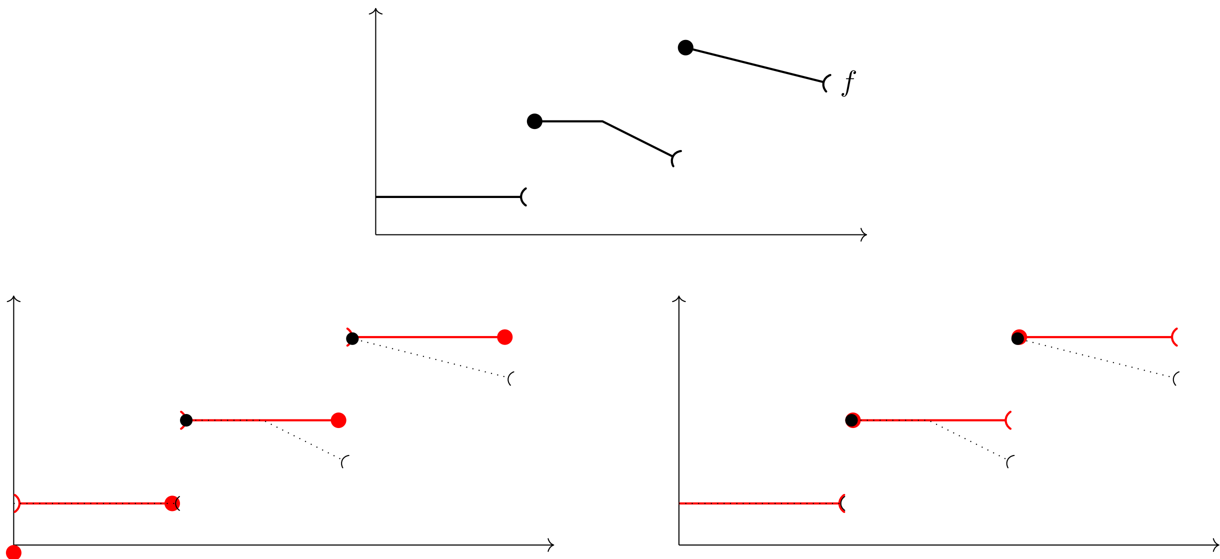


FIGURE 6.1 – Différence entre Clôture croissante à gauche (Définition 6.1) tracée à gauche et Clôture croissante (Définition 5.11) tracée à droite pour une même fonction f.

une propriété supplémentaire que pour chaque fonction `flow_cc_val`, il existe une fonction équivalente `f : R+ → R+`. La notation `%:nngnum` permet de transformer un réel positif en réel.

Comme expliqué dans la Remarque 6.1, nous pouvons retrouvé les propriétés qui font correspondre notre définition avec la Définition 1.1 de [BBLC18]. La propriété que nous allons montrer permet de montrer qu'une fonction $\mathcal{C}$ démarre en 0.

Lemme 6.3 (`left_cont_closure_flow_cc, flow_cc0`) Pour $f \in \mathcal{C}$, la clôture croissante stricte de f est égale à cette fonction en tout point et démarre en 0.

Le prochain lemme nous permet d'établir la propriété de continuité à gauche sur l'ensemble $\mathcal{C}$.

Lemme 6.4 (`flow_cc_left_cont`) Une fonction $f \in \mathcal{C}$ est continue à gauche, c'est à dire que pour tout $x \in \mathbb{R}_+$, nous avons, pour tout $\epsilon > 0$:

$$\exists \eta > 0, \forall y \in \mathbb{R}_+,\ x - \eta \leqslant y \leqslant x \implies f(x) - \epsilon < f(y) \leqslant f(x).$$

Enfin, nous sommes capable de trouver une fonction de $\mathbb{R}_+$ vers $\mathbb{R}_+$ correspondant à une fonction dans $\mathcal{C}$. Cette prochaine propriété nous permettra de savoir que chaque valeur d'une fonction de $\mathcal{C}$ est finie alors que ce n'est pas le cas pour les valeurs de fonctions de $\mathcal{F}^{\uparrow}$ (car $\mathcal{F}^{\uparrow} = \big(\mathbb{R}_+ \to \overline{\mathbb{R}}_+\big)$).

Lemme 6.5 (`fin_flow_ccE`) Pour tout fonction $f \in \mathcal{C}$, il existe $f_R : \mathbb{R}_+ \to \mathbb{R}_+$ tel que :

$$\forall t \in \mathbb{R}_+, f(t) = f_R(t).$$

Cette propriété est possible grâce à la propriété amenée par `pselect` dans la formalisation Coq de la définition des courbes cumulatives.

L'ensemble utilisé dans Définition 6.2 est un sous ensemble de $\mathcal{F}^{\uparrow}$, noté `Fup` en Coq, ce qui nous permet d'utiliser l'instance de *dioïde complet* correspondant à cet ensemble et l'ensemble des propriétés de la théorie des *dioïdes* (cf. Chapitre 5). Nous ne prenons pas le type `F0up`, donné dans la Remarque 5.10, pour justement utiliser ces propriétés surtout que la propriété amenée par le type `F0up` se retrouve avec le Lemme 6.3.

6.1.1.2 Addition vers la librairie `bigop`

Nous allons ensuite définir une addition sur l'ensemble des fonctions $\mathcal{C}$. Cette addition sera ensuite étendu sur une somme d'ensemble avec la bibliothèque `bigop.v` [Ber+08]. Nous ajouterons ensuite un élément neutre à cette définition, il s'agit d'une fonction de $\mathcal{C}$ qui retourne 0 pour toute valeur de $t \in \mathbb{R}_+$. L'objectif de l'utilisation de cette librairie est de pouvoir écrire $\sum_i A_i$ pour $n \in \mathbb{N}$ et $A \in \mathcal{C}^n$ et d'utiliser des propriétés sur les opérateurs d'ensemble.

Définition 6.3 (Addition de $\mathcal{C}$, `flow_cc_plus`**)** Pour f et $g \in \mathcal{C}$, nous définissons l'addition $f + g$ l'opération $\mathcal{C} \to \mathcal{C} \to \mathcal{C}$, donc $f + g \triangleq t \mapsto f(t) + g(t)$.

Définition 6.4 (Fonction 0 de $\mathcal{C}$, `flow_cc_0`**)** La fonction $f_0 \in \mathcal{C}$ est définie par $f_0 \triangleq t \mapsto 0$.

La formalisation en Coq de ces deux définitions a été faite à l'aide d'un `Program Definition`. En effet, il faut pour chaque définition apporter la preuve que la fonction obtenue appartient à l'ensemble $\mathcal{C}$. Comme auparavant dans ce manuscrit, la commande `Program Definition` permet de faire générer le but à prouver par Coq directement.

De plus, nous donnons suffisamment de propriétés pour déclarer une instance de *monoïde* (cf. Définition 5.1). En effet, l'opération d'addition avec l'ensemble $\mathcal{C}$ est un *monoïde*. Cette propriété est similaire à la propriété qui prouve que l'addition de $\mathcal{F}$ avec $\mathcal{F}$ est un *monoïde*. (cf. Lemme 5.49).

Lemme 6.6 (`flow_cc_comoid`) L'ensemble $\mathcal{C}$ avec l'addition définie en 6.3 est un *monoïde commutatif* avec la fonction f_0 (cf. Définition 6.4) comme élément neutre.

Cette propriété nous permet d'utiliser la librairie `bigop.v` [Ber+08].

Pour permettre d'utiliser la `Notation` Coq + pour additionner deux fonctions de $\mathcal{C}$, nous ajoutons une notation à cet effet et un `Scope`. Celui ci est déclaré par :

```
Declare Scope flow_cc_scope.
Delimit Scope flow_cc_scope with C.
Bind Scope flow_cc_scope with flow_cc.

Infix "+ " := flow_cc_plus : flow_cc_scope.
```

qui ajoute donc le `Scope` `%C` pour les opérations sur des courbes cumulatives.

De plus, pour utiliser des sommes Σ sur $\mathcal{C}$, nous ajoutons à ces notations :

```
Notation "\sum_ ( i | P ) F" :=
  (\big[flow_cc_plus/flow_cc_0]_(i | P%B) F%F) : flow_cc_scope.
```

Cette somme s'établit sur des éléments finis comme c'est le cas pour la somme de fonction de $\mathcal{F}^+$, définie dans la Section 5.2.4. Dans cette notation, il faut comprendre que l'élément `F` représente une liste de fonctions de $\mathcal{C}$, que i est un élément de `P` et que `P` est une partie de $\mathbb{N}$.

6.1.2 Serveurs

Nous allons maintenant formaliser la notion de serveur. En *Calcul réseau*, un serveur est un élément du réseau qui traite des données mais ne modifie en rien les quantités de données : la quantité de données entrante est égale à la quantité de données sortante. Pour la modélisation, nous avons déjà expliqué que le serveur utilise une fonction cumulative $A \in \mathcal{C}$ pour les arrivées et $D \in \mathcal{C}$ pour les départs. L'ensemble de cette partie se trouve dans le fichier `servers.v`.

Nous avons pour modéliser un serveur, deux définitions. Cette différence avec la modélisation originale de [BBLC18] sera discutée à la suite des définitions.

Définition 6.5 (Serveur partiel, `partial_server`) Un *serveur partiel* $S_p \subseteq \mathcal{C} \times \mathcal{C}$ est une relation entre A et D, notée $(A, D) \in S_p$, telle que, pour tout A et $D \in \mathcal{C}$, $(A, D) \in S_p$ implique que $D \leqslant A$. Nous notons l'ensemble des *serveurs partiel* avec $\mathcal{S}_p$

Définition 6.6 (Serveur, `server`) Un *serveur* S est un serveur partiel tel que, pour tout $A \in \mathcal{C}$, il existe $D \in \mathcal{C}$ tel que $(A, D) \in S$. Nous notons l'ensemble des *serveurs* avec $\mathcal{S}$.

Ainsi, un serveur partiel est un élément du réseau qui n'est pas obligé d'avoir de sortie pour toutes les entrées. Le relation ajoutée par la Définition 6.6 permet d'avoir une relation totale à gauche. Cette séparation nous permet de créer des serveurs dont la relation totale à gauche ne peut pas être démontrée. Nous reviendrons sur ce point au cours de la déclaration d'un *serveur résiduel contraint.*

Par la suite, nous utiliserons des serveurs qui comportent plusieurs entrées et plusieurs sorties. Il se trouve que ces serveurs sont modélisés avec le même nombre d'entrées et sorties donc nous pouvons avoir une définition de ces serveurs, pour n entrées/sorties.

Définition 6.7 (Serveur multiples entrées/sorties, `nserver`) Pour tout $n \in \mathbb{N}$, un *serveur multiples entrées/sorties* $S_n \subseteq \mathcal{C}^n \times \mathcal{C}^n$ est une relation entre A et D, notée $(A, D) \in S_n$ tel que :

$$\forall A \in \mathcal{C}^n, \exists D \in \mathcal{C}^n, (A, D) \in S_n$$
$$\forall A, D \in \mathcal{C}^n, (A, D) \in S_n \implies \forall i, D_i \leqslant A_i$$

Nous disons alors que $S_n \in \mathcal{S}_n$.

En Coq, nous avons formalisé une notation pour de tels serveurs avec `n.-server`. Avec cette notation, pour déclarer, pour $n \in \mathbb{N}$, un serveur S_n, nous utiliserons :

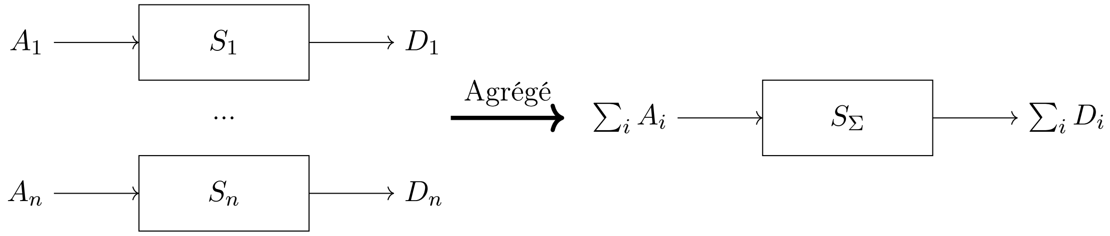


FIGURE 6.2 – Agrégation d'un serveur S_n

```
Section NotationServer.

Variable (n : nat).
Variable (S : n.-server).
```

Manipuler ces serveurs n'est pas toujours obligatoire et nous aurons tendance à agréger ces serveurs à multiples entrées/sorties. Nous donnons une illustration de cette agrégation en Figure 6.2.

Cette agrégation permet donc de sommer les flux entrants et sortants du serveur. La formalisation de cette notion est l'objet de la prochaine définition.

Définition 6.8 (Serveur agrégé, `aggregate_server`**)** Soit $n \in \mathbb{N}^\star$ et $S_n \in \mathcal{S}_n$. Nous appelons *serveur agrégé*, le serveur S_Σ défini par

$$S_\Sigma \triangleq \left\{ (A_\Sigma, D_\Sigma) \in \mathcal{C} \times \mathcal{C} \;\middle|\; \exists A, D \in \mathcal{C}^n, (A, D) \in S_n n \wedge A_\Sigma = \sum_{i=0}^{n-1} A_i \wedge D_\Sigma = \sum_{i=0}^{n-1} D_i \right\}.$$

Pour traduire cette dernière définition, il faut d'abord montrer que S_Σ est un serveur partiel puis ensuite un serveur. Nous avons donc séparé cette définition dans deux `Program Definition` nommés `aggregate_server_cond` et `aggregate_server`. De plus, nous prenons une `Notation` déclarée dans le fichier `cumulatives_curves.v` et que nous avons présentée en fin de Section 6.1.1.

La prochaine définition est la notion de serveur résiduel. En effet, pour analyser l'effet produit par un serveur sur un flux particulier, il faut être capable de séparer les flux au sein d'un serveur avec plusieurs entrées/sorties. Nous avions déjà expliqué cette notion dans la Figure 3.9 au Chapitre 3.

Définition 6.9 (Serveur résiduel, `residual_server`**)** Soit $n \in \mathbb{N}$. Nous appelons *serveur résiduel* d'un serveur S_n pour un flux $i < n$ le serveur S_i définie par

$$S_i \triangleq \{(A_r, D_r) \in \mathcal{C} \times \mathcal{C} \mid \exists A, D, (A, D) \in S_n \wedge A_i = Ar \wedge D_i = D_r\}.$$

Enfin, nous formalisons la notion de concaténation de serveurs montrée au Chapitre 3, dans la Figure 3.7. Cette notion nous permet de mettre les serveurs en série.

Définition 6.10 (Serveur concaténé, `server_concat`**)** Pour S_1 et $S_2 \in \mathcal{S}_p$, nous appelons *serveur concaténé* le serveur $(S_1; S_2)$ définie par :

$$(S_1; S_2) \triangleq \{(A, C) \in \mathcal{C} \times \mathcal{C} \mid \exists B, (A, B) \in S_1 \wedge (B, C) \in S_2\}.$$

Pour le développement en Coq, nous avons ajouté une notation pour utiliser la concaténation. Celle ci est

```
Notation "S1 '';' S2" := (server_concat S1 S2) (at level 100) : Server_scope.
```

et nécessite `at level` 100 puisque elle utilise un nouveau token '; . Cette notation montre aussi le `Scope` crée à cette occasion pour les serveurs.

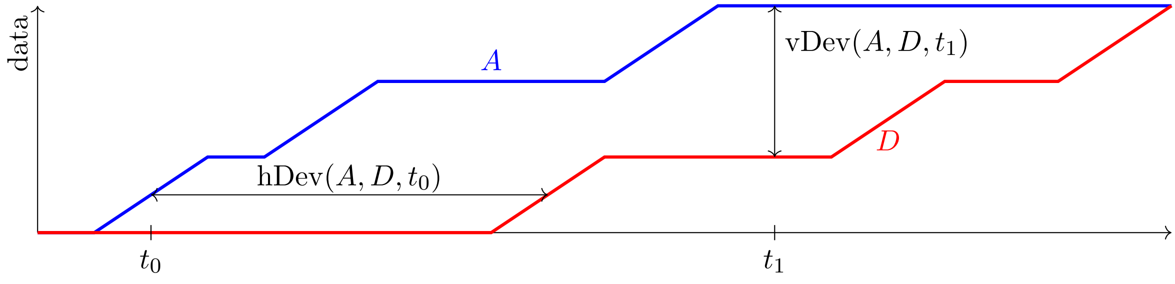


FIGURE 6.3 – Déviations horizontales à t_0 et verticales à t_1

6.1.3 Déviations horizontales et verticales

Nous allons formaliser dans cette partie les notions de déviations horizontales et verticales. Avant d'expliquer le sens de ces notions, nous donnons la définition de ces termes. Celles ci sont proches des définitions données dans [BBLC18]. Les traductions en Coq se trouvent dans le fichier `deviations.v`.

Définition 6.11 (Déviation horizontale, `hDev_at`, `hDev`) Pour toute fonctions f et $g \in \mathcal{F}$ et $t \in \mathbb{R}_+$, nous définissons la *déviation horizontale à t* comme la valeur :

$$\mathrm{hDev}(f, g, t) \triangleq \inf \{d \mid f(t) \leqslant g(t+d)\}.$$

De plus, nous appelons *déviation horizontale* de f et g la fonction hDev : $\mathcal{F} \to \mathcal{F} \to \overline{\mathbb{R}}_+$ définie par :

$$\mathrm{hDev}(f, g) \triangleq \sup \{\mathrm{hDev}(f, g, t) \mid t \in \mathbb{R}_+\}.$$

Définition 6.12 (Déviation verticale, `vDev_at`, `vDev`) Pour toute fonctions f et $g \in \mathcal{F}$ et $t \in \mathbb{R}_+$, nous définissons la *déviation verticale à t* la valeur :

$$\mathrm{vDev}(f, g, t) \triangleq f(t) - g(t)$$

De plus, nous appelons *déviation verticale* de f et g la fonction vDev : $\mathcal{F} \to \mathcal{F} \to \overline{\mathbb{R}}$ définie par :

$$\mathrm{vDev}(f, g) \triangleq \sup \{\mathrm{vDev}(f, g, t) \mid t \in \mathbb{R}_+\}.$$

En *Calcul réseau*, la déviation horizontale représente le délai subit par un message, puisque qu'elle correspond à l'intervalle de temps pour que la quantité de données en sortie soit au moins égale à la quantité de données en entrée. La déviation verticale permet de représenter la quantité de données présente dans un élément du réseau à un instant t. En effet, la différence de quantité entre la quantité d'entrée et la quantité de sortie est la quantité qui n'a pas encore été traitée. Nous donnons une illustration de ces deux définitions dans la Figure 6.3.

Ces définitions sont accompagnées de propriétés. La première que nous montrons permet de réécrire la déviation horizontale à t avec un sup lorsque les fonctions sont des fonctions de l'ensemble $\mathcal{F}^{\uparrow}$, comme c'est le cas des fonctions cumulatives. Il s'agit de la Proposition 5.12 de [BBLC18],

Lemme 6.7 (`sup_horizontal_deviations`) Pour tout f et $g \in \mathcal{F}^{\uparrow}$, nous avons pour tout t :

$$\mathrm{hDev}(f, g, t) = \max(0, \sup \{d \mid g(t+d) < f(t)\}).$$

Nous avons ensuite des propriétés de monotonie des déviations pour des fonctions positives. Ces propriétés sont aussi dans la Proposition 5.12 de [BBLC18].

Lemme 6.8 (`hDev_monotonicity`) Pour tout f, f', g et $g' \in \mathcal{F}^{+}$, tel que $f' \leqslant f$ et $g \leqslant g'$, nous avons :

$$\mathrm{hDev}(f', g') \leqslant \mathrm{hDev}(f, g).$$

Lemme 6.9 (`vDev_monotonicity`) Pour tout f, f', g et $g' \in \mathcal{F}^{+}$, tel que $f' \leqslant f$ et $g \leqslant g'$, nous avons :

$$\mathrm{vDev}(f', g') \leqslant \mathrm{vDev}(f, g).$$

Les traductions en Coq de ces dernier lemmes reprennent les coercions sur `F` pour utiliser l'ordre sur `F` défini dans le chapitre précédent, lors de la déclaration de *dioïde* de l'instance $(\mathcal{F}, \min, *)$. Le Lemme 6.9 est donné par le code suivant :

```
Proposition vDev_monotonicity (f f' g g' : Fplus) :
  (f' ⩽ f :> F)%O → (g ⩽ g' :> F)%O → (vDev f' g' ⩽ vDev f g)%E.
```

Pour rappel, `%O` est le `Scope` pour utiliser les notations de `order.v` de `MathComp`. Cette utilisation est possible puisque l'association entre `F` et cette librairie a été fait au cours du chapitre précédent.

6.1.4 Délai et *backlog*

Comme nous l'avons expliqué plus tôt, les définitions de hDev et vDev correspondent respectivement aux notions de délai et *backlog*. Cette première a déjà été introduite dans la présentation de ce manuscrit. Le délai à t est la durée entre l'instant t d'entrée d'une donnée dans un serveur et l'instant de sortie de cette donnée. Donc, le délai est la plus grande de toutes ces durées. Formellement, cela correspond directement avec la définition de hDev.

Définition 6.13 (Délai, `delay`) Pour A et $D \in \mathcal{F}$, le *délai* est défini par $d(A, D) \triangleq \text{hDev}(A, D)$.

Pour cette définition, nous n'avons pas l'hypothèse que A et D correspondent respectivement à l'entrée et sortie d'un serveur S. Cependant, l'utilisation de cette définition sera souvent accompagnée de l'hypothèse : $(A, D) \in \mathcal{S}$, où $S \in \mathcal{S}$.

Le *backlog* à t correspond à la quantité de données présente dans un serveur à l'instant t, et donc à la différence entre la quantité de données entrée et la quantité de données sortie. Le *backlog* correspond à la quantité maximale présente dans un serveur, et donc au maximum des *backlogs* à t. Cette valeur est déjà représentée par la déviation verticale. Cependant, nous avons besoin pour définir le *backlog* de la notion suivante.

Définition 6.14 (Clôture positive) Pour tout $x \in \overline{\mathbb{R}}$, nous notons $[x]^+ = \max(0, x)$.

> **Remarque 6.2** (*Différence entre* $\max(0, \cdot)$ *et* $[\cdot]^+$) La différence entre la définition de $[\cdot]^+$ et la définition de $\max 0, \cdot$ est l'ensemble sur lequel ces fonctions évoluent. En effet, en Coq, nous aurons $\max(0, \cdot) : \overline{\mathbb{R}} \to \overline{\mathbb{R}}$ alors que $[\cdot]^+ : \overline{\mathbb{R}} \to \overline{\mathbb{R}}_+$. Cette différence nous permet en Coq de ne pas avoir à inclure la preuve que $\max 0, \cdot$ est forcément un nombre positif. Nous reviendrons sur ce détail technique dans l'explication de la fonction Coq `insubd`.

Ainsi nous pouvons définir le *backlog* similairement à la Définition 6.11 mais avec la clôture positive.

Définition 6.15 (`backlog_at`) Pour tout A et $D \in \mathcal{C}$ et pour tout $t \in \mathbb{R}_+$, le *backlog at* t de A et D est la fonction $\mathcal{C} \to \mathcal{C} \to \mathbb{R}_+ \to \mathbb{R}_+$ définie par :

$$b(A, D, t) \triangleq [A(t) - D(t)]^+ .$$

Dans la traduction Coq, la clôture positive est traduite avec une fonction déjà existante.

```
Definition backlog_at (A D : flow_cc) t : R+ :=
  insubd 0%:nng ((A%:fcc t)%:nngnum -(D%:fcc t)%:nngnum).
```

La notation `%:nng` permet de transformer un réel en réel positif. La notation `%:fcc` nous permet d'obtenir la transformation d'une fonction `flow_cc` vers une fonction `R+ → R+` comme cela a été expliqué au cours de la Définition 6.2.

La fonction `insubd` est la fonction qui nous permet de formaliser en Coq la notation $[x]^+$. En réalité, la fonction `insubd` n'est pas un max entre 0 et une valeur x, il s'agit d'une fonction qui retourne un

élément par défaut si x n'appartient par à un certain type. Dans notre cas, pour correspondre à $[x]^+$, il faut que x soit de type {`nonneg R`} (donc $\mathbb{R}_+$) et nous utiliserons la valeur par défaut `0%:nng` (donc 0).

De plus, par choix de conception, il n'est pas nécessaire dans la Définition 6.15 de donner un serveur. Si nous avons l'hypothèse $(A, D) \in S$ avec S un serveur partiel, alors le $[\cdot]^+$ n'est plus nécessaire. Cela fait l'objet d'une propriété.

Lemme 6.10 (`backlog_atE`) Soit $S \in \mathcal{S}_p$. Pour tout A et D tel que $(A, D) \in S$, nous avons :

$$\forall t, b(A, D, t) = A(t) - D(t).$$

qui se traduit en Coq avec :

```
Lemma backlog_atE A D t (S : partial_server) :
  S A D → (backlog_at A D t)%:nngnum = (A%:fcc t)%:nngnum -(D%:fcc t)%:nngnum.
```

dans lequel nous avons donc enlevé la fonction `insubd`. Ce lemme permettra alors de réécrire la fonction `backlog_at` à chaque fois que nous aurons l'hypothèse $S A D$ qui est la formalisation Coq de $(A, D) \in S$.

Comme pour la déviation verticale, le *backlog* est le pire *backlog* pour tout t.

Définition 6.16 (Backlog, `backlog`) Pour tout A et $D \in \mathcal{C}$, nous définissons le *backlog* de A et D la fonction $\mathcal{C} \to \mathcal{C} \to \overline{\mathbb{R}}_+$ définie par :

$$b(A, D) \triangleq \sup \{b(A, D, t) \mid t \in \mathbb{R}_+\}.$$

Dans cette définition, il est possible de montrer que $b(A, D)$, pour toute fonction cumulative A et D, est un nombre positif grâce à la Définition 6.15.

Remarque 6.3 (*Différence entre* vDev *et backlog*) La fonction vDev est de type $\mathcal{F} \to \mathcal{F} \to \overline{\mathbb{R}}$ puisque la définition et la formalisation en Coq de la fonction `vDev_at` est de type :

* Reponses *

```
vDev_at : F → F → {nonneg R} → \bar R
```

La fonction de *backlog* est différente puisque son type est $\mathcal{C} \to \mathcal{C} \to \overline{\mathbb{R}}_+$. Nous pouvons certifier un nombre positif en sortie puisque `backlog_at` est de type :

* Reponses *

```
backlog_at : flow_cc → flow_cc → {nonneg R} → {nonneg R}
```

Cela conclut l'ensemble des développements que nous avons effectués pour définir les comportements réels d'un réseau avec le point de vue du *Calcul réseau*.

6.2 Formalisation des courbes de contraintes

L'objectif de cette partie est de formaliser les contraintes des comportements réels comme il est d'usage en *Calcul réseau* (cf. Introduction au *Calcul réseau*, Chapitre 3). En effet, les résultats obtenus sur cette théorie n'utilisent pas les modèles de comportement réels mais plutôt des contrats établis sur ceux ci. Ces contrats permettent de représenter un ensemble de comportements avec un seul contrat. Ainsi, ils permettent de manipuler un ensemble de courbes cumulatives avec une seule courbe.

Nous allons ajouter donc des contrats sur les notions de la section précédente. Pour cela, nous donnerons en premier quelques fonctions simples, ainsi que quelques propriétés, sur des fonctions qui

seront utilisées dans les contrats. Ensuite, nous suivrons le même ordre que la dernière section, nous donnerons les contrats pour les fonctions cumulatives de données puis nous donnerons les contrats sur les serveurs.

6.2.1 Fonctions usuelles

Nous souhaitons dans cette partie donner les définitions des fonctions qui seront utilisées pour les courbes de contrats. Les traductions des définitions et propriétés associées se trouvent dans le fichier `usual_functions.v`. La première d'entre elles est une fonction qui vaut 0 jusqu'à un certain point puis $+\infty$ ensuite.

Définition 6.17 (Fonction δ, `delta`) Pour tout $d \in \overline{\mathbb{R}}$, la fonction $\delta \in \mathcal{F}^{\uparrow}$ est définie par :

$$\delta_d \triangleq t \mapsto \begin{cases} 0 & \text{si } t \leqslant d, \\ +\infty & \text{sinon.} \end{cases}$$

Une illustration de cette fonction est incluse dans l'Exemple 6.1.

La traduction en Coq nécessite de montrer que cette fonction est un `Fup`. Pour cela, nous utilisons deux constructeurs : le premier est `Build_Fup` pour avoir un `Fup`. Ce constructeur attend en entrée un fonction de type `Fplus` c'est pourquoi nous devons prendre un second constructeur : `Build_Fplus`. Ensuite, l'utilisation de `Program Definition` suivi des deux `Next Obligation` permet de bien séparer la preuve en deux parties : une première pour montrer que $\delta \in \mathcal{F}^{+}$ et la seconde pour montrer que $\delta \in \mathcal{F}^{\uparrow}$.

La propriété évidente qui vient immédiatement est celle qui démontre la valeur de $\delta_{+\infty}$.

Lemme 6.11 (`delta_infty`) Nous avons $\delta_{+\infty} = f_0^{\uparrow}$ (cf. Définition 5.14).

La Définition 6.17 est souvent reliée à la notion de délai pur. Un délai pur est un délai apporté par un serveur à toutes les données, quelle que soit leur date d'arrivée. Nous reviendrons dessus plus tard dans ce chapitre. La formalisation de cette notion est apportée par la prochaine propriété. Elle réutilise le produit de convolution défini à la Définition 5.2.

Lemme 6.12 (`delta_prop_conv`) Pour tout $f \in \mathcal{F}^{\uparrow}$ et d et $t \in \mathbb{R}_{+}$, nous avons :

$$(f * \delta_d)(t) = f([t-d]^{+}).$$

Nous réutilisons la fonction `insubd`, présenté dans la Définition 6.15 pour traduire en Coq ce lemme.

Exemple 6.1. Exemple sur le Lemme 6.12

Une illustration de ce lemme est donnée en Figure 6.4. Sur celle ci, nous prenons une fonction $f \in \mathcal{F}^{\uparrow}$. Nous faisons la convolution avec une fonction δ_d, avec un d positif.

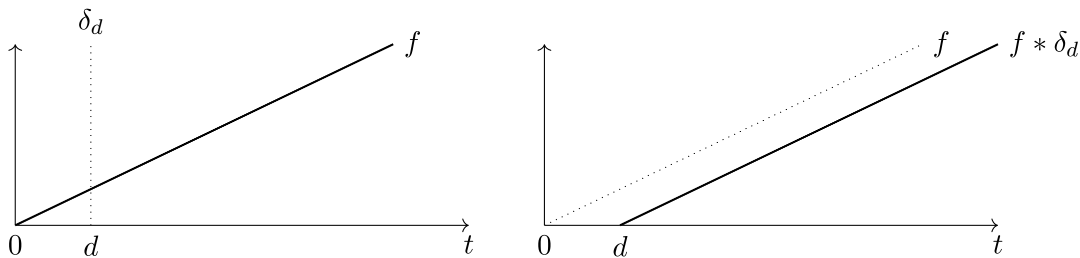


FIGURE 6.4 – Illustration Lemme 6.4

La notion de délai pur apparaît ici : considérons que f est une fonction cumulative, alors la convolution avec δ_d aurait pour conséquence de retarder chaque donnée de f d'une durée d puisqu'elle seront en sortie, sur $f * \delta_d$, après une durée d.

Nous allons introduire une autre fonction du *Calcul réseau*. C'est une fonction affine positive croissante.

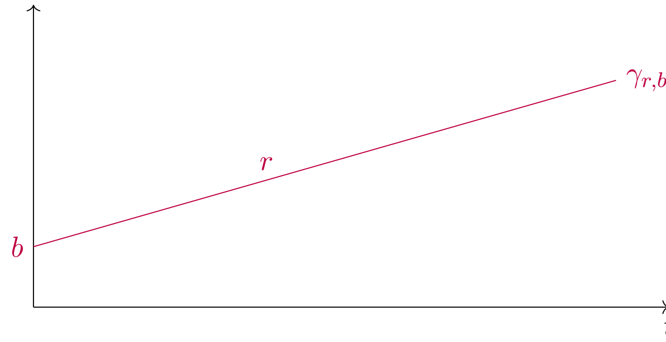


FIGURE 6.5 – Exemple d'une fonction γ

Définition 6.18 (Fonction γ, `gamma_h`) Pour tout r et $b \in \mathbb{R}_+$, la fonction $\gamma \in \mathcal{F}^{\uparrow}$ est définie par :

$$\gamma_{r,b} \triangleq t \mapsto rt + b.$$

> **Remarque 6.4** (*Différence de la définition* γ) La Définition 6.18 de la fonction γ est différente de [BBLC18], à la Définition 3.1. Dans cette définition originale, la fonction *token-bucket* est égale à celle de notre définition, sauf en 0 où elle vaut 0.

Comme pour la fonction δ, la traduction en Coq fait appel aux constructeurs `Build_Fup` et `Build_Fplus` pour démontrer que `gamma_h` est de type `Fup`.

6.2.2 Courbes d'arrivée

Les courbes d'arrivée permettent d'établir une contrainte sur une courbe cumulative de donnée. Elles sont donc des contraintes à un point du réseau. Elles sont définies de la manière suivante et traduites en Coq dans le fichier `arrival_curves.v`.

Définition 6.19 (Arrivée maximale, `is_maximal_arrival`) Pour tout $A \in \mathcal{C}$, nous appelons courbe d'*arrivée maximale de* A la fonction $\alpha \in \mathcal{F}$ telle que, pour tout t et $d \in \mathbb{R}_+$:

$$A(t+d) - A(t) \leqslant \alpha(d).$$

Nous disons alors que la courbe α est une courbe d'arrivée pour A ou alors que A est contrainte par la courbe d'arrivée α. Nous donnons un exemple en Figure 6.6. Dans cette figure, nous donnons une courbe d'arrivée $A \in \mathcal{C}$ et une courbe d'arrivée α qui est une contrainte de A.

L'objectif de ces courbes d'arrivée est de contraindre un fonctionnement réel. Nous l'utiliserons pour déclarer les propriétés importantes du *Calcul réseau*. Les preuves seront aussi associées à cette définition donc nous donnons plusieurs propriétés.

La première d'entre elles consiste à décrire une équivalence entre la définition de courbe d'arrivée, donnée ci dessus, et le produit de convolution décrit dans le Chapitre précédent, à la Définition 5.2.

Lemme 6.13 (`maximal_arrival_alt_def`) Pour toute fonction $A \in \mathcal{C}$, la fonction $\alpha \in \mathcal{F}$ est une courbe d'arrivée pour A si et seulement si :

$$A \leqslant A * \alpha.$$

La traduction en Coq consiste à reprendre le produit de convolution décrit dans la loi multiplicative du *dioïde* (cf. Description du fichier `dioid.v`). L'équivalence en Coq nous permettra de réécrire directement cette propriété dans les preuves et ainsi utiliser tout les résultats du *dioïde* montrés au Chapitre 5.

Les prochaines propriétés vont nous permettre de dériver des courbes d'arrivée à partir d'autres courbes. La première permet de déterminer que pour deux courbes α et α' qui sont des courbes d'arrivée

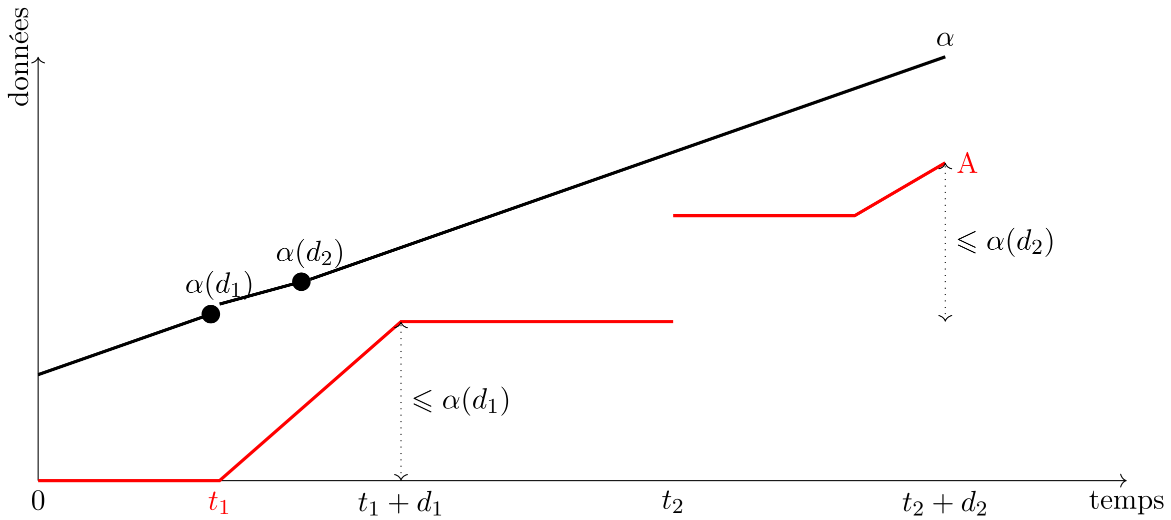


FIGURE 6.6 – Exemple de courbe d'arrivée α pour une courbe cumulative A. Sur cette courbe, nous voyons qu'en tout intervalle, A progresse moins que la valeur de α. Par exemple, les points $\alpha(d_1)$ et $\alpha(d_2)$ sont respectivement supérieur à $A(t_1 + d_1) - A(t_1)$ et $A(t_2 + d_2) - A(t_2)$. Nous voyons aussi sur cette figure qu'il est localement possible que la pente de A soit plus forte que la pente de α.

pour une même courbe A, alors le minimum entre α et α' est aussi une courbe d'arrivée pour cette courbe. La seconde permet de conclure sur un majorant d'une courbe d'arrivée. Enfin, la dernière propriété montre que la fonction valant 0 en 0 et $+\infty$ pour tout $t > 0$ est une courbe d'arrivée pour toute courbe cumulative, puisque toute fonction de $\mathcal{C}$ démarre en 0.

Lemme 6.14 (`maximal_arrival_F_min`) Pour tout $A \in \mathcal{C}$ et α et α' tel que α et $\alpha' \in \mathcal{F}$ sont des courbes d'arrivée pour A, alors $\min(\alpha, \alpha')$ est une courbe d'arrivée pour A.

Lemme 6.15 (`maximal_arrival_le`) Pour tout $A \in \mathcal{C}$ et α et $\alpha' \in \mathcal{F}$ tel que α est une courbe d'arrivée pour A, si $\alpha \leqslant \alpha'$ alors α' est une courbe d'arrivée pour A.

Lemme 6.16 (`maximal_arrival_delta_0`) Pour tout $A \in \mathcal{C}$, la fonction δ_0 est une courbe d'arrivée pour la courbe A.

La traduction de ces preuves en Coq fait appel à la théorie des *dioïdes*. Donc, nous faisons souvent appel à la réécriture `F_dioidE` qui permet de traduire une opération ou un opérateur de l'instance $\mathcal{F}$ vers une opération du *dioïde* (cf. Le *dioïde* des fonctions *(min, +)*, Section 5.2).

Nous avons aussi des propriétés qui utilisent la déconvolution du *dioïde* des fonctions *(min, +)*. Elles nous permettent de déterminer que la déconvolution par elle même d'une fonction cumulative A quelconque est forcément une courbe d'arrivée pour A et qu'il s'agit de la plus petite des courbes d'arrivée possibles. Pour rappel, nous avons une notation de la déconvolution pour chaque instance de *dioïde*. Dans les deux prochaines propriétés, nous utiliserons la déconvolution sur $\mathcal{F}$.

Lemme 6.17 (`maximal_arrival_deconv1`) Pour tout $A \in \mathcal{C}$, la fonction défini par $A \oslash_{\mathcal{F}} A$ est une courbe d'arrivée pour A.

Lemme 6.18 (`maximal_arrival_deconv2`) Pour tout $A \in \mathcal{C}$ et $\alpha \in \mathcal{F}$ tel que α soit une courbe d'arrivée de A, nous avons : $A \oslash_{\mathcal{F}} A \leqslant \alpha$.

Donc la déconvolution d'une fonction par elle même est une courbe d'arrivée pour toute fonction cumulative avec le Lemme 6.17. De plus, il s'agit de la plus petite courbe d'arrivée pour toute courbe cumulative d'arrivée avec le Lemme 6.18.

La traduction en Coq est assez ressemblante pour les deux lemmes. Dans le second, nous devons utiliser une `Coercion` vers `F` pour ne pas utiliser l'ordre du *dioïde*. Nous avons donc la traduction suivante :

```
Lemma maximal_arrival_deconv2 A alpha :
  is_maximal_arrival A alpha → ((A : F) / A)%D ⩽ alpha :> F.
```

où `is_maximal_arrival A alpha` est bien l'hypothèse du Lemme 6.18. Il nous faut faire deux `Coercion` : une pour utiliser l'ordre sur `F` comme expliqué au dessus. La seconde `Coercion` est pour utiliser la déconvolution avec la notation / sur l'instance de *dioïde* `F`. Cela est nécessaire vu que `A` est du type `flow_cc` (le type pour $\mathcal{C}$). Coq ajoute alors ici la fonction qui permet de transformer un `flow_cc` en `F` et agit similairement sur l'autre opérande de l'opération. Ainsi, nous avons besoin de noter `A : F` une seule fois.

Lemme 6.19 (`maximal_arrival_sum_flow_cc`) Pour tout $n \in \mathbb{N}^\star$, $A \in \mathcal{C}^n$ et $\alpha \in \mathcal{F}^n$, si pour tout $i \in [\![0, n-1]\!]$, α_i est une courbe d'arrivée pour A_i, alors la courbe $\sum_i \alpha_i$ est une courbe d'arrivée pour $\sum_i A_i$.

Pour traduire ce lemme, nous reprenons la notation utilisée dans la Définition 6.8. Celle ci nous permet d'avoir $\sum_i A_i \in \mathcal{C}$.

6.2.3 Courbes de service

Nous allons dans cette partie introduire les courbes de contrats pour les serveurs. Cette contrainte sera ensuite manipulée avec les courbes d'arrivée pour donner des propriétés sur les temps de traversée. La traduction de cette partie se trouve dans le fichier `services.v`.

La première courbe de service est une courbe qui va borner le minimum de données que peut servir un serveur. Cette contrainte est donc un minorant pour la sortie du serveur.

Définition 6.20 (Service minimum *(min, +)* pour *A*, `is_min_service_for`**)** Pour tout $S \subseteq \mathcal{C} \times \mathcal{C}$ et $A \in \mathcal{C}$, la courbe $\beta \in \mathcal{F}$ est un *service minimum de* S *pour* A si, pour tout $D \in \mathcal{C}$,

$$(A, D) \in S \implies A * \beta \leqslant D.$$

Cette définition s'applique pour une entrée A particulière. Nous avons séparé cette partie pour une utilisation particulière de la définition. Nous reviendrons dessus ultérieurement. En généralisant la définition précédente à toute entrée, nous obtenons la définition de la courbe de service minimum pour toute entrée.

Définition 6.21 (Service minimum *(min, +)*, `is_min_service`**)** Pour tout $S \subseteq \mathcal{C} \times \mathcal{C}$, la courbe $\beta \in \mathcal{F}$ est un *service minimum de* S si, pour tout $A \in \mathcal{C}$, la courbe β est un service minimum de S pour A.

La traduction en Coq de ces définitions reprend la convolution *(min, +)* en Coq comme nous l'avons déjà vu plus haut dans ce chapitre.

Nous souhaitons maintenant introduire et formaliser la notion de service strict minimum. Pour cela, nous devons introduire la notion de période de *backlog*, notion utilisée dans la définition de service strict minimum.

6.2.3.1 Période de backlog

Une période de *backlog* est un intervalle pendant lequel un serveur est occupé par des données. Pour donner une idée, cela correspond pour un serveur $S \in \mathcal{S}$ et A et $D \in \mathcal{C}$ tel que $(A, D) \in S$ aux intervalles de temps pendant lesquels $D < A$. La traduction en Coq de cette partie se situe dans le fichier `backlogged_period.v`.

Définition 6.22 (Période de *backlog*, `backlogged_period`**)** Pour tout A et $D \in \mathcal{F}$ et pour tout u et $v \in \mathbb{R}_+$, l'intervalle $]u; v]$ est une *période de backlog* pour A et D si $u \leqslant v$ et :

$$\forall x \in \left]u; v\right], D(x) < A(x).$$

Nous souhaitons pouvoir manipuler les conditions sur la période de *backlog*. Ainsi, nous serons capables, pendant les preuves Coq, de modifier certaines conditions. La première d'entre elle est l'intervalle même de la période. Ainsi, si un intervalle I' est inclus dans un autre intervalle I et que cet intervalle est une période de *backlog*, alors I' est aussi une période de *backlog*.

Lemme 6.20 (`backlogged_period_sub_interval`) Pour tout A et D et pour tout u, u', v et $v' \in \mathbb{R}_+$ tel que $u \leqslant u'$, $v' \leqslant v$ et $u' \leqslant v'$, si l'intervalle $]u; v]$ est une période de *backlog* pour A et D alors $]u'; v']$ est une période de *backlog* pour A et D.

La période de *backlog* sera utilisée sur des serveurs agrégés. Donc, nous serons amenés à manipuler des sommes de courbes cumulatives. Ainsi, si pour un ensemble P de courbes cumulatives, un intervalle est une période de *backlog*, alors cet intervalle est aussi une période de *backlog* pour tout sous ensemble de P.

Lemme 6.21 (`backlogged_period_weaken_cond`) Soit $n \in \mathbb{N}$ et $S \in \mathcal{S}^n$. Pour tout A et $D \in \mathcal{C}^n$ tel que $(A, D) \in S$, pour tout P et $P' \subseteq \mathbb{N}$ tel que $P \subseteq P'$ et pour tout u et $v \in \mathbb{R}_+$, si $]u; v]$ est une période de *backlog* pour $\sum_{i \in P} A_i$ et $\sum_{i \in P} D_i$ alors $]u; v]$ est une période de *backlog* pour $\sum_{i \in P'} A_i$ et $\sum_{i \in P'} D_i$.

6.2.3.2 Service strict minimum

En reprenant les périodes de *backlog* introduites précédemment, le *Calcul réseau* utilise des courbes de service minimum strict. Cette notion est très similaire au service minimum *(min, +)* défini plus tôt dans ce chapitre. Le développement de cette notion se trouve dans le fichier `services.v`.

Définition 6.23 (Service minimum strict, `is_strict_min_service`) Soit $S \subseteq \mathcal{C} \times \mathcal{C}$. La courbe $\beta_s \in \mathcal{F}$ est un *service minimum strict* de S si, pour tout A et D tel que $(A, D) \in S$ et pour tout u et v tel que $]u, v]$ est une période de *backlog* pour A et D, nous avons :

$$\beta_s([v - u]^+) \leqslant D(v) - D(u).$$

Nous avons besoin de noter dans la formalisation Coq $[v - u]^+$ pour avoir cet élément dans $\mathbb{R}_+$. En effet, il est clair que $v - u$ est forcément positif avec la définition de la période de *backlog* (cf. Définition 6.22). Cependant, nous devons spécifier à Coq que cet élément est dans $\mathbb{R}_+$ et donc utiliser $[\cdot]^+$.

> **Remarque 6.5** (*Service minimal (min, +) et strict*) De manière générale, nous utiliserons le service minimal *(min, +)* plutôt que le service minimal strict. Donc, nous noterons *service minimal* pour faire référence au service minimal *(min, +)* et spécifierons la notion stricte quand cela sera nécessaire.

6.2.3.3 Service maximal et gabarit

Pour donner une borne maximale sur la sortie d'un serveur, le *Calcul réseau* utilise un service maximal. Nous utiliserons cette notion pour démontrer la propriété de propagation. Le développement de cette partie se trouve dans le fichier `services.v`.

Définition 6.24 (Service maximal, `is_max_service`) Pour tout $S \subseteq \mathcal{C} \times \mathcal{C}$, la courbe β_{max} est un *service maximal* de S si, pour tout A et $D \in \mathcal{C}$:

$$(A, D) \in S \implies D \leqslant A * \beta_{max}.$$

Il aurait été possible de faire comme dans pour le service minimal (cf. Définition 6.21) avec une définition intermédiaire : `is_maximal_service_for`. Une telle définition n'a pas été nécessaire dans la suite du développement donc nous n'avons pas procédé de cette manière.

Donc, avec cette définition, une fonction qui vaut 0 en 0 puis $+\infty$ pour $t > 0$ est un service maximal pour tout serveur. Une telle définition est la fonction $f_{\bar{1}}$ (cf. Définition 5.4).

Lemme 6.22 (`Fup_zero_is_max_service, Fup_one_is_max_service`) Pour tout $S \in \mathcal{S}_p$, les fonctions $f_{\bar{0}}$ et $f_{\bar{1}}$ sont des services maximal de S.

Il est à noter que cette propriété ne s'applique que si $S \in \mathcal{S}_p$ alors que la Définition 6.24 prend $S \subseteq \mathcal{C} \times \mathcal{C}$. En effet, nous devons utiliser la propriété $D \leqslant A$ pour A et D respectivement une entrée et une sortie du serveur pour prouver ce lemme. Dans la traduction, ce lemme se nomme `server_le`.

Pour spécifier le gabarit que peut prendre la sortie d'un serveur, le *Calcul réseau* utilise la définition suivante.

Définition 6.25 (`shaper_server`) Pour tout $S \subseteq \mathcal{C} \times \mathcal{C}$, la fonction σ est un *gabarit de* S si, pour tout A et $D \in \mathcal{C}$:

$$(A, D) \in S \implies D \leqslant D * \sigma.$$

De même, la fonction qui vaut $+\infty$ pour tout t est un gabarit pour tout serveur S. Cette fonction est la fonction $f_{\bar{0}}$ (cf Définition 5.3).

Lemme 6.23 (`Fup_zero_is_shaper_server, Fup_one_is_shaper_server`) Pour tout S, les fonctions $f_{\bar{0}}$ et $f_{\bar{1}}$ sont des gabarits de S.

Cette propriété vient conclure la définition et la traduction des notions de courbe de service du *Calcul réseau*. Par ailleurs, nous avons donné les définitions des notions du *Calcul réseau* suffisantes pour formaliser des bornes. La suite de ce chapitre va alors consister à donner de telles bornes pour analyser la traversée d'un réseau.

6.3 Formalisation des bornes pour la traversée d'un réseau

Maintenant que les notions du *Calcul réseau* ont été définies et formalisées, nous pouvons donner les traductions de propriétés du *Calcul réseau*. Ces propriétés portent sur l'analyse de réseaux temps réel en donnant des bornes sur les délais et sur les quantités de données.

6.3.1 Borner le délai et le *backlog*

Le délai que subissent les données dans un serveur est donnée par la définition du délai. La quantité maximale de données qui sera présente dans un serveur est donnée par le *backlog*. Respectivement, ces notions ont été définies dans les définitions 6.13 et 6.16. Pour établir une borne sur ces notions pour un serveur, le *Calcul réseau* utilise la prochaine proposition. Celle ci utilise les contrats sur l'arrivée de donnée et sur le service minimal que peut rendre un serveur.

Proposition 6.1 (`delay_bounds, backlog_bounds`) Pour tout $S \in \mathcal{S}_p$, α et $\beta \in \mathcal{F}^{\uparrow}$, A et $D \in \mathcal{C}$ tel que $(A, D) \in S$, si α est une courbe d'arrivée pour A et si β est un service minimal, pour S alors

$$d(A, D) \leqslant \mathrm{hDev}(\alpha, \beta) \quad \wedge \quad b(A, D) \leqslant \mathrm{vDev}(\alpha, \beta).$$

La preuve de cette propriété est donnée dans le fichier `servives.v`. Cette proposition est très importante, elle permet d'établir un majorant au pire délai et au pire *backlog* à partir des courbes d'arrivée et de service.

La prochaine propriété consiste à borner la sortie d'un serveur. Nous avons ici un majorant et un minorant qui permettent de considérer un serveur comme une gigue. Celle ci est comprise entre le plus petit délai et le plus grand délai.

Proposition 6.2 (`Server_as_a_jitter_min`) Pour tout $S \in \mathcal{S}_p$, A et $D \in \mathcal{C}$ tel que $(A, D) \in S$, et pour tout d_m tel que $d_m \leqslant \inf \{d(A, D, t) \mid t \in \mathbb{R}_+\}$, nous avons : $D \leqslant A * \delta(d_m)$.

Proposition 6.3 (`Server_as_a_jitter_max`) Pour tout $S \in \mathcal{S}_p$, A et $D \in \mathcal{C}$ tel que $(A, D) \in S$, et pour tout d_M tel que $d(A, D) \leqslant d_M$, nous avons : $A * \delta(d_M) \leqslant D$.

En utilisant cette borne maximale, nous sommes capable avec la Proposition 6.1 de trouver un service minimal pour un serveur.

Proposition 6.4 (`Service_for_a_jitter`) Pour tout $S \in \mathcal{S}_p$, $A \in \mathcal{C}$, α et $\beta \in \mathcal{F}^{\uparrow}$ tel que α soit une courbe d'arrivée pour A et β soit un service minimal de S, la courbe $\delta_{\mathrm{hDev}(\alpha,\beta)}$ est un service minimum pour A de S.

Nous utilisons dans cette proposition la Définition 6.20. L'idée est maintenant de pouvoir concaténer aussi des serveurs. Pour cela, nous avons la prochaine proposition.

Proposition 6.5 (`server_concat_conv`) Pour tout S_1 et $S_2 \in \mathcal{S}_p$ et pour tout β_1 et $\beta_2 \in \mathcal{F}^{\uparrow}$ tel que pour $i = \{1, 2\}$, β_i est une courbe de service de S_i, la courbe $\beta_1 * \beta_2$ est une courbe de service pour le serveur concaténé $(S_1; S_2)$.

Ainsi, nous sommes capables de déterminer une courbe de service pour deux serveur mis en concaténation. Cette proposition vient conclure les formalisations des propriétés sur les borne de délais et *backlog*. Nous réutiliserons ces résultats dans l'étude de cas, en fin de ce chapitre.

6.3.2 Propagation des contraintes

Pour analyser un réseau complet, il faut être capable de connaître l'état en sortie de chaque serveur. En *Calcul réseau*, cela revient à déterminer un contrat en sortie de celui ci. Dans [BBLC18], de telles propriétés sont présentées dans le Théorème 5.3 et le Corollaire 5.3. Nous formalisons donc cette partie de la théorie dans le fichier `propagation.v`. Nous avons ici une première proposition qui permet de déterminer une courbe d'arrivée pour la sortie d'un serveur.

Proposition 6.6 (`output_arrival_curve`) Soit $S \subseteq \mathcal{C} \times \mathcal{C}$ et A et D tel que $(A, D) \in S$. Pour tout $\alpha \in \mathcal{F}$ tel que α est une courbe d'arrivée maximale pour A, pour tout β_m et $\beta_M \in \mathcal{F}$ tel que β_m est un service minimal de S et β_M est un service maximal de S et pour tout $\sigma \in \mathcal{F}^{\uparrow}$ tel que σ est un gabarit de S, la fonction définie par

$$\min\left((\alpha * \beta_M) \oslash_{\mathcal{F}} \beta_m, \sigma\right)$$

est une courbe d'arrivée maximale pour la sortie D.

Cette proposition demande beaucoup d'hypothèses. En effet, le service du serveur est contraint par trois hypothèses. En *Calcul réseau*, il est plus fréquent de n'utiliser qu'une courbe de service minimale qui est déterminée à partir du plus petit service que peut rendre un serveur. Ainsi, la contrainte en sortie d'un serveur est simplifiée. Nous la donnons dans la prochaine proposition.

Proposition 6.7 (`output_arrival_curve_mp_F`) Soit $S \subseteq \mathcal{C} \times \mathcal{C}$ et A et D tel que $(A, D) \in S$. Pour tout $\alpha \in \mathcal{F}$ tel que α est une courbe d'arrivée pour A et pour tout $\beta \in \mathcal{F}$ tel que β est un service minimal de S, la fonction définie par :

$$\alpha \oslash_{\mathcal{F}} \beta$$

est une courbe d'arrivée maximale pour la sortie D.

Proposition 6.8 (`output_arrival_curve_mp_Fplus`) Soit $S \in \mathcal{S}_p$ et A et D tel que $(A, D) \in S$. Pour tout $\alpha \in \mathcal{F}^{+}$ tel que α soit une courbe d'arrivée pour A et pour tout $\beta \in \mathcal{F}^{+}$ tel que β soit un service minimal de S, la fonction

$$\alpha \oslash_{\mathcal{F}^+} \beta$$

est une courbe d'arrivée maximale pour la sortie D.

Proposition 6.9 (`output_arrival_curve_mp_Fup`) Soit $S \in \mathcal{S}_p$ et A et D tel que $(A, D) \in S$. Pour tout $\alpha \in \mathcal{F}^{\uparrow}$ tel que α soit une courbe d'arrivée pour A et pour tout $\beta \in \mathcal{F}^{\uparrow}$ tel que β soit un service minimal de S, la fonction

$$\alpha \oslash_{\mathcal{F}^{\uparrow}} \beta$$

est une courbe d'arrivée maximale pour la sortie D.

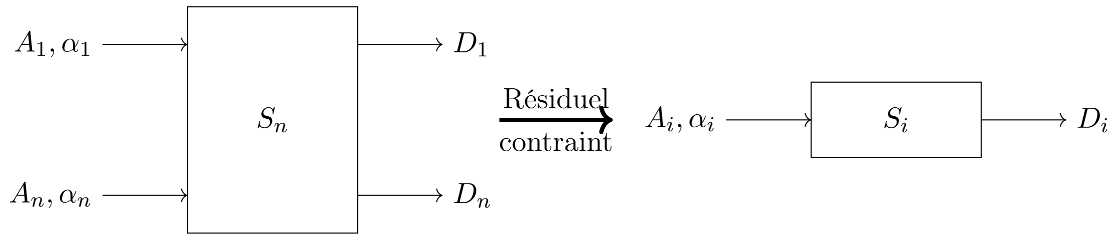


FIGURE 6.7 – Serveur résiduel contraint

Nous obtenons la Proposition 6.9 à partir de la Proposition 6.8 puisque les déconvolutions de $\mathcal{F}^+$ et $\mathcal{F}^\uparrow$ sont équivalentes (cf. Lemme 5.57). Les résultats de ces deux propositions sont néanmoins différentes de la Propositions 6.7 puisque, comme nous l'avons dit dans la Section 5.2.6, nous n'avons pas d'égalité entre $\oslash_\mathcal{F}$ et $\oslash_{\mathcal{F}^+}$.

De plus, par déduction de la Section 5.2.6, le résultat de la propagation des contraintes obtenu avec l'opération $\oslash_\mathcal{F}$ est inférieur aux résultats de $\oslash_{\mathcal{F}^+}$ et $\oslash_{\mathcal{F}^\uparrow}$. Il se trouve cependant qu'une telle différence n'aura pas tant d'importance. En effet, l'exemple pris dans la Section 5.2.6, à la Figure 5.3, n'est pas réaliste puisque la courbe de service utilisée, la fonction g dans cette figure, ne correspond à aucun serveur. De plus, si $D \in \mathcal{C}$ admet α comme courbe d'arrivée, elle admet aussi $[\alpha]^+$.

Ces propositions supplémentaires viennent conclure les propriétés que nous avons formalisées sur la propagation des contraintes dans un réseau.

6.3.3 Ordonnancement FIFO

Nous cherchons à formaliser des bornes sur les délais et sur les *backlogs* en utilisant le *Calcul réseau* pour des serveurs avec plusieurs entrées et plusieurs sorties. Ces bornes sont dépendantes de l'ordonnancement du serveur comme cela est expliqué dans le Chapitre 7 de [BBLC18]. L'objectif est donc d'être capable de formaliser une borne pour chaque politique d'ordonnancement.

Une première politique d'ordonnancement que nous souhaitons formaliser est un ordonnancement *FIFO*. Dans cette politique, le premier message arrivant dans un serveur est le premier à être traité. Nous allons donc dans cette partie donner la définition formelle de cette politique. Puis, nous donnerons dans une proposition une borne sur le délai pour un serveur avec cette politique.

La définition de la politique donnée dans la Définition 7.7 de [BBLC18] fait mention d'un service résiduel contraint par une courbe d'arrivée. Cette nouvelle notion est le service que le serveur à n entrées/sorties peut offrir à chaque flux, comme un serveur résiduel (Définition 6.9), mais en imposant que chaque flux doit être contraint par une courbe d'arrivée. Afin de définir la politique *FIFO*, nous formalisons cette notion de service résiduel contraint. La traduction de cette définition se trouve dans le fichier `servers.v`.

Définition 6.26 (Serveur résiduel contraint, `residual_server_constr`) Soit $n \in \mathbb{N}$, $i < n$, $S \in S_n$ et $\alpha \in \mathcal{F}^n$. Nous appelons *serveur résiduel de S pour i contraint par α* , le serveur partiel S_R défini par :

$$S_R \triangleq \{(A_r, D_r) \in \mathcal{C} \times \mathcal{C} | \exists A, D, (A, D) \in S \wedge A_r = A_i \wedge D_r = D_i \wedge (\forall j, A_j \leqslant A_j * \alpha_j)\}.$$

Nous donnons une illustration explicative de cette définition dans la Figure 6.7. Cette définition est donc un ajout par rapport à [BBLC18], elle n'était pas formalisée dans une définition mathématique mais uniquement suggérée en anglais.

> **Remarque 6.6** (*Serveur partiel dans la Définition 6.9*) Le serveur résiduel contraint par une courbe d'arrivée n'est pas un serveur total, il est un serveur partiel (cf. Définition 6.5).

Nous avons tout les éléments pour définir formellement la politique *FIFO*. Nous avons traduis en Coq l'ensemble du reste de cette partie dans le fichier `fifo.v`.

Définition 6.27 (Politique *FIFO*, `FIFO_service_policy`) Soit $n \in \mathbb{N}^\star$. Le serveur $S \in \mathcal{S}_n$ est un serveur avec une *politique FIFO* si, pour tout A et D tel que $(A, D) \in S$, pour tout i et $j \in [\![0, n-1]\!]$ et pour tout t et $u \in \mathbb{R}_+$ nous avons :

$$A_i(u) < D_i(t) \implies A_j(u) \leqslant D_j(t).$$

Ainsi, si une donnée d'un flux i est arrivée à un temps u et est partie à un temps t (c'est à dire $A_i(u) < D_i(t)$), alors c'est aussi le cas pour tout les autres flux arrivées avant u. Nous savons que un serveur de $\mathcal{S}_1$ respecte forcément cette politique.

Lemme 6.24 (`FIFO_one_server`) Un serveur $\mathcal{S}_1$ est un serveur avec une politique *FIFO*.

De plus, dans un serveur *FIFO*, si l'agrégation des arrivées est inférieure à l'agrégation des départs alors toutes les arrivées sont inférieures aux départs et réciproquement.

Lemme 6.25 (`FIFO_aggregate_server`) Soit $n \in \mathbb{N}^\star$ et $S \in \mathcal{S}_n$ avec une politique *FIFO*. Pour tout A et D tel que $(A, D) \in S$, nous avons pour tout u et t :

$$\sum_{i=0}^{n-1} A_i(u) \leqslant \sum_{i=0}^{n-1} D_i(t) \iff \left(\forall i \in [\![0, n-1]\!], A_i(u) \leqslant D_i(t)\right).$$

Nous donnons maintenant une borne sur la traversée d'un serveur avec une politique *FIFO*. Cette borne est formée avec une courbe de service utilisant la fonction δ (cf. Définition 6.17).

Proposition 6.10 (`FIFO_delay`) Soient $n \in \mathbb{N}^\star$ et $S \in \mathcal{S}_n$ avec une politique *FIFO*. Pour $\alpha \in \mathcal{F}^{\uparrow n}$ et $\beta \in \mathcal{F}_0^{\uparrow}$ tel que β est une courbe de service minimum pour le serveur agrégé S, pour tout $i < n$, la courbe

$$\delta_d \text{ avec } d \triangleq \text{hDev}\left(\sum_{j=0}^{n-1} \alpha_j, \beta\right)$$

est une courbe de service pour le serveur résiduel pour i contraint par α.

Avec cette proposition, nous avons donc une courbe de service qui s'applique à un flux i en particulier. Pour obtenir une borne, il nous faut utiliser ensuite la Proposition 6.1. Ainsi, nous savons que le délai et le *backlog* maximal d'un serveur *FIFO* est plus petit que la déviation horizontale et respectivement verticale entre la courbe d'arrivée α et la courbe de service δ_d définie ci dessus.

Cette proposition est la conclusion des propriétés et définitions que nous avons formalisées sur l'ordonnancement *FIFO*. Nous utiliserons ces résultats sur un cas d'étude, plus loin dans ce chapitre.

6.3.4 Ordonnancement avec priorité statique et préemption

L'objectif maintenant est de formaliser une borne avec une autre politique d'ordonnancement. Cette politique est une politique avec une priorité statique. Cette priorité associe à chaque flux une priorité fixe : un message prioritaire peut interrompre l'émission d'un message moins prioritaire. Nous allons donc formaliser cette définition en nous basant sur la Définition 7.8 de [BBLC18]. Ensuite, nous formaliserons en Coq le théorème permettant de déterminer le service résiduel offert par un serveur suivant cette politique. Cette formalisation se basera sur le Théorème 7.6 de [BBLC18].

Cependant, avant de définir la politique, nous allons donner une notion et quelques propriétés qui sont nécessaires dans la preuve de ce théorème principal. Cette notion est une extension de la période de *backlog* qui a été définie dans la Section 6.2.3 à la Définition 6.22.

La période de *backlog*, quand celle ci existe, possède forcément un instant de départ. Il est défini par la prochaine définition. Nous ferons le lien avec la période de *backlog* dans un prochain lemme. Nous avons mis cette définition et les propriétés associées dans le fichier `backlogged_period.v`.

Définition 6.28 (Début de période de *backlog*, `start`) Pour tout A et $D \in \mathcal{F}$ et pour tout $t \in \mathbb{R}_+$, nous appelons $\text{start}_{(A,D)}(t)$ le *début de période de backlog* défini par :

$$\text{start}_{(A,D)}(t) \triangleq \max\left(0, \sup\left\{u \mid u \leqslant t \wedge A(u) = D(u)\right\}\right).$$

Le début de la période de *backlog* est utilisé dans des propriétés du *Calcul réseau*. Pour en faciliter les preuves, nous avons alors plusieurs propriétés. Une première propriété porte sur la monotonie de start. Une autre propriété permettra de montrer que deux fonctions au point start de ces fonctions sont égales. Enfin, nous verrons que le start(t) pour tout t est forcément avant t.

Lemme 6.26 (`start_non_decr`) Pour tout A et $D \in \mathcal{C}$ et pour tout x et $y \in \mathbb{R}_+$, nous avons

$$x \leqslant y \implies \text{start}_{(A,D)}(x) \leqslant \text{start}_{(A,D)}(y).$$

Lemme 6.27 (`start_eq`) Nous avons pour tout A et $D \in \mathcal{C}$ pour tout t :

$$A(\text{start}_{(A,D)}(t)) = D(\text{start}_{(A,D)}(t)).$$

Lemme 6.28 (`start_le`) Nous avons pour tout A et $D \in \mathcal{F}$ pour tout t : $\text{start}_{(A,D)}(t) \leqslant t$.

Afin de lier la période de *backlog* avec start, qui est le début de cette période, nous donnons la prochaine propriété.

Lemme 6.29 (`start_t_is_a_backlogged_period`) Pour tout $S \in \mathcal{S}_p$, A et $D \in \mathcal{C}$ et $t \in \mathbb{R}_+$, si nous avons $(A, D) \in S$ alors $\left]\text{start}_{(A,D)}(t); t\right]$ est une période de *backlog* pour A et D.

Ce lemme conclut sur les propriétés que nous utiliserons sur la période de *backlog*.

6.3.4.1 Définitions et borne sur le délai et *backlog*

Nous souhaitons maintenant définir la politique par priorité statique. Cette notion sera ensuite utilisée dans une proposition qui déterminera un service résiduel pour un serveur avec cette politique. Le développement de cette partie se trouve dans le fichier `static_priority.v`. Les prochain lemmes sont nécessaires pour manipuler les agrégations de flux et les périodes de *backlog*.

Lemme 6.30 (`sum_flow_cc_equiv`) Pour tout $n \in \mathbb{N}$, A, $D \in \mathcal{C}$ et $S \in S_n$ tel que $(A, D) \in S$, nous avons :

$$(\forall i < n, A_i = D_i) \iff \left(\sum_{i=0}^{n-1} A_i = \sum_{i=0}^{n-1} D_i\right).$$

La formalisation en Coq de ce dernier lemme demande une trentaine de lignes de code. En effet, il est nécessaire d'utiliser la librairie `bigop` ([Ber+08]) pour montrer que l'égalité entre la somme des arrivées et la somme des départs implique que chaque arrivées est égale au départ correspondant.

Lemme 6.31 (`backlogged_period_server_eqv`) Soit $n \in \mathbb{N}^\star$ et $S \in S_n$, pour tout A et D tel que $(A, D) \in S$ et pour tout s et $t \in \mathbb{R}_+$, l'intervalle $]s; t]$ est une période de *backlog* pour $\sum_{i=0}^{n-1} A_i$ et $\sum_{i=0}^{n-1} D_i$ si et seulement si

$$\forall u \in \left]s; t\right], \exists i \in [\![0, n-1]\!], D_i(u) < A_i(u).$$

Lemme 6.32 (`backlogged_period_aggregate`) Soit $n \in \mathbb{N}^\star$ et $S \in S_n$, pour tout A et D tel que $(A, D) \in S$, pour tout s et $t \in \mathbb{R}_+$ et pour tout $i \in [\![0, n-1]\!]$, si $]s; t]$ est une période de *backlog* pour A_i et D_i alors $]s; t]$ est une période de *backlog* pour $\sum_{i=0}^{n-1} A_i$ et $\sum_{i=0}^{n-1} D_i$.

Ces propriétés seront utilisées dans la preuve du théorème principal que nous énoncerons dans cette partie.

Remarque 6.7 (*Généralisation des conditions*) Des propriétés intermédiaires ont été démontrée pour montrer les lemmes 6.30, 6.31 et 6.32. Ces lemmes prennent l'hypothèse supplémentaire : dans chaque lemme, i appartient à P qui est une partie de $[\![0, n-1]\!]$.

Avec ces dernières propriétés, nous avons assez de matériels pour démontrer le théorème principal de cette partie. Nous donnons dans la prochaine définition la formalisation de la politique d'ordonnancement étudiée dans cette partie.

Définition 6.29 (Préemption priorité statique, `preemptive_SP`) Soit $n \in \mathbb{N}^\star$ et $S \in \mathcal{S}_n$, pour une priorité $\Gamma : [\![0, n-1]\!] \to \mathbb{N}$, le serveur S est préemptif à priorité statique Γ si, pour tout A et D tel que $(A, D) \in S$, pour tout $i \in [\![0, n-1]\!]$ et $s, t \in \mathbb{R}_+$ tel que $]s; t]$ est une période de *backlog* pour

$$\sum_{\substack{j=0 \\ \Gamma(j)<\Gamma(i)}}^{n-1} A_j \quad \text{et} \quad \sum_{\substack{j=0 \\ \Gamma(j)<\Gamma(i)}}^{n-1} D_j$$

nous avons : $D_i(s) = D_i(t)$.

Nous parlons de préemption puisque que le serveur peut interrompre les messages en cours d'émission pour laisser la priorité à un autre message. Il existe des cas sans préemption mais nous ne traitons pas de ce type de serveur dans ce manuscrit.

Dans cette dernière définition, nous n'évoquons pas le cas où les priorités sont égales. Nous éviterons un tel cas en ajoutant dans les propriétés concernant un serveur à priorité statique, l'hypothèse que la fonction de priorité Γ est injective. Quand on n'est pas dans ce cas, on peut toutefois s'y ramener en agrégeant les flux de même priorité avec une politique FIFO.

Une telle définition nous permet de poser l'hypothèse qu'un serveur S est un serveur avec une politique préemptive à priorité statique. Avec une telle hypothèse, nous sommes capables comme pour la politique *FIFO* d'obtenir une courbe de service résiduelle contrainte par une courbe d'arrivée.

Proposition 6.11 (`SP_residual_service_curve`) Soit $n \in \mathbb{N}^\star$ et une priorité injective Γ (i.e pour tout x, y, si $\Gamma(x) = \Gamma(y)$, alors $x = y$), soit un serveur $S \in \mathcal{S}_n$ préemptif avec une priorité statique Γ, pour tout β tel que β est une courbe de service minimum stricte pour le serveur agrégé S, pour tout $\alpha \in \mathcal{F}^n$ et pour tout $i \in [\![0; n-1]\!]$, la courbe

$$\left(t \mapsto \max\left(0, \beta(t) - \sum_{\substack{j=0 \\ \Gamma(j)<\Gamma(i)}}^{n-1} \alpha_j\right)\right)_\uparrow$$

est une courbe de service minimale stricte pour le serveur résiduel contraint au flux i par α.

Une telle courbe est une fonction de $\mathcal{F}$. Donc, nous n'avons pas besoin d'utiliser $[\cdot]^+$ dans cette définition puisque il nous suffit d'avoir des éléments dans $\overline{\mathbb{R}}$ et non dans $\overline{\mathbb{R}}_+$.

Comme pour la Proposition 6.10, nous pouvons utiliser ce résultat pour borner le délai et le *backlog* d'un serveur avec cette politique en utilisant la Proposition 6.1.

Cette sous section conclut la formalisation des propriétés et définitions du *Calcul réseau*. Cette formalisation s'est basée sur [BBLC18] et nous avons vu que quelques différences avaient été nécessaires. Nous en discuterons en conclusion de ce chapitre. Nous allons maintenant appliquer les propriétés à un cas d'étude. Ce cas d'étude va reprendre différents éléments traduits en Coq ici donc nous pourrons traduire directement ce cas d'étude dans ce langage.

6.4 Cas d'étude

L'objectif de cette section est d'appliquer les résultats de ce chapitre à un cas d'étude. Ce cas d'étude a été utilisé dans les travaux que nous avons présentés dans [RBR19]. Nous donnerons ici une explication plus détaillée. De plus, ce cas d'étude doit se rapprocher d'un cas réel sans toutefois s'éloigner des hypothèses établies dans la théorie du chapitre précédent. Par exemple, nous ne pouvons pas analyser de dépendances cycliques car nous n'avons pas formalisé sa résolution.

Les hypothèses que nous pouvons prendre sont celles qui ont été formalisées précédemment. Nous pouvons prendre des serveurs traitant plusieurs flux, nous pouvons concaténer des serveurs. Le cas d'étude que nous prenons est schématisé en Figure 6.8. L'ensemble de la formalisation de cette partie se trouve dans le fichier `examples/case_study/arbitrary_flows.v`.

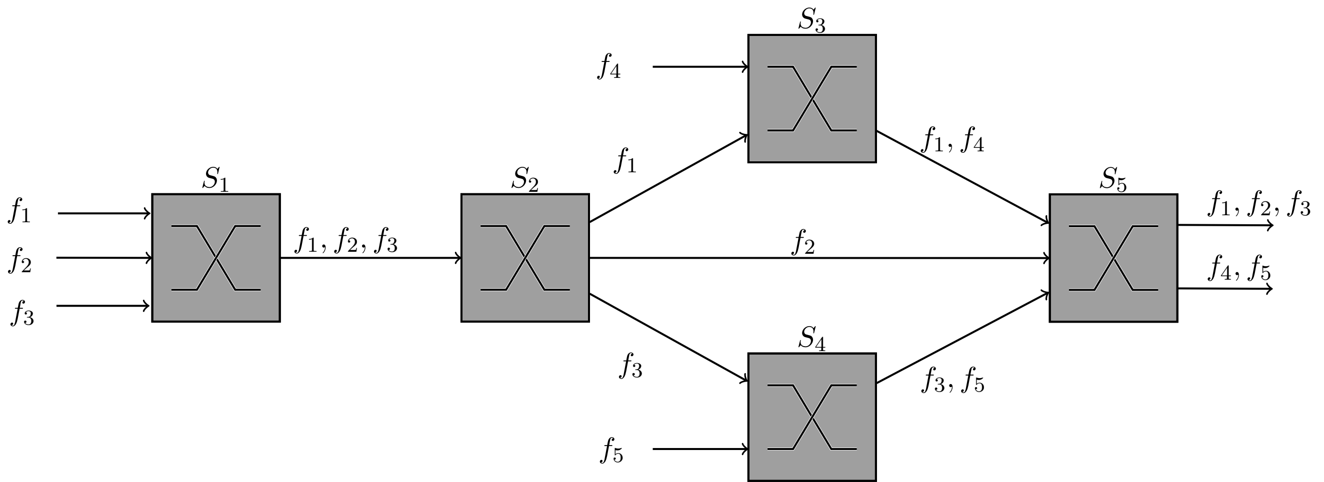


FIGURE 6.8 – Réseau du cas d'étude. Sur cette figure, nous pouvons voir que notre cas d'étude contient 5 serveurs : S_1, S_2, S_3, S_4 et S_5. Ils sont traversés par 5 flux : f_1, f_2, f_3, f_4 et f_5. Les serveur S_1 et S_2 sont traversés par les flux f_1, f_2 et f_3 Le serveur S_3 est traversé par les flux f_1 et f_4 et le serveur S_4 est traversé par les flux f_3 et f_5. Enfin, le serveur S_5 est traversé par les 5 flux.

Dans cet exemple, nous posons quelques hypothèses. Les serveurs ont tous la même politique d'ordonnancement : une politique *FIFO* comme définie dans la Définition 6.27. De plus, nous avons aussi l'hypothèse que les débits en entrée sont bornés et les serveurs peuvent assurer un service minimal de traitements des données. L'objectif de ce cas d'étude est de déterminer des bornes sur les temps de traversée des messages pour chacun de ces 5 flux.

Nous allons dans un premier temps analyser les temps de traversé de ce réseau en utilisant les notions et propriétés du *Calcul réseau* que nous avons définies et expliquées dans ce chapitre. Puis, nous formaliserons en Coq cette analyse. Nous utiliserons alors pour cela une partie de notre formalisation Coq des notions et propriétés du *Calcul réseau*.

6.4.1 Preuve papier

Pour analyser le réseau en utilisant le *Calcul réseau*, nous allons dans un premier temps supposer que chaque flux f_i dispose d'une courbe cumulative $F_i \in \mathcal{C}$. Cette courbe cumulative représente le cumul d'entrée pour chaque flux à son entrée du réseau : donc à l'entrée de S_1 pour F_1, F_2 et F_3, à l'entrée de S_3 pour F_4 et à l'entrée de S_4 pour F_5. Chacune de ces courbes cumulatives est alors contrainte par une courbe d'arrivée $\alpha_i \in \mathcal{F}^{\uparrow}$. Nous supposerons que chaque courbe d'arrivée respecte la Définition 6.19. Donc, par exemple, pour F_1, nous avons l'hypothèse : α_1 est une courbe d'arrivée pour F_1. Dans cet exemple, nous supposerons que chaque flux à son entrée dans le réseau est constant et qu'il a un débit d'un message de 8kBit toutes les 400ms.

Nous allons ensuite supposer que chaque serveur S_i est contraint par une courbe de service minimal β_i respectant la Définition 6.21. Ainsi, pour le serveur S_1 par exemple, nous avons l'hypothèse : la courbe β_1 est un service minimum pour le serveur agrégé S_1. Par exemple, nous supposons que chaque serveur a un débit de $100^{\text{kBit}}/\text{s}$, ce service sera offert à la somme des arrivées du serveur.

Ainsi, avec ces valeurs numériques, nous obtenons la Figure 6.9. Dans notre exemple, les courbes tracées sont applicables à tous les serveurs et tous les flux.

Dans notre cas d'étude, la sortie du serveur S_1 est regroupée sur un seul flux pour représenter un multiplexage des flux f_1, f_2 et f_3 dans le serveur S_1. Pour analyser un tel multiplexage avec le *Calcul réseau*, il nous faudra faire évoluer notre représentation du réseau. Pour bien comprendre comment fonctionne cette partie du réseau, nous donnons la Figure 6.10. Dans cette figure, nous avons explicité les comportements des serveurs S_1 et S_2 avec les flux qui les traversent.

Donc, du point de vue du *Calcul réseau*, il nous faut traiter le serveur S_2 comme trois serveurs avec un unique flux. Cela nous permettra d'avoir l'hypothèse que chaque serveur a le même nombre d'entrées que de sorties. Cette modification sera aussi apportée au serveur S_5 puisqu'il dispose aussi

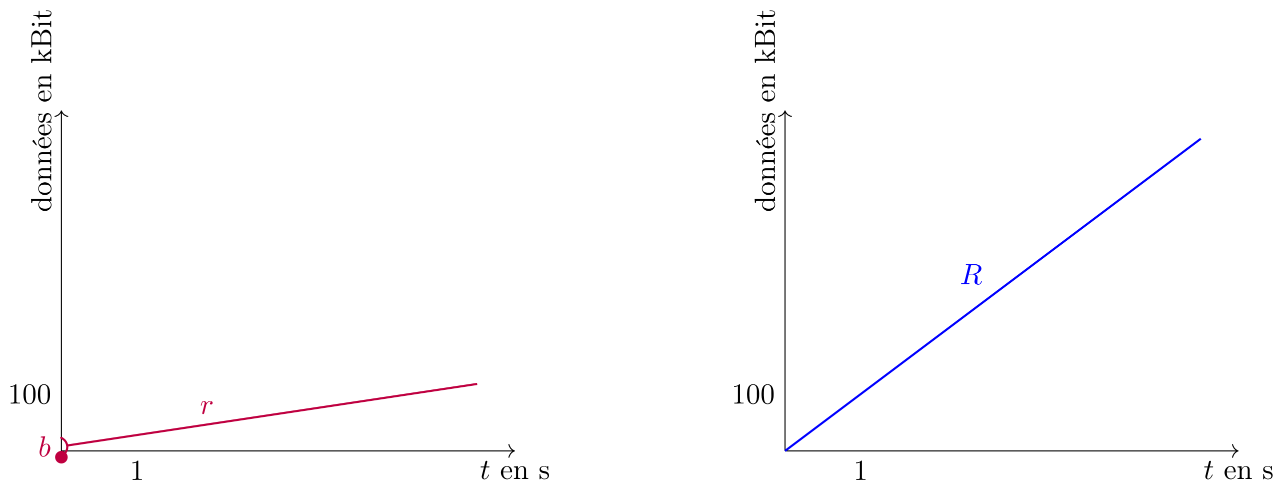


FIGURE 6.9 – Courbes d'arrivée et de service du cas d'étude. Nous avons r = 20k et b = 8k pour la courbe d'arrivée et R = 100k pour la courbe de service.

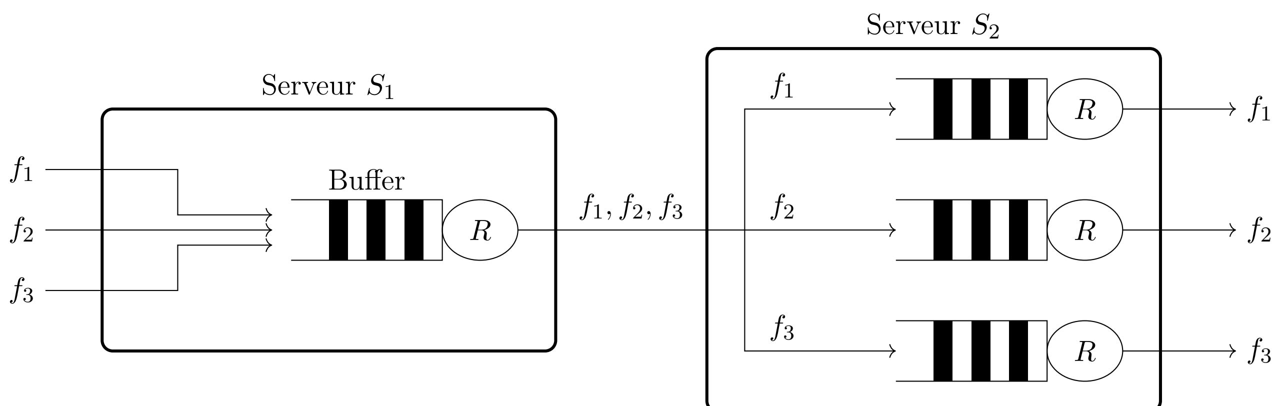


FIGURE 6.10 – Serveurs S_1 et S_2. Le serveur S_1 est équipé d'un buffer recevant les messages des trois flux f_1, f_2 et f_3. Le serveur S_2 est équipé de trois buffers traitant les données des flux de manière séparée.

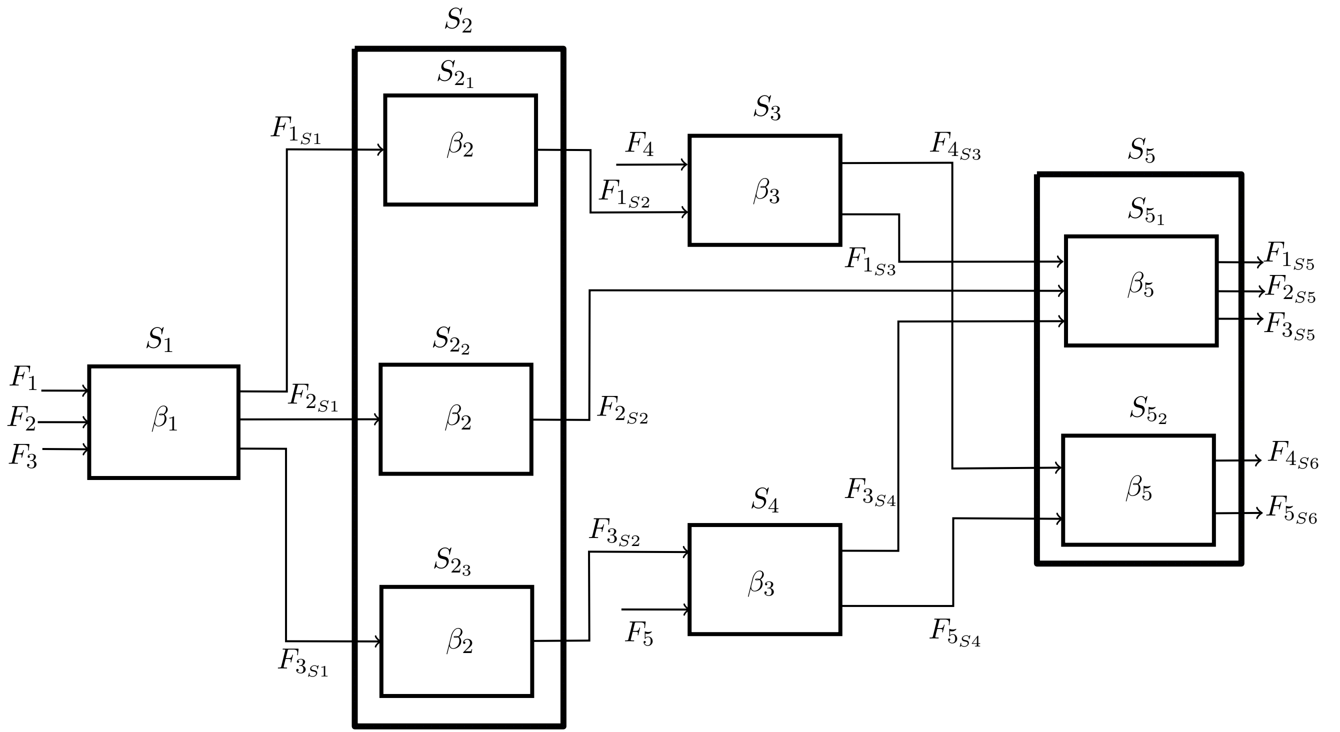


FIGURE 6.11 – Réseau détaillé du cas d'étude. Le serveur S_2 est séparé en trois serveurs S_{2_1}, S_{2_2} et S_{2_3} avec pour chacun une seule entrée et une seule sortie. C'est aussi le cas pour S_5 qui est séparé en deux avec S_{5_1} (trois entrées et trois sorties) et S_{5_2} (deux entrées et deux sorties).

de séparation et multiplexage de flux. Ainsi, nous mettons à jour le modèle premièrement donné pour qu'il puisse être analysé à l'aide du *Calcul réseau*. Nous donnons en Figure 6.11 un tel modèle. Il contient les nouvelles représentations des serveurs et les noms des courbes cumulatives en chaque point du réseau. La nomenclature choisie est : si une courbe cumulative F_i est une entrée d'un serveur S_j, alors la courbe cumulative de sortie est $F_{i_{S_j}}$.

Pour expliquer comment nous procédons pour obtenir le temps de traversée de chaque flux, nous proposons d'expliquer comment le *Calcul réseau* permet de déterminer une borne sur le temps de traversé du flux f_1. Avant tout, il convient de déterminer une expression de ce temps de traversé. Nous appelons d_1 le délai maximum subi par les messages du flux f_1 entre leur entrée et leurs sortie du réseau, donc, $d_1 \triangleq d(F_1, F_{1_S}5)$, d'après la Figure 6.11 et la Définition 6.13.

Le premier serveur traversé par ce flux est S_1. Il en résulte une courbe cumulative de sortie, déjà notée $F_{1_{S1}}$. Nous avons donc l'hypothèse :

$$(F_1, F_{1_{S1}}) \in S_1 \tag{6.1}$$

Puisque S_1 est un serveur *FIFO*, nous pouvons utiliser la Proposition 6.10 pour établir une courbe de service résiduelle pour f_1 que nous noterons β_{S1} :

$$\beta_{S1} \triangleq \delta_d \text{ avec } d \triangleq \text{hDev}(\alpha_1 + \alpha_2 + \alpha_3, \beta_1) \tag{6.2}$$

Cela nous permet de démontrer la propriété suivante.

Lemme 6.33 (`min_residual_service_S1`) Pour tout $i \in [\![1, 3]\!]$, la courbe β_{S1} est une courbe de service pour le serveur résiduel de S_1 contraint pour i pour la courbe α_i.

Le *Calcul réseau* utilise ensuite la Proposition 6.9 pour établir une contrainte sur la sortie de ce premier serveur. La nomenclature pour les contraintes de sorties est similaire à celle des courbes cumulatives de sorties de serveur. Connaissant le résultat de la proposition, nous posons :

$$\alpha'_{1_{S1}} \triangleq \alpha_1 \oslash_{\mathcal{F}^\uparrow} \beta_{S1}$$

De plus, si $\alpha'_{1_{S1}}$ est une courbe d'arrivée pour $F_{1_{S1}}$, alors $min(\delta_0, \alpha'_{1_{S1}})$ l'est aussi grâce aux lemmes 6.14 et 6.16 et elle est plus précise que $\alpha'_{1_{S1}}$. Nous définissons alors :

$$\alpha_{1_{S1}} \triangleq \min\left(\alpha'_{1_{S1}}, \delta_0\right) \tag{6.3}$$

Ainsi, pour $t = 0$, $\alpha_{1_{S1}}(t) = \delta_0$ puis, pour $t > 0$, $\alpha_{1_{S1}}(t) = \alpha_{1_{S1}}$. Nous appliquons ensuite ce résultat aux autre courbes d'arrivée des départs de S_1.

$$\alpha_{2_{S1}} \triangleq \min\left(\alpha_2 \oslash_{\mathcal{F}\uparrow} \beta_{S1}, \delta_0\right) \tag{6.4}$$

$$\alpha_{3_{S1}} \triangleq \min\left(\alpha_3 \oslash_{\mathcal{F}\uparrow} \beta_{S1}, \delta_0\right). \tag{6.5}$$

qui nous permet de démontrer la propriété suivante :

Lemme 6.34 (`maximal_departure_S1`) Pour tout $i \in [\![1, 3]\!]$, les courbes $\alpha_{i_{S1}}$ sont des courbes d'arrivée maximale pour les fonctions cumulatives $F_{i_{S1}}$.

Démonstration (du Lemme 6.34) Montrons que $\alpha_{1_{S1}} = \min\left(\alpha_1 \oslash_{\mathcal{F}\uparrow} \beta_{S1}, \delta_0\right)$ est une courbe d'arrivée pour $F_{1_{S1}}$. Nous savons grâce au Lemme 6.16 que δ_0 est une courbe d'arrivée pour la fonction $F_{1_{S1}}$. En utilisant la Proposition 6.9, nous savons que $\alpha_1 \oslash_{\mathcal{F}\uparrow} \beta_{S1}$ est une courbe d'arrivée pour $F_{1_{S1}}$. Finalement, en utilisant le Lemme 6.14, nous savons que le minimum entre deux fonctions qui sont des courbes d'arrivée est aussi une courbe d'arrivée. Donc $\min\left(\alpha_1 \oslash_{\mathcal{F}\uparrow} \beta_{S1}, \delta_0\right)$ est une courbe d'arrivée pour $F_{1_{S1}}$. La preuve est similaire pour $\alpha_{2_{S1}}$ et $\alpha_{3_{S1}}$. □

Il faut alors continuer cette opération avec l'ensemble des serveurs traversés par f_1. Le serveur suivant est S_{2_1} donc nous avons l'hypothèse que : $(F_{1_{S1}}, F_{1_{S2}}) \in S_{2_1}$ et en posant :

$$\beta_{S2_1} \triangleq \delta_{\mathrm{hDev}(\alpha_{1_{S1}}, \beta_2)} \quad \text{et} \quad \alpha_{1_{S2}} \triangleq \min\left(\alpha_{1_{S1}} \oslash_{\mathcal{F}\uparrow} \beta_{S2_1}, \delta_0\right) \tag{6.6}$$

nous avons :

Lemme 6.35 (`min_residual_service_S2_1, maximal_departure_S2_1`) La courbe β_{S2_1} est une courbe minimale de service pour le serveur résiduel de S_{2_1} contraint au flux $F_{2_{S1}}$ par la courbe $\alpha_{1_{S1}}$. La courbe $\alpha_{1_{S2}}$ est une courbe d'arrivée pour la fonction $F_{1_{S2}}$.

Le serveur suivant est le serveur S_3. Pour celui ci, nous prenons α_4 pour une courbe d'arrivée de F_4 et la précédente courbe $\alpha_{1_{S2}}$. Nous définissons ainsi :

$$\beta_{S3} \triangleq \delta_{\mathrm{hDev}(\alpha_{1_{S2}} + \alpha_4, \beta_3)} \quad \text{et} \quad \alpha_{1_{S3}} \triangleq \min\left(\alpha_{1_{S2}} \oslash_{\mathcal{F}\uparrow} \beta_{S3}, \delta_0\right) \tag{6.7}$$

La propriété attachée est similaire au Lemme 6.35.

Lemme 6.36 (`min_residual_service_S3, maximal_departure_S3`) La courbe β_{S3} est une courbe minimale de service du serveur résiduel de S_3 contraint aux flux $\{F_{1_{S2}}, F_3\}$ par les courbes $\{\alpha_{1_{S2}}, \alpha_4\}$. La courbe $\alpha_{1_{S3}}$ est une courbe d'arrivée pour la fonction $F_{1_{S3}}$.

Pour terminer, le flux f_1 traverse le serveur S_5. Pour obtenir les mêmes contraintes que dans les serveurs précédents, nous devons connaître les contraintes des autres entrées de ce serveur. En effet, nous n'avons pas encore défini des contraintes pour les courbes cumulées $F_{2_{S2}}$ et $F_{3_{S4}}$. Pour la première, nous pouvons déjà définir :

$$\beta_{S2_2} \triangleq \delta_{\mathrm{hDev}(\alpha_{2_{S1}}, \beta_2)} \quad \text{et} \quad \alpha_{2_{S2}} \triangleq \min\left(\alpha_{2_{S2}} \oslash_{\mathcal{F}\uparrow} \beta_{S2_2}, \delta_0\right), \tag{6.8}$$

qui nous permet de démontrer la propriété suivante.

Lemme 6.37 (`min_residual_service_S2_2, maximal_departure_S2_2`) La courbe β_{S2_2} est une courbe minimale de service pour le serveur résiduel S_{2_2} contraint au flux $F_{2_{S2}}$ par $\alpha_{2_{S1}}$. La courbe $\alpha_{2_{S2}}$ est une courbe d'arrivée pour la fonction $F_{2_{S2}}$.

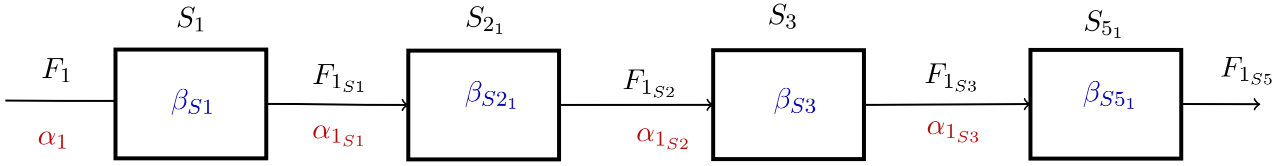


FIGURE 6.12 – Résumé des contraintes rencontrées par le flux f_1. En rouge, nous donnons les courbes d'arrivée pour la courbe cumulative correspondante en noir établies avec la Proposition 6.9. En bleu, nous donnons le service résiduel offert au flux f_1 pour chaque serveur établi avec la Proposition 6.10.

Nous avons notre contrainte pour $F_{2_{S2}}$. Pour obtenir celle de $F_{3_{S4}}$, il nous faut de manière similaire déterminer la contrainte de sortie de S_{2_3} :

$$\beta_{S2_3} \triangleq \delta_{\mathrm{hDev}(\alpha_{3_{S1}}, \beta_2)} \quad \text{et} \quad \alpha_{3_{S2}} \triangleq \min\left(\alpha_{3_{S2}} \oslash_{\mathcal{F}\uparrow} \beta_{S2_3}, \delta_0\right), \tag{6.9}$$

Lemme 6.38 (`min_residual_service_S2_3, maximal_departure_S2_3`) La courbe β_{S2_3} est une courbe minimale de service résiduel pour le serveur S_{2_3} contraint au flux $F_{3_{S2}}$ par $\alpha_{3_{S1}}$. La courbe $\alpha_{3_{S2}}$ est une courbe d'arrivée pour la fonction $F_{3_{S2}}$.

Puis, nous devons supposer que α_5 est une courbe d'arrivée pour F_5 pour définir :

$$\beta_{S4} \triangleq \delta_{\mathrm{hDev}(\alpha_{3_{S2}} + \alpha_5, \beta_4)} \quad \text{et} \quad \alpha_{3_{S4}} \triangleq \min\left(\alpha_{3_{S2}} \oslash_{\mathcal{F}\uparrow} \beta_{S4}, \delta_0\right) \tag{6.10}$$

Ainsi, nous pouvons montrer la propriété suivante.

Lemme 6.39 (`min_residual_service_S4, maximal_departure_S4`) La courbe β_{S4} est une courbe minimale de service de le serveur résiduel de S_4 contraint aux flux $\{F_{3_{S2}}, F_5\}$ par les courbes $\{\alpha_{3_{S2}}, \alpha_5\}$. La courbe $\alpha_{3_{S4}}$ est une courbe d'arrivée pour la fonction $F_{3_{S4}}$.

Nous avons tout les éléments pour déterminer une contrainte satisfaite par f_1 au serveur S_5. Nous pouvons donc définir :

$$\beta_{S5_1} \triangleq \delta_{\mathrm{hDev}(\alpha_{1_{S3}} + \alpha_{2_{S2}} + \alpha_{3_{S4}}, \beta_5)} \tag{6.11}$$

et ainsi démontrer la propriété suivante.

Lemme 6.40 (`min_residual_service_S5`) La courbe β_{S5_1} est une courbe minimale de service pour le serveur S_{5_1} contraint aux flux $\{F_{1_{S3}}, F_{2_{S2}}, F_{3_{S4}}\}$ par les courbes $\{\alpha_{1_{S3}}, \alpha_{2_{S2}}, \alpha_{3_{S4}}\}$.

Nous donnons une figure représentant la traversée du réseau vue par le flux f_1 dans la Figure 6.12. Cette figure résume les contraintes que nous avons établies précédemment.

Nous cherchons ensuite à déterminer une courbe de service qui sera offerte au flux f_1 pour la traversée de bout en bout du réseau. En effet, cette courbe de service sera la courbe de service minimale du serveur S_{f_1}. Ce serveur est le serveur vu par le flux f_1.

Pour cela, nous prenons tous les services que reçoit le flux f_1 pour les concaténer. En effet, comme cela est montré sur la Figure 6.12 et défini dans la Définition 6.10, nous pouvons définir :

$$S_{f_1} \triangleq (S_1; S_{2_1}; S_3; S_{5_1})$$

Lemme 6.41 (`f1_servers_crossed`) Nous avons $(F_1, F_{1_{S5}}) \in S_{f_1}$.

En reprenant la Proposition 6.5, nous pouvons établir une courbe de service pour un tel serveur

$$\beta_{1_{E2E}} \triangleq \beta_{S1} * \beta_{S2} * \beta_{S3} * \beta_{S5}. \tag{6.12}$$

où "E2E" signifie *end to end* donc de bout en bout.

Pour borner le délai de bout en bout, nous allons reprendre la Proposition 6.1. Cette propriété permet de borner la valeur de d_1 (pour rappel, $d_1 \triangleq d(F_1, F_{1_{S5}})$). Finalement, nous avons la propriété suivante.

Lemme 6.42 (`f1_E2E_delay`) Nous avons : $d_1 \leqslant \mathrm{hDev}(\alpha_1, \beta_{1_{E2E}})$.

Et ainsi, nous avons une borne sur le temps de traversée des messages utilisant le flux f_1. Il faut suivre le même cheminement pour les autres flux.

6.4.2 Preuve Coq

Formalisons maintenant cette partie en Coq. Nous avons placé le code complet de cette partie dans la fichier :

`examples/case_study/arbitrary_flows.v`

Nous allons commencer par poser des hypothèses sur les flux. Nous le ferons dans un second temps pour les serveurs. Les flux sont modélisés par des fonctions dans $\mathcal{C}$ donc des `flow_cc` et disposeront d'une contrainte, une courbe d'arrivée, dans $\mathcal{F}^{\uparrow}$ donc `Fup`. Elles seront liées ensuite avec la définition de courbe maximale d'arrivée (cf. Définition 6.19). En Coq, cette traduction donne pour le flux f_1 :

```
Section CaseStudy.

(** Entrance flows *)
Variables F1 F2 F3 F4 F5 : flow_cc.

(** Arrival for flow F1, F2, F3, F4 and F5 *)
Variables alpha1 alpha2 alpha3 alpha4 alpha5 : Fup.

(** Each arrival is a constraint of a entrance flow*)
Hypothesis Halpha1 : is_maximal_arrival F1 alpha1.
Hypothesis Halpha2 : is_maximal_arrival F2 alpha2.
Hypothesis Halpha3 : is_maximal_arrival F3 alpha3.
Hypothesis Halpha4 : is_maximal_arrival F4 alpha4.
Hypothesis Halpha5 : is_maximal_arrival F5 alpha5.
```

Nous respectons la nomenclature suivante : `Fi` pour les courbes cumulatives et `alphai` pour les courbes d'arrivée respectives.

Pour les serveurs, nous allons créer les variables de serveurs en fonctions de la Figure 6.11. Par exemple, pour S_1, nous avons :

```
Variables (S1 : 3.-server).
```

puisqu'il dispose de trois entrées et sorties. Puisque nous suivons la décomposition de la Figure 6.11, le serveur S_2 sera séparé en trois serveurs avec une seule entrée et sortie. Nous avons ainsi :

```
Variables (S2_1 : 1.-server) (S2_2 : 1.-server) (S2_3 : 1.-server).
```

Pour les serveurs S_3 et S_4 nous faisons comme pour le serveur S_1 avec deux entrées. Pour le serveur S_5, nous créons (pour les mêmes raisons que S_2) deux serveurs qui vont représenter les transferts des flux f_1, f_2 et f_3 dans `S5_1` et f_4 et f_5 dans `S5_2`.

```
Variables (S5_1 : 3.-server) (S5_2 : 2.-server).
```

Nous ajoutons ensuite les hypothèses que ces serveurs sont avec une politique *FIFO*.

```
(** FIFO hypothesis for S1, S3, S4, S5_1 and S5_2 *)
Hypothesis (H_S1 : FIFO_service_policy S1).
Hypothesis (H_S3 : FIFO_service_policy S3).
Hypothesis (H_S4 : FIFO_service_policy S4).
Hypothesis (H_S5_1 : FIFO_service_policy S5_1).
Hypothesis (H_S5_2 : FIFO_service_policy S5_2).
```

Nous ajoutons ensuite à chaque serveur une courbe de service minimal (cf. Définition 6.21). Par exemple, pour le serveur S_1, la courbe de service est β_{S1} et donc l'hypothèse de contrainte est :

```
Variable beta1 beta2 beta3 beta4 beta5 : F0up.

(** Each service is a constraint for a server *)
Hypothesis Hbeta_S1 : is_min_service (aggregate_server S1) beta_S1.
Hypothesis Hbeta_S2_1 : is_min_service (aggregate_server S2_1) beta2.
Hypothesis Hbeta_S2_2 : is_min_service (aggregate_server S2_2) beta2.
Hypothesis Hbeta_S2_3 : is_min_service (aggregate_server S2_3) beta2.
Hypothesis Hbeta_S3 : is_min_service (aggregate_server S3) beta3.
Hypothesis Hbeta_S4 : is_min_service (aggregate_server S4) beta4.
Hypothesis Hbeta_S5 : is_min_service (aggregate_server S5_1) beta5.
Hypothesis Hbeta_S6 : is_min_service (aggregate_server S5_2) beta5.
```

Nous devons déclarer des courbes de services dans $\mathcal{F}_0^{\uparrow}$ pour pouvoir utiliser la formalisation de la Proposition 6.10 sur les délais *FIFO*.

Nous pouvons maintenant nous occuper de traduire en Coq la topologie du modèle. Cela consiste à créer les variables de représentation des courbes cumulatives en sortie de chaque serveur comme tracé en Figure 6.11. En Coq, pour le serveur S_1, cela revient à :

```
(* Courbes cumulatives en sorties de S1 *)
Variables (F1_S1 F2_S1 F3_S1 : flow_cc).

Hypothesis H_server_S1 :
  S1 (finfun_of_tuple [tuple F1; F2; F3])
     (finfun_of_tuple [tuple F1_S1; F2_S1; F3_S1]).
```

La fonction `finfun_of_tuple` permet de regrouper les flux dans un seul vecteur. Ainsi, nous pouvons utiliser le fait que `S1` prend en entrée et sortie un vecteur de trois flux. Nous avons une hypothèse similaire pour chaque serveur du réseau.

```
(** Server 2 -1 *************************************************)
Variables F1_S2 : flow_cc.

Hypothesis H_server_S2_1 : S2_1 [ffun⇒ F1_S1] [ffun⇒ F1_S2].

(** Server 2 -2 *************************************************)
Variables F2_S2 : flow_cc.

Hypothesis H_server_S2_2 : S2_2 [ffun⇒ F2_S1] [ffun⇒ F2_S2].

(** Server 2 -3 *************************************************)
Variables F3_S2 : flow_cc.

Hypothesis H_server_S2_3 : S2_3 [ffun⇒ F3_S1] [ffun⇒ F3_S2].

(** Server 3 ****************************************************)
Variables F1_S3 F4_S3 : flow_cc.

Hypothesis H_server_S3 :
  S3
    (finfun_of_tuple [tuple F1_S2; F4])
    (finfun_of_tuple [tuple F1_S3; F4_S3]).

(** Server 4 ****************************************************)
Variables F3_S4 F5_S4 : flow_cc.

Hypothesis H_server_S4 :
  S4
    (finfun_of_tuple [tuple F3_S2; F5])
    (finfun_of_tuple [tuple F3_S4; F5_S4]).

(** Server 5 -1 ************************************************)
Variables F1_S5 F2_S5 F3_S5 : flow_cc.

Hypothesis H_server_S5_1 :
  S5_1
    (finfun_of_tuple [tuple F1_S3; F2_S2; F3_S4])
    (finfun_of_tuple [tuple F1_S5; F2_S5; F3_S5]).

(** Server 5 -2 ************************************************)
Variables F4_S5 F5_S5 : flow_cc.

Hypothesis H_server_S5_2 :
  S5_2
    (finfun_of_tuple [tuple F4_S3; F5_S4])
    (finfun_of_tuple [tuple F4_S5; F5_S5]).
```

Nous souhaitons formaliser en Coq une borne sur le temps de traversée du flux f_1, comme nous l'avons fait dans la preuve papier écrite dans la partie précédente. Nous avons déjà formulé les hypothèses sur les arrivées de chaque flux et sur les services rendus par chaque serveur. De plus, nous avons aussi formalisé les hypothèses sur les courbes d'arrivées, sur les services minimums rendues par chaque serveurs et sur leurs politiques d'ordonnancement.

Pour cela, nous allons reprendre les équations et les propriétés de cette partie et les formaliser en Coq. La première définition est la définition de la courbe β_{S1}, définie en (6.2), la courbe de service résiduelle du serveur résiduel de S_1. Ainsi, nous définissons :

```
Definition beta_S1_delay : Fup :=
  delta (hDev (alpha1 + alpha2 + alpha3)%F beta_S1)%:nngenum.
```

Nous souhaitons ensuite formaliser le Lemme 6.33. Pour cela, nous écrivons d'abord la vectorisation des contraintes sur les arrivées de S_1.

```
Definition alpha_S1 : Fup^3 := finfun_of_tuple [tuple alpha1; alpha2; alpha3].
```

Il s'agit là d'un détail Coq nous permettant d'améliorer la lisibilité de la formalisation du Lemme 6.33. Celui ci donne :

```
Lemma min_residual_service_S1 i :
  is_min_service
    (residual_server_constr S1 i [ffun i => alpha_S1 i : F]) beta_S1_delay.
```

Nous allons ensuite définir les contraintes de sortie pour le serveur S_1. Nous formalisons donc les équations (6.3, 6.4) et (6.5) :

```
Definition alpha1_S1 := ((alpha1 / beta_S1) + delta 0)%D.
Definition alpha2_S1 := ((alpha2 / beta_S1) + delta 0)%D.
Definition alpha3_S1 := ((alpha3 / beta_S1) + delta 0)%D.

Let alpha_S1_out : Fup^3 := finfun_of_tuple [tuple alpha1_S1; alpha2_S1; alpha3_S1].
```

Nous montrons ensuite le Lemme 6.34.

```
Lemma maximal_departure_S1 i :
  is_maximal_arrival (finfun_of_tuple [tuple F1_S1; F2_S1; F3_S1] i) (alpha_S1_out i).
```

La suite est similaire à la preuve papier. Nous donnons les services résiduels du serveur S_2, donc les courbes `beta_S2_1`, `beta_S2_2` et `beta_S2_3` définies respectivement en (6.6, 6.8) et (6.9). Cependant, il n'est pas nécessaire de donner les hypothèses de politique *FIFO* pour les serveurs S_{2_1}, S_{2_2} et S_{2_3} puisque ce sont des serveurs à un seul flux. Ainsi, les preuves des lemmes 6.35, 6.37 et 6.38 font appel à la propriété donnant que chaque serveur à une seule entrée/sortie satisfait forcément une politique *FIFO* (cf. Lemme 6.24).

Nous agissons pareil pour donner les contraintes de sorties de S_2. Ainsi, les fonctions `alpha1_S2` , `alpha2_S2` et `alpha3_S2` sont aussi données en (6.6), (6.8) et (6.9) et sont formalisées de la même manière que les courbes d'arrivée précédentes.

Pour prouver en Coq la borne sur le temps de traversés de f_1, nous allons considérer que les définitions des termes données en (6.7), (6.10) et (6.11) ont été formalisées suivant la technique donnée ci dessus. Il en est de même pour les propriétés liées à ces termes, donc les lemmes 6.36, 6.39 et 6.40.

L'idée maintenant est de démontrer le Lemme 6.41. Pour cela, il nous faut formaliser le serveur S_{f_1}. Il nous faut en Coq donner le détail des serveurs résiduel contraint traversés par le flux f_1. Donc, il nous faut pour chaque serveur donner le numéro du flux contraint. Pour le serveur S_1, il s'agit du flux numéro 0 (les indices commencent à 0). En Coq, il faut indiquer ce numéro par (`inord` 0) car il s'agit d'un nombre naturel dont on sait qu'il ne peut pas être supérieur à n. Nous avons donc, pour déclarer le premier serveur résiduel S_1 contraint par le tuple de `alpha_S1` :

```
  residual_server_constr S1 (inord 0) [ffun i => alpha_S1 i : F] '
```

Enfin, pour concaténer l'ensemble de ces serveurs résiduel, nous reprenons la notation "'; " déclarée à la suite de la Définition 6.10. Le serveur traversé par f_1 est donc formalisé de la manière suivante :

```
Definition f1_servers :=
  (residual_server_constr S1 (inord 0) [ffun i ⇒ alpha_S1 i : F] ';
   residual_server_constr S2_1 (inord 0) [ffun _ ⇒ alpha1_S1 : F] ';
   residual_server_constr S3 (inord 0) [ffun i ⇒ alpha_S3 i : F] ';
   residual_server_constr S5 (inord 0) [ffun i ⇒ alpha_S5 i : F])%S.
```

Le serveur `f1_servers` est le serveur S_{f_1}. Le Lemme 6.41 se formalise par :

```
Lemma f1_servers_crossed : f1_servers F1 F1_S5.
```

Pour formaliser la Définition 6.12, nous avons :

```
Definition beta1_E2E := (beta_S1 ∗ beta_S2_1 ∗ beta_S3 ∗ beta_S5_1)%D.
```

Cette dernière définition nous permet enfin d'établir la borne sur le délai pour le flux f_1. Cette propriété correspond au Lemme 6.42 :

```
Lemma f1_E2E_delay :
  ((delay F1 F1_S5)%:nngenum ⩽ (hDev alpha1 beta1_E2E_service)%:nngenum)%E.
```

Ce dernier lemme termine la formalisation d'une borne sur le temps de traversée des messages à travers f_1. Pour borner les autres flux, il faut procéder de la même manière. Ces résultats se trouvent dans le fichier cité en début de cette section : `examples/case_study/arbitrary_flow.v`.

Un tel résultat nous donne des expressions algébriques. Pour appliquer ces expressions sur un cas réel, elles doivent être calculées avec des valeurs concrètes. Nous donnerons une telle application dans le chapitre suivant.

6.5 Conclusion

L'objectif de ce chapitre était d'augmenter le niveau de confiance dans les propriétés et résultats du *Calcul réseau*. Nous avons dans ce chapitre formalisé en Coq l'ensemble des définitions et propriétés nécessaires pour être appliquées sur un cas d'étude. Il s'agit des formalisations des définitions de courbes cumulatives, des notions de déviations horizontales et verticales, des notions de *backlog* et délais, de quelques fonctions usuelles, des courbes d'arrivée maximale, des courbes de service minimale et de la politique d'ordonnancement *FIFO*. Pour le cas d'étude, nous avons eu besoin de formaliser en Coq les propriétés pour borner les délais, pour établir les services minimum de serveur concaténé, pour connaître la courbe d'arrivée d'un flux sortant et du service minimal que peut offrir un serveur *FIFO*.

Nous avons également développé d'autres propriétés, non utilisées dans notre cas d'étude, mais disponibles pour une application sur d'autres cas. Nous avons formalisé la notion de période de *backlog* ainsi que des résultats autour du début de cette période, la notion de service minimum strict, de la politique d'ordonnancement à priorité fixe avec préemption et du service minimum strict offert par des serveurs suivant cette politique.

Au total, cette partie du développement est compris dans plus de 2000 lignes de code Coq. Celles ci peuvent se trouver dans le code disponible sur :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien`

Cependant, comme nous avons pu le voir à la fin de notre cas d'étude, les résultats que nous obtenons sont des expressions algébriques. Ces expressions algébriques sont validées par le noyau de Coq mais le calcul de ces valeurs ne l'est pas. Pour calculer ces valeurs, nous pouvons détailler les calculs en Coq pour calculer des valeurs concrètes sur ces expressions. Nous avons appliqué cette méthode dans [RBR19] en utilisant les librairies sur les nombres rationnels de Coq, `QArith` et `Qreals`. Cette méthode est longue et guère envisageable pour une mise à l'échelle puisqu'il est nécessaire de prouver chaque calcul en Coq.

L'alternative que nous avons choisie est d'utiliser un outil extérieur. Plusieurs outils permettent de calculer des convolutions ou encore des déviations horizontales. L'utilisation de ces algorithmes de calcul et leur implémentation va cependant altérer la confiance que nous avons dans la valeur numérique. En effet, il est possible que l'implémentation soit erronée pour une valeur particulière qui n'a pas encore été découverte. Nous allons donc, dans la suite de ce manuscrit, nous intéresser à la vérification de tel développement dans l'idée d'obtenir le même niveau de confiance que les preuves apportées dans ce chapitre.

Chapitre 7

Vérification de calculs de fonctions *(min, +)*

Sommaire

7.1 Introduction au calcul *(min, +)* . . . **89**

7.1.1 Sous classes de fonctions pour un calcul effectif . . . 90

7.1.2 Application avec un interpréteur de commande *(min, +)* sur le cas d'étude . . 90

7.2 Définitions des classes de fonctions . . . **94**

7.2.1 Fonctions ultimement pseudo périodique (UPP) . . . 94

7.2.2 UPP et fonctions affines par morceau . . . 95

7.3 Stabilités de $\mathcal{F}_{UPP}$ par les opérations *(min, +)* . . . **97**

7.4 Stabilités de $\mathcal{F}_{UPP-PA}$ par les opérations *(min, +)* . . . **99**

7.5 Vérification des opérations sur $\mathcal{F}_{UPP-PA}$ . . . **101**

7.5.1 Critère d'égalité fini sur $\mathcal{F}_{UPP-PA}$ . . . 102

7.5.2 Critère majorant . . . 103

7.6 Implémentation . . . **103**

7.7 Exemple sur le cas d'étude . . . **104**

7.8 Conclusion . . . **107**

L'objectif de ce chapitre est de présenter nos travaux autour de la vérification de calculs de fonctions *(min, +)*. Ces travaux sont toujours justifiés par l'envie d'augmenter la confiance dans les bornes temporelles établies à l'aide du *Calcul réseau*.

Comme nous avons pu le voir, les bornes obtenues aux cours du chapitre précédent s'expriment à l'aide d'expressions algébriques utilisant des opérations du *dioïde* des fonctions *(min, +)*. Les deux premiers chapitres ont permis d'augmenter la confiance dans ces bornes et nous avons vu que nous étions capable d'appliquer cette confiance à un cas d'étude.

Dans ce chapitre, nous allons prendre un oracle : un outil capable d'effectuer les opérations du *dioïde* des fonctions *(min, +)*. L'idée est de demander à cet oracle d'effectuer un calcul de fonctions *(min, +)*. Nous prenons ensuite la fonction obtenue pour tester sa validité. Ce test va s'appuyer sur des critères formalisés et calculables par Coq. Si le test est correct, alors nous aurons vérifié en Coq que l'oracle a donné un bon résultat.

Nous avons proposé dans nos travaux présentés dans [RRB21] un développement permettant de remplir cet objectif. Ce chapitre a donc pour fonction de restituer ces travaux. La version du code que nous allons présenter ici est disponible dans une branche particulière :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/nfm_21_submission`

Pour pouvoir compiler pas à pas cette partie, il faut installer les librairies générales et de *dioïde* comme expliqué dans l'Annexe A.

7.1 Introduction au calcul *(min, +)*

Cette partie a pour but de présenter comment les valeurs concrètes sont calculés dans un des outils du *Calcul réseau*. En effet, ceux ci utilisent des sous classes de fonctions plus restrictives que les classes généralistes introduites dans ce manuscrit. Ainsi, dans un premier temps, nous donnerons une présentation des fonctions utilisées dans les algorithmes utilisés par Bouillard et Thierry dans [BT08]. Ces sous classes ont été choisies dans la suite de nos travaux dans [RRB21].

À la suite de cette présentation, nous allons reprendre le cas d'étude du chapitre précédent pour y appliquer des valeurs concrètes. Ainsi, nous présenterons un outil de calcul du *Calcul réseau* : l'interpréteur de commande *(min, +)* [Rea10] donné à l'adresse :

`https://www.realtimeatwork.com/minplus-playground`

Nous présenterons cet outil en détaillant un calcul du cas d'étude. Le script complet que nous présentons en suivant se trouve dans le fichier `examples/case_study/CaseStudy.nc`.

7.1.1 Sous classes de fonctions pour un calcul effectif

Alors que la théorie du *Calcul réseau* se fonde sur des classes de fonctions assez générales, pour obtenir des valeurs numériques, il faut se restreindre à des classes calculables et stables. Dans [BT08], Bouillard et Thierry nous donnent plusieurs ensembles de fonctions. Nous allons les reprendre et les présenter ici pour pouvoir ensuite les formaliser.

En *Calcul réseau*, il est commun d'avoir un comportement périodique. Pour les décrire, nous allons utiliser des fonctions qui sont en définitive pseudo périodiques ou *ultimately pseudo-periodic* (UPP). Cet ensemble sera dénoté $\mathcal{F}_{UPP}$ et nous formaliserons plus loin dans ce chapitre qu'une fonction f appartient à $\mathcal{F}_{UPP}$ si, pour un segment initial T, une période d et un incrément c, pour tout t après T, nous avons $f(t+d) = f(t) + c$. Pour avoir une description finie des ces fonctions, il suffit d'avoir : les valeurs de T, d et c ainsi que les valeurs de la fonction jusqu'à $T + d$.

Nous allons aussi prendre la sous classe de $\mathcal{F}$ des fonctions qui sont affines par morceaux ou *Piecewise Affine* (PA). Pour ces fonctions, il est suffisant de donner, pour chaque morceau, le point de discontinuité, le pente de la fonction et le décalage de la fonction (*offset*). Ces paramètres peuvent être enregistrés dans une liste mais celle ci est infinie et donc non implémentable.

Nous définirons plus formellement par la suite l'ensemble $\mathcal{F}_{UPP-PA}$ qui est l'intersection entre $\mathcal{F}_{UPP}$ et les fonctions PA. Ces éléments peuvent être alors représentés de manière finie en donnant les paramètres T, d et c de $\mathcal{F}_{UPP}$ et les segments initiaux de la liste des fonctions PA représentant la fonction sur $[0; T + d[$. Nous donnons un exemple de fonction $\mathcal{F}_{UPP-PA}$ dans la Figure 7.1. Cet exemple est composé de deux fonctions f et g définies par :

$$f : t \mapsto \min\left(2t, \left\lceil\frac{t}{2}\right\rceil + \left\lceil\frac{t}{4}\right\rceil\right) \qquad g : t \mapsto \min\left(\frac{1}{3}t, \frac{t+8}{11}\right)$$

Un outil nous donne une fonction UPP-PA nommée h et affirme que cette fonction est :

$$h : t \mapsto f(t) + g(t).$$

et nous donne les paramètres de cette fonction UPP-PA affirmant cette égalité. Alors, pour prouver l'égalité $f + g = h$, il est suffisant de vérifier que $f(t_i) + g(t_i) = h(t_i)$ pour la liste de t_i suivante :

$$[0;\ 0.1;\ 0.9;\ 1;\ 1.1;\ 1.9;\ 2;\ 2.1;\ 2.9;\ 3;\ 3.1;\ 3.9;\ 4;\ 4.1;\ 5.9;\ 6;\ 6.1;\ 7.9]\,.$$

Nous verrons dans ce chapitre que cette liste est l'union des points d'abscisses données en Figure 7.1 pour vérifier l'égalité en ces points. Nous y ajouterons deux points intermédiaires pour vérifier la partie affine entre chaque point de cette union.

7.1.2 Application avec un interpréteur de commande *(min, +)* sur le cas d'étude

Au cours du chapitre précédent, nous avons déroulé un cas d'étude en utilisant notre formalisation du *Calcul réseau*. Au cours de celui ci, nous avons déterminé des expressions algébriques qui permettent de borner des temps de traversées sur le réseau utilisé dans cet exemple. Nous avons déroulé l'étude sur le flux f_1 et avons établi au cours du Lemme 6.42 la relation suivante :

$$d_1 \leqslant \mathrm{hDev}(\alpha_1, \beta_{1_{E2E}})$$

où d_1 est formellement défini comme le délai (cf. Définition 6.13) entre les courbes cumulatives en entrée et en sortie du réseau du flux f_1. La courbe α_1 est une courbe d'arrivée pour la courbe cumulative

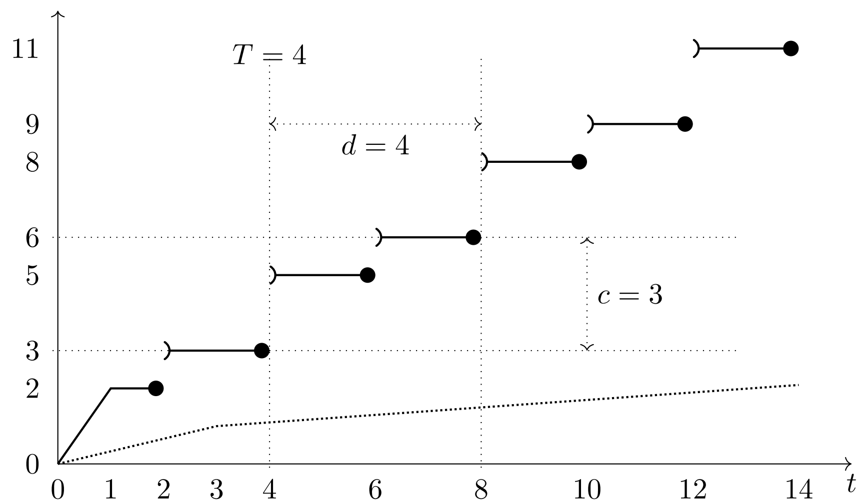


FIGURE 7.1 – f (pleine) et g (pointillé) sont des fonctions UPP-PA. Nous avons une représentation finie avec les éléments :

— $T = 4$, $d = 4$, $c = 3$,
— discontinuités :
  — abscisses $[0, 1, 2, 4, 6]$,
  — ordonnées $[0, 2, 2, 3, 5]$,
— pentes : $[2, 0, 0, 0, 0]$
— décalage : $[0, 2, 3, 5, 6]$.

— $T = 4$, $d = 4$, $c = \frac{4}{11}$,
— discontinuités :
  — abscisses $[0, 3]$,
  — ordonnées $[0, 1]$,
— pentes : $\left[\frac{1}{3}, \frac{1}{11}\right]$
— décalage : $[0, 1]$.

d'entrée de f_1 et $\beta 1_{E2E}$ est une courbe de service pour le serveur résiduel complet que traverse f_1. Nous avons détaillé ces termes et les manipulations nécessaire pour les obtenir au cours de la Section 6.4.

Nous allons ici nous intéresser aux calculs de valeurs concrètes sur le terme $\text{hDev}(\alpha_1, \beta 1_{E2E})$. En effet, pour appliquer cette expression sur un cas concret, il nous faut pouvoir déterminer que vaut la déviation entre les deux fonctions α_1 et $\beta 1_{E2E}$.

Pour cela, nous pouvons utiliser des outils du *Calcul réseau* existant. Nous en avons présenté une liste dans le Chapitre 2. L'un d'entre eux est l'interpréteur de commande *(min, +)* en ligne [Rea10] et disponible à l'adresse suivante :

`https://www.realtimeatwork.com/minplus-playground`

Nous proposons de décomposer le calcul de ce terme dans cette partie pour comprendre les enjeux de ce chapitre. Le développement complet se trouve dans le fichier :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/blob/phd-lucien/examples/case_study/CaseStudy.nc`

(Nous appellerons ce fichier plus communément `CaseStudy.nc`). Contrairement à ce que nous avons évoqué en début de chapitre, ce fichier peut se trouver dans la branche `phd-lucien`. Nous considérons que le lecteur a pris connaissance du détail théorique développé dans la Section 6.4 du chapitre précédent.

Au début de cette même section, nous avons attribué des valeurs pour les arrivées des flux et le traitement offert par les serveurs. Nous allons ici les reprendre pour les intégrer dans nos calculs avec l'interpréteur de commande.

Les messages arrivant par chaque flux sont des messages de 8kBit. Le débit des informations à travers ces flux est de 20kBit/s. Pour modéliser une courbe d'arrivée sur ces flux, nous écrivons :

α_1 : `alpha1 := bucket(20000, 8000)` **Commande 1**

signifiant littéralement que `alpha1` est une fonction qui vaut :

$$\alpha_1 \triangleq x \in \mathbb{R}_+ \mapsto \begin{cases} 0 & \text{si } x = 0, \\ 20000x + 8000 & \text{sinon.} \end{cases} \tag{7.1}$$

L'interpréteur nous donne alors le message suivant :

» uaf([(0,0)]](0,8000)20000(+Infinity,+Infinity)[)

Cette fonction est donc une fonction affine PA au sens introduit dans la partie précédente. Elle est composé de deux morceaux : [(0,0)] est le premier segment, il se compose d'un seul point 0 en 0. Le second segment est](0,8000)20000(+Infinity,+Infinity)[, il s'agit d'un segment avec pour abscisses $]0, +\infty[$ et pour ordonnées une fonction affine avec un offset de 8000 et une pente de 20000.

Comme chaque flux est supposé fonctionner de la même manière, nous donnons alors :

```
alpha2 := alpha1
alpha3 := alpha1
alpha4 := alpha1
alpha5 := alpha1
```

Les serveurs sont supposés traiter au minimum 100kBit/s. Nous avons donc une fonction affine de pente 100000 qui vaut 0 en 0. Nous donnons alors :

β_1 : beta1:= affine(100000, 0)	**Commande 2**

pour indiquer que `beta1` est une fonction valant :

$$\beta_1 \triangleq x \mapsto 100000x \tag{7.2}$$

Dans ce cas là, l'interpréteur de commande nous indique que la fonction `beta1` est :

» uaf([(0,0)100000(+Infinity,+Infinity)[)

Cette fonction est aussi PA et elle est composée d'un unique segment d'abscisses $[0; +\infty[$ sur une fonction affine sans offset et une pente de 100000.

Comme pour les courbes d'arrivées, nous répétons l'opération pour `beta2` jusqu'à `beta5` puisque les serveurs sont supposés avoir le même fonctionnement minimal.

Nous allons dans un premier temps chercher à calculer la valeur de β_{S1} définie en (6.2). Nous savons alors qu'il est nécessaire de calculer la somme des arrivées de ce premier serveur. Posons alors $\alpha_{S1} \triangleq \alpha_1 + \alpha_2 + \alpha_3$. Nous donnons alors :

α_{S1} : alpha_S1:= alpha1 + alpha2 + alpha3	**Commande 3**

La calculette *(min, +)* a effectué le calcul de l'addition des trois fonctions α_1, α_2 et α_3. A l'aide de la fonction obtenue, nous pouvons calculer $d_{\beta S1} \triangleq \text{hDev}(\alpha_{S1}, \beta_1)$, déjà donné en (6.2).

$d_{\beta_{S1}}$: beta_S1_delay:= hdev(alpha_S1, beta1)	**Commande 4**

Avec cette commande, nous demandons à l'interpréteur de calculer la déviation horizontale entre deux fonctions. En poursuivant le processus, nous établissons donc la fonction $\beta_{S1} \triangleq \delta_{d_{\beta_{S1}}}$, avec δ définie en Définition 6.17, en utilisant la commande suivante :

β_{S1} : beta_S1 := delay(beta_S1_delay)	**Commande 5**

où la fonction `delay` correspond à la fonction δ et permet de terminer la définition de la fonction (6.2). La fonction obtenue est :

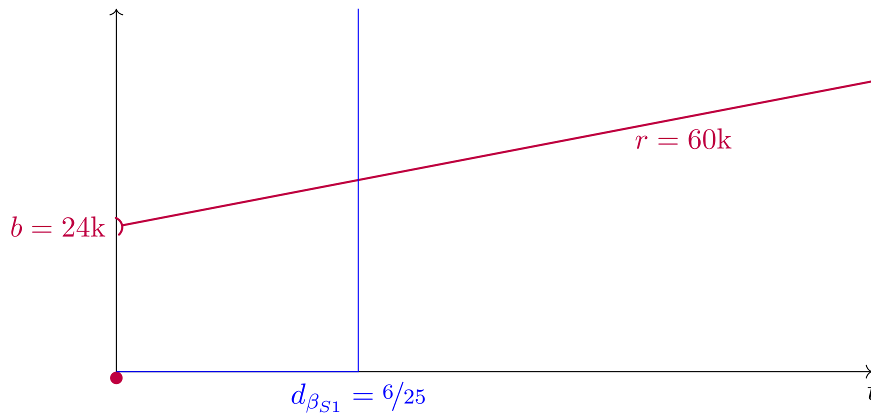


FIGURE 7.2 – Affichage de l'interpréteur. Nous traçons en rouge la somme des arrivées α_{S1} et en bleu la courbe de service résiduelle β_{S1}.

```
» uaf([(0,0)0(6/25,0)] ](6/25,+Infinity)0(+Infinity,+Infinity)[)
```

Cette fonction PA a deux segments. Le premier s'exprime dans $[0; 6/25]$ et vaut 0 en tout point. Le second est évalué dans $]6/25; +\infty[$ est n'a pas besoin de pente puisqu'il dispose d'un offset de $+\infty$.

L'interpréteur permet d'afficher le tracé de certaines fonctions. Nous donnons en Figure 7.2 l'affichage des fonctions α_{S1} et β_{S1}.

Pour donner d'autres opérations de l'interpréteur et continuer à calculer des valeurs concrètes du cas d'étude de la Section 6.4, nous allons définir $\alpha_{1_{S1}}$ dans le langage de l'interpréteur. D'après la définition (6.3), nous savons que

$$\alpha_{1_{S1}} \triangleq \min(\alpha_1 \oslash_{\mathcal{F}^{\uparrow}} \beta_{S1}, \delta_0)$$

Dans un premier temps, définissons δ_0.

δ_0 : `d0 := delay(0)`	**Commande 6**

Puis, donnons une valeur intermédiaire à $\alpha_{1_{S1}}$ avec :

$\alpha'_{1_{S1}}$: `alpha1_S1':= ( alpha1 / beta_S1 )`	**Commande 7**

où l'opérateur / correspond à la déconvolution. Nous n'avons pas de différence entre les déconvolutions comme cela a pu être mentionné au cours de la Remarque 5.9. Finalement, nous donnons $\alpha_{1_{S1}}$ avec

$\alpha_{1_{S1}}$: `alpha1_S1 := alpha1_S1' /\ d0`	**Commande 8**

Cette fonction est aussi PA :

```
» uaf([(0,0)]](0,12800)20000(+Infinity,+Infinity)[)
```

avec un point en 0 puis un segment sur $]0; +\infty[$ avec un offset de 12800 et une pente de 20000.

Au cours de la Section 6.4, nous avons aussi donné les expressions de $\alpha_{2_{S1}}$ (6.4) et $\alpha_{3_{S1}}$ (6.5). Le calcul et les commandes sont similaires à `alpha1_S1` donc nous ne donnons pas le détail de `alpha2_S1` et `alpha3_S1`.

Pour rappel, les expressions que nous venons de calculer correspondent aux expressions démontrées dans les lemmes 6.33 et 6.34. Nous continuons à calculer les valeurs des courbes de contraintes du flux f_1 pour terminer l'analyse temporelle de ce flux. Ainsi, nous avons les variables `beta_S2_1` et `alpha1_S2`, nous reprenons la définition de (6.6). Nous faisons de même pour `beta_S3` et `alpha_S3` avec (6.7) en calculant les variables intermédiaires nécessaires. Elles ne sont pas détaillés ici, cependant leurs expressions se trouvent dans la preuve papier de 6.4. Il va en de même pour le calcul de `beta_S5_1` correspondant à (6.11). Le résumé de ces courbes a été tracé dans le chapitre précédent, à la Figure 6.12.

Ainsi, nous sommes capable de calculer la courbe $\beta_{1_{E2E}}$ comme définie en (6.12). Cette courbe est, pour rappel, une courbe de service pour le réseau traversé par f_1.

$\beta_{1_{E2E}}$: `beta_1_E2E = beta_S1 * beta_S2 * beta_S3 * beta_S5` **Commande 9**

La fonction obtenue est PA :

» `uaf([(0,0)0(3612/3125,0)]](3612/3125,+Infinity)0(+Infinity,+Infinity)[)`

avec deux segments (comme pour la fonction de la Commande 5). Nous avons un premier segment en $[0; 3612/3125]$ qui vaut 0 en tout point et un second segment de $]3612/3125; +\infty[$ valant $+\infty$.

Cette courbe de service nous permet de calculer une borne sur les délais subis par f_1. En suivant le Lemme 6.42.

$\mathrm{hDev}(\alpha_1, \beta_{1_{E2E}})$: `f1_E2E_delay:= hdev(alpha1, beta_1_E2E)` **Commande 10**

L'interpréteur nous donne alors une valeur numérique nous disant que : `f1_E2E_delay` $= 3612/3125$. Donc, d'après l'interpréteur de commande et avec les entrées du problème que nous avons décrites, nous avons

$$d_1 \leqslant 1,15584 \text{ s}$$

L'objectif de ce chapitre est de trouver des mécanismes et critères qui nous permettront de vérifier un tel résultat. Cela aura pour but d'augmenter notre confiance dans cette borne. En effet, l'expression théorique de cette borne a déjà été démontrée et formalisée mais son application sur une valeur concrète se base sur des outils externes.

7.2 Définitions des classes de fonctions

Nous allons dans cette partie donner la formalisation des fonctions introduites en début de ce chapitre. Nous avons placé l'ensemble des traductions en Coq de ces définitions et propriétés dans la branche `nfm_21_submissions`, à l'adresse suivante :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/nfm_21_submission`

Nous donnerons ensuite au cas par cas les fichiers Coq dans lesquels les formalisations se trouvent.

7.2.1 Fonctions ultimement pseudo périodique (UPP)

Nous allons dans cette partie présenter la définition formelle de l'ensemble des fonction ultimement pseudo périodiques (UPP). Cette traduction se trouve dans le fichier `UPP.v`.

Définition 7.1 (Fonctions UPP, `F_UPP`) $\mathcal{F}_{UPP}$ est l'ensemble des fonctions $f \in \mathcal{F}$ tel qu'il existe $T \in \mathbb{Q}_+$, $d \in \mathbb{Q}_+^\star$ et $c \in \mathbb{Q}$, respectivement appelé segment initial, période et incrément pour lesquels

$$\forall t \in \mathbb{R}_+, t \geqslant T \implies f(t+d) = f(t) + c. \tag{7.3}$$

La traduction en Coq utilise un `Record`. Nous avons :

```
1  Record F_UPP := {
2    F_UPP_val :> F;
3    F_UPP_T : Q+ ; F_UPP_d : Q+ *; F_UPP_c : Q;
4    _ : ∀ t : R+ , toR F_UPP_T%:nngnum ⩽ t%:nngnum →
5       (F_UPP_val (t%:nngnum + toR F_UPP_d%:num)%R%:nng = F_UPP_val t + (toR F_UPP_c)%:E)%E
6  }
```

qui signifie qu'une valeur du type `F_UPP` contient :

ligne 2 une fonction `F_UPP_val` de type `F`

ligne 3 `F_UPP_T`, `F_UPP_d` et `F_UPP_c`, les trois paramètres T, d et c de (7.3).

lignes 4 et 5 une preuve de la propriété (7.3). Nous utilisons `toR` pour transformer un rationnel en réel.

La commande `Record` crée un constructeur de `F_UPP` appelé `Build_F_UPP`. Pour déclarer une valeur de `F_UPP`, Coq requiert une fonction, trois paramètres et la preuve de (7.3).

Remarque 7.1 (*Valeurs dans $\mathbb{Q}_+$ pour la Définition 7.1*) Les valeurs de T, d et c auraient pu être dans $\mathbb{R}$. Cependant, nous savons de [BT08] que $\mathcal{F}_{UPP}$ est stable par plus d'opérateurs si T, d et c sont rationnels. Ce n'est pas une restriction puisqu'en en pratique, l'ensemble $\mathbb{Q}$ est l'ensemble utilisé pour l'implémentation.

Une propriété vient à la suite de cette définition. Elle permet d'étendre une fonction UPP avec un entier naturel k. En effet, il est possible d'étendre k fois la période d'une fonction UPP, nous arrivons alors sur cette fonction plus k fois l'incrément.

Lemme 7.1 (`UPP_extension_prop`) Pour tout $f \in \mathcal{F}_{UPP}$ avec T, d et c comme dans la précédente définition, pour tout $k \in \mathbb{N}$ et pour tout T' tel que $T \leqslant T'$, nous avons :

$$\forall t, T' \leqslant t \implies f(t + kd) = f(t) + kc.$$

Nous définirons en suivant dans ce chapitre des opérations sur l'ensemble $\mathcal{F}_{UPP}$. Nous établirons avec ces opérations des propriétés de stabilité.

7.2.2 UPP et fonctions affines par morceau

Nous avons brièvement introduit en début de ce chapitre l'ensemble $\mathcal{F}_{UPP-PA}$. Il s'agit des fonctions qui sont à la fois UPP et PA. Nous donnons ici une définition plus formelle.

Dans [BT08], cet ensemble est définie comme l'intersection des ensembles $\mathcal{F}_{UPP}$ et l'ensemble des fonctions PA. Nous choisissons de formaliser directement le sous ensemble des fonctions dans $\mathcal{F}_{UPP}$ qui sont PA. Cela nous permettre de faciliter la formalisation de l'ensemble de fonctions UPP-PA.

Pour définir les fonctions PA, nous avons besoin de formaliser la liste des points de discontinuités et changement de pentes. Nous appelons une telle liste une séquence de sauts ou *jump sequences* (JS). La traduction de cette partie en Coq se trouve dans le fichier `jump_sequences.v`.

Définition 7.2 (Séquence de sauts, `JS`) Pour tout $n \in \mathbb{N}^*$, nous appelons la *Séquence de saut* (JS) le tuple $a \in \mathbb{Q}_+^n$ tel que $a_0 = 0$ et : $\forall i \in \{0, \dots, n-2\}, a_i < a_{i+1}$. Nous appelons n la *taille* de la JS et l'ensemble des JS de taille n est noté JS_n.

Le nom donné à cette liste provient de l'utilisation que nous en aurons et de leur définition initiale dans [BT08]. En effet, nous utiliserons la *séquence de saut* pour enregistrer les sauts entre morceaux de fonctions pour une fonction affine par morceau. La traduction en Coq de cette définition est :

```
Record JS := {
  JS_list :> seq Q+ ;
  _ : (JS_list != [::]) && (head 0 JS_list == 0) && sorted < JS_list }.
```

Une `JS` est une liste `JS_list` de $\mathbb{Q}_+$. Nous ajoutons les propriétés ensuite dictées dans la définition : ce n'est pas une liste vide (dénoté avec [::]), dont l'élément initial est 0 et qui est triée (`sorted`) par l'ordre strict $<$. La fonction `head` est une fonction totale : elle retourne le premier élément de la liste ou une valeur par défaut si celle ci est vide, ici 0.

Chaque séquence est linéaire sur un intervalle avec une pente et un décalage. Nous donnons dans la prochaine définition une fonction pour modéliser cette partie.

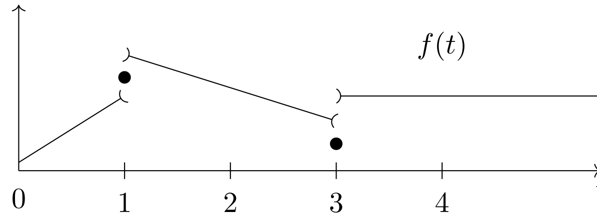


FIGURE 7.3 – La fonction f est une fonction affine par morceau. $a \triangleq \{0,1,3\}$ et $b \triangleq \{0,1,2,3\}$ sont des JS de cette fonction : $a \in JS(f)$ et $b \in JS(f)$. Notons que nous avons $c \triangleq \{0,2,4\} \in JS$ mais $c \notin JS(f)$.

Définition 7.3 ((ρ, σ)-*affine* sur, `r_s_affine_on`) Pour $\rho, \sigma \in \mathbb{Q}$ et $x, y \in \mathbb{Q}_+$, une fonction $f \in \mathcal{F}$ est appelée (ρ, σ)-*affine* sur $]x; y[$ si, pour tout $t \in]x; y[$:

$$f(t) = \rho(t - x) + \sigma. \tag{7.4}$$

Nous traduisons cette partie avec :

```
Definition r_s_affine_on (f : F) (rho sigma : Q) (x y : Q+ ) :=
  ∀ t : R+ , x < t < y → f t = toR rho ∗ (t -x) + toR sigma.
```

Nous voulons définir un sous ensemble de $\mathcal{F}$. Donc, nos fonctions peuvent prendre des valeurs infinies. Pour régler ce point, nous étendons la définition précédente.

Définition 7.4 (Affine sur, `affine_on`) Une fonction $f \in \mathcal{F}$ est *affine sur* $]x; y[$ si

$$(\forall t \in]x; y[\,, f(t) = +\infty) \tag{7.5}$$
$$\vee (\forall t \in]x; y[\,, f(t) = -\infty) \tag{7.6}$$
$$\vee (\exists \rho, \sigma \in \mathbb{Q}, f \text{ est } (\rho,\, \sigma)\text{-}\textit{affine} \text{ sur }]x; y[). \tag{7.7}$$

Nous traduisons cette partie en Coq avec :

```
Variant affine_on (f : F) (x y : Q+ ) :=
  | affine_on_p_infty of ∀ t : R+ , x < t < y → f t = +∞
  | affine_on_m_infty of ∀ t : R+ , x < t < y → f t = −∞
  | affine_on_finite rho sigma of r_s_affine_on f rho sigma x y.
```

Nous utilisons `Variant` qui est une version disjonctive de `Record`.

Les fonctions PA sont donc des fonctions qui sont *affine sur* tout les intervalles d'une JS.

Définition 7.5 (JS pour une fonction, `JS_of`) Soit $n \in \mathbb{N}^\star$, $a \in JS_n$ et $f \in \mathcal{F}$. Nous disons que a est une JS de f, dénoté $a \in JS(f)$, si pour tout $i < n - 1$, on a f *affine sur* $]a_i; a_{i+1}[$.

La traduction en Coq nous donne :

```
Definition JS_of a (f : F) :=
  ∀ i, (i.+ 1 < size a) → r_s_affine_on f (nth 0 a i) (nth 0 a i.+ 1).
```

Donc, d'après la définition précédente, chaque fonction PA est associée à une JS mais celle ci n'est pas unique. Nous donnons une illustration de cet aspect dans la Figure 7.3. Notons aussi qu'une fonction $f \in \mathcal{F}$ avec $a \in JS(f)$ est une fonction PA au moins jusqu'au dernier point de a.

Définition 7.6 (Fonctions UPP-PA, `F_UPP_PA`) L'ensemble $\mathcal{F}_{UPP-PA}$ des fonctions UPP-PA est l'ensemble des fonctions $f \in \mathcal{F}_{UPP}$ avec T pour segment initial et d pour période, tel qu'il existe $a \in JS(f)$ et $last(a) = T + d$.

Nous représentons $\mathcal{F}_{UPP-PA}$ en Coq comme suit :

```
Record F_UPP_PA := {
  F_UPP_PA_UPP :> F_UPP;
  F_UPP_PA_JS : JS;
  _ : JS_of F_UPP_PA_JS F_UPP_PA_UPP;
  _ : last 0 F_UPP_PA_JS = F_UPP_T F_UPP_PA_UPP + F_UPP_d F_UPP_PA_UPP }.
```

La fonction présentée dans la Figure 7.1 appartient à l'ensemble $\mathcal{F}_{UPP-PA}$. La liste des abscisses des discontinuités donnée en description de cette figure correspond à une JS de cette fonction.

Une fonction UPP-PA avec un segment initial T et une période d est PA dans $[0; T+d[$ par construction, et aussi PA après $T+d$ par périodicité. Ce point est montré dans la prochaine propriété.

Lemme 7.2 (`F_UPP_PA_JS_upto_spec`) Soit $f \in \mathcal{F}_{UPP-PA}$ avec $a \in JS(f)$. Pour tout $l \in \mathbb{Q}_+$ tel que $last(a) \leqslant l$, il existe $a' \in JS$ tel que $a' \in JS(f)$ et $last(a') = l$.

Nous avons défini dans cette section l'intersection des fonctions qui sont UPP et qui sont PA. Afin de pouvoir formaliser en Coq les opérations utilisées en *Calcul réseau*, nous allons devoir démontrer que ces ensembles de fonctions sont stables par ces opérations.

7.3 Stabilités de $\mathcal{F}_{UPP}$ par les opérations *(min, +)*

Nous voulons maintenant prouver la stabilité de $\mathcal{F}_{UPP}$ sur les opérations *(min, +)*. Nous allons dans cette partie formaliser l'addition, le minimum et la convolution. Nous avons besoin d'un nouvel opérateur sur les nombres rationnels : une notion de plus petit commun multiple $lcm_{\mathbb{Q}_+^\star}$ tel que, pour tout $d, d' \in \mathbb{Q}_+^\star$, il existe $k, k' \in \mathbb{N}$ satisfaisant $kd = k'd' = lcm_{\mathbb{Q}_+^\star}(d, d')$.

Définition 7.7 (***$lcm_{\mathbb{Q}_+^\star}$***) Pour tout $d, d' \in \mathbb{Q}_+^\star$, pour tout $a, a' \in \mathbb{Z}$ et $b, b' \in \mathbb{N}^\star$ tel que $d = \frac{a}{b}$ et $d' = \frac{a'}{b'}$, nous définissons

$$lcm_{\mathbb{Q}_+^\star}(d, d') \triangleq \frac{\operatorname{lcm}\left(a\frac{\operatorname{lcm}(b,b')}{b}, a'\frac{\operatorname{lcm}(b,b')}{b'}\right)}{\operatorname{lcm}(b, b')} \tag{7.8}$$

où lcm est le plus petit multiple commun sur $\mathbb{Z}$.

Nous pourrions nous débarrasser des rationnels et nous ramener à un problème n'impliquant que des entiers en multipliant toutes les données par le plus petit multiple de tous les dénominateurs en jeu. Toutefois, dans notre cas, cette simplification ne nous arrange pas. En effet, faire un lcm sur toutes les fonctions de contraintes du réseau peut donner des résultats plus grands qu'en faisant un lcm sur ces fonctions une à une. Par exemple, pour trois fonctions f, g et h de $\mathcal{F}_{UPP-PA}$, le calcul de X comme le lcm des périodes de f et g et le calcul du lcm entre X et h est peut-être plus petit que le calcul du lcm des périodes de f, g et h.

Lemme 7.3 (`dvdq_lcml`) Pour tout $d, d' \in \mathbb{Q}_+$, il existe $k \in \mathbb{N}$ tel que $lcm_{\mathbb{Q}_+^\star}(d, d') = k\,d$.

Nous traduisons cette définition et ce lemme en Coq avec :

```
Definition lcm_Q (d d' : Q) : Q :=
  fracq (lcmz (numq d * (lcmz (denq d) (denq d') %/ denq d))
              (numq d' * (lcmz (denq d) (denq d') %/ denq d')),
        lcmz (denq d) (denq d')).
Program Definition lcm_posQ (d d' : Q+ *) : Q+ * := mk_posQ (lcm_Q d d') _.
Lemma dvdq_lcml d d' : ∃ k : nat, lcm_posQ d d' = k * d.
```

Nous définissons premièrement `lcm_Q` : la définition de $lcm_{\mathbb{Q}_+^\star}$ sur l'ensemble $\mathbb{Q}$. Les fonctions `fracq`, `numq` et `denq` sont respectivement des constructeurs et destructeurs de `Q`. Comme utilisé précédemment, la commande `Program Definition` nous permet de donner la preuve après la déclaration de définition.

Pour faciliter les notations, nous voulons transformer cet opérateur binaire en une opération d'ensemble. L'objectif ici est de pouvoir utiliser la librairie de `MathComp` donnée dans le fichier `bigop.v`. Comme nous l'avons fait pour des opérations binaires pour utiliser complètement cette librairie (par exemple l'opération d'addition pour $\mathcal{F}$ page 50), nous devons prouver que $lcm_{\mathbb{Q}_+^\star}$ correspond à la définition d'un monoide : un ensemble muni d'une opération associative et d'un élément neutre pour cette opération (cf. Définition 5.1).

Pour cela, nous devons montrer que $lcm_{\mathbb{Q}_+^\star}$ est associatif et possède un élément neutre. Cependant, $lcm_{\mathbb{Q}_+^\star}$ n'a pas d'élément neutre. Le plus petit multiple commun sur $\mathbb{N}$ a un élément neutre 1. Ce n'est pas le cas pour $lcm_{\mathbb{Q}_+^\star}$: par exemple $lcm_{\mathbb{Q}_+^\star}\left(1, \frac{2}{3}\right) = 2$.

Pour pouvoir régler ce problème, nous devons étendre la définition de $lcm_{\mathbb{Q}_+^\star}$:

```
Definition olcm_posQ (x y : option Q+ *) : option Q+ * := match x, y with
 | None, _ ⇒ y | _, None ⇒ x | Some x, Some y ⇒ Some (lcm_posQ x y)
 end.
```

Le type `option` est utilisé pour étendre le type de `Q+ *` avec l'élément `None`. Alors, cet élément est un élément neutre pour cette définition optionnelle de $lcm_{\mathbb{Q}_+^\star}$. Nous ajoutons ensuite une `Notation` pour l'opération d'ensemble

```
Notation "\biglcm_posQ_ ( i < n ) F" :=
  (odflt one_posQ (\big[olcm_posQ/None]_(i < n) some F)) : ring_scope.
```

`\big[oclm_posQ\None]_(i < n)some F` est l'application itérée de `oclm_posQ` pour tout `i` tel que `i < n` sur `some F`. La fonction `odflt` enlève l'`option` quand le résultat est `Some` et retourne une valeur par défaut sinon.

Les prochaines propriétés démontrent la stabilité de $\mathcal{F}_{UPP}$ par addition, minimum et convolution.

Lemme 7.4 (`F_UPP_n_add`) Soit $n \in \mathbb{N}^\star$, $f \in \mathcal{F}_{UPP}^n$ avec comme segments initiaux $T \in \mathbb{Q}_+^n$, des périodes $d \in (\mathbb{Q}_+^\star)^n$ et des incréments $c \in \mathbb{Q}^n$ respectivement. La somme $\sum_i f_i$ est une fonction UPP avec comme segment initial $\max_i\{T_i\}$, comme période $\underset{i}{lcm_{\mathbb{Q}_+^\star}}(d_i)$ et comme incrément $\underset{i}{lcm_{\mathbb{Q}_+^\star}}(d_i)\left(\sum_i \frac{c_i}{d_i}\right)$.

Lemme 7.5 (`F_UPP_n_min`) Soit $n \in \mathbb{N}^\star$ et $f \in \mathcal{F}_{UPP}^n$ avec comme segments initiaux $T \in \mathbb{Q}_+^n$, des périodes $d \in (\mathbb{Q}_+^\star)^n$ et des incréments $c \in \mathbb{Q}^n$ respectivement. Nous définissons :

$$s \triangleq \min_{i \in [0;n-1]} \left(\frac{c_i}{d_i}\right) \qquad I \triangleq \left\{ i \in [0;n-1] \middle| \frac{c_i}{d_i} = s \right\} \tag{7.9}$$

et supposons qu'il existe $M \in \mathbb{Q}$ et $m \in \mathbb{Q}^n$ tel que :

$$\exists i \in I, \forall t \in [T_i; T_i + d_i[\,, f_i(t) \leqslant M + s\,t \tag{7.10}$$

$$\forall i \notin I, \forall t \in [T_i; T_i + d_i[\,, m_i + \frac{c_i}{d_i} t \leqslant f_i(t) \tag{7.11}$$

la fonction $\min_{i=1}^n\{f_i\}$ est UPP avec comme segment initial $\tilde{T}$, comme période $\tilde{d}$ et comme incrément $\tilde{c}$ avec $\tilde{d} \triangleq \underset{i \in I}{lcm_{\mathbb{Q}_+^\star}}(d_i)$, $\tilde{c} \triangleq \tilde{d}\, s$ et

$$\tilde{T} = \max\left(\max_{i \notin I}\left(\frac{M - m_i}{\frac{c_i}{d_i} - s}\right), \max_{j \in [0;n-1]}\{T_j\}\right).$$

Ces propriétés sont des généralisations directes de la Proposition 6 de [BT08], dans lequel cette stabilité était prouvée pour les opérations binaires d'addition et de minimum. Cette généralisation sera

utile pour une prochaine propriété sur la convolution de deux fonctions UPP.

> **Remarque 7.2** (*Trouver des valeurs de M et m_i*) Dans le cas de fonctions PA, nous pourrons trouver des valeurs de M et m_i qui satisfassent les relations (7.10) et (7.11) en calculant les bornes : $\sup_{t\in[T_i;T_i+d_i[} \{f_i(t) - s\,t\}$ et $\inf_{t\in[T_i;T_i+d_i[} \left\{f_i(t) - \frac{c_i}{d_i}t\right\}$.

Lemme 7.6 (`F_UPP_conv`) Soit $f, f' \in \mathcal{F}_{UPP}$ avec comme segment initiaux $T, T' \in \mathbb{Q}_+$, des périodes $d, d' \in \mathbb{Q}_+^\star$ et des incréments $c, c' \in \mathbb{Q}$ respectivement. Pour tout $M, M', m, m' \in \mathbb{Q}$ tel que

$$M \geqslant \sup_{t\in[T,T+d[} \left\{f(t) - \frac{c}{d}(t+T')\right\} + f'(T') \tag{7.12}$$

$$m' \leqslant \inf_{t\in[0,T[} \left\{f(t) - \frac{c'}{d'}t\right\} + \inf_{t\in[T',T'+d'[} \left\{f'(t) - \frac{c'}{d'}t\right\} \tag{7.13}$$

et similairement pour M' et m, en permutant les valeurs primées et non primées, la convolution $f * f'$ est une fonction UPP avec une période $\tilde{d} \triangleq lcm_{\mathbb{Q}_+^\star}(d, d')$, un incrément $\tilde{c} \triangleq \tilde{d}\min\left(\frac{c}{d}, \frac{c'}{d'}\right)$ et un segment initial

$$\tilde{T} = \begin{cases} T + T' + lcm_{\mathbb{Q}_+^\star}(d, d') & \text{si } \frac{c}{d} = \frac{c'}{d'} \\ \max\left(\frac{M-m'}{\frac{c'}{d'}-\frac{c}{d}}, T + T' + lcm_{\mathbb{Q}_+^\star}(d, d')\right) & \text{si } \frac{c}{d} < \frac{c'}{d'} \\ \max\left(\frac{M'-m}{\frac{c}{d}-\frac{c'}{d'}}, T + T' + lcm_{\mathbb{Q}_+^\star}(d, d')\right) & \text{si } \frac{c'}{d'} < \frac{c}{d} \end{cases} \tag{7.14}$$

Ce lemme permet aussi de généraliser la Proposition 6 de [BT08] en donnant cette fois ci une valeur pour $\tilde{T}$. La Remarque 7.2 s'applique aussi pour cette propriété.

7.4 Stabilités de $\mathcal{F}_{UPP-PA}$ par les opérations *(min, +)*

Nous nous intéressons maintenant à la stabilité de $\mathcal{F}_{UPP-PA}$ sur des opérations *(min, +)*. Nous devons pour cela définir plusieurs opérations, notamment sur les séquences de saut. Nous allons dans un premier temps définir l'union de deux JS.

Définition 7.8 (Union de deux JS, `union`) Pour tout $n, m \in \mathbb{N}^*, a \in JS_n, b \in JS_m$, le tuple de taille $\#(\{a_i|0 \leqslant i < n\} \cup \{b_j|0 \leqslant j < m\})$ contenant les éléments de $\{a_i|0 \leqslant i < n\} \cup \{b_j|0 \leqslant j < m\}$ triés par ordre croissant, est appelé *union* des séquences a et b. Cette union est notée $a \cup b$.

Nous utilisons la notation $\#s$ pour exprimer le cardinal d'une séquence s. Si les séquences de saut sont implémentées par des listes, l'union peut être implémentée par la fusion du tri fusion suivie d'une suppression des éléments en double.

La prochaine propriété donne une séquence de saut pour la somme de fonctions PA.

Lemme 7.7 (JS pour l'addition n-aire, `JS_of_n_add`) Pour tout $n \in \mathbb{N}^\star$, $f \in \mathcal{F}^n$ et $a \in JS^n$, si pour tout i, $a_i \in JS(f_i)$ et tous les derniers points de a sont égaux ($\forall i, j, last(a_i) = last(a_j)$), alors nous avons $\bigcup_i a_i \in JS\left(\sum_i f_i\right)$.

Nous traduisons ce lemme en Coq de la manière suivante :

```
Lemma JS_of_n_add n (f : 'I_n.+1 → F) (a : 'I_n.+1 → JS) :
  (∀ i, JS_of (a i) (f i)) → (∀ i j, last 0 (a i) = last 0 (a j)) →
  JS_of (\bigcup_i a i) (\sum_i f i).
```

Le terme `\bigcup_i` est une notation pour $\bigcup_i$. Grâce au Lemme 7.2, l'égalité entre les derniers points

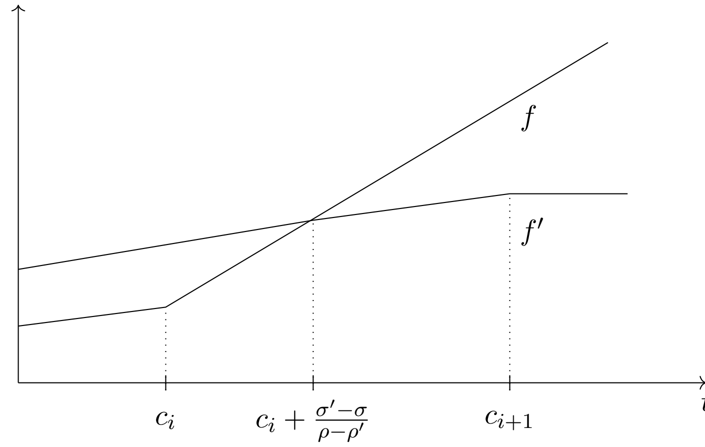


FIGURE 7.4 – Exemple de point ajouté par l'opération min sur des JS. f et f' sont respectivement (ρ, σ)-*affine* et (ρ', σ')-*affine* sur $]c_i; c_{i+1}[$ avec des pentes différentes ρ etρ'. Vu que le point $c_i + \frac{\sigma'-\sigma}{\rho-\rho'} \in]c_i; c_{i+1}[$, celui ci doit être ajouté à la séquence de saut de la fonction $\min(f, f')$.

peut toujours être obtenue. La stabilité de $\mathcal{F}_{UPP-PA}$ par l'addition n-aire peut être dérivée à partir de cette dernière propriété et des lemmes 7.2 et 7.4.

Tandis que la séquence de sauts pour l'addition est l'union de ces séquences, le minimum entre deux fonctions PA peut introduire de nouveaux points. Nous donnons un exemple de cet aspect dans la Figure 7.4. Pour correspondre avec l'exemple donné, nous donnons la prochaine définition qui formalise ces nouveaux points.

Définition 7.9 (`union_min`) Soit f et $f' \in \mathcal{F}$ avec $a \in JS(f)$ et $a' \in JS(f')$ tel que $last(a) = last(a')$. Posons $c \triangleq a \cup a'$. Nous définissons l'opération $\cup_{\min}$ comme

$$\cup_{\min}(f, f', a, a') \triangleq c \cup \left\{ c_i + \frac{\sigma' - \sigma}{\rho - \rho'} \middle| \begin{array}{l} \exists i, i < \#c - 1 \\ \wedge f \text{ est } (\rho, \sigma) - \textit{affine} \text{ sur }]c_i, c_{i+1}[\\ \wedge f' \text{ est } (\rho', \sigma') - \textit{affine} \text{ sur }]c_i, c_{i+1}[\\ \wedge \rho \neq \rho' \wedge c_i < c_i + \frac{\sigma'-\sigma}{\rho-\rho'} < c_{i+1}. \end{array} \right\} \tag{7.15}$$

En utilisant l'opération $\cup_{\min}$, nous pouvons établir comme pour l'addition une JS pour le minimum n-aire.

Lemme 7.8 (`JS_of_n_min`) Pour tout $n \in \mathbb{N}^\star$ et $f \in \mathcal{F}^n$, si pour tout i, $a_i \in JS(f_i)$ et tout les derniers points de a sont égaux, alors nous avons :

$$\left(\bigcup_{i,j \in [0,n-1]} \cup_{\min}(f_i, f_j, a_i, a_j) \right) \in JS\left(\min_i \{f_i\} \right). \tag{7.16}$$

Comme nous l'avons mentionné pour l'addition, cette propriété et les lemmes 7.2 et 7.5 sont suffisants pour démontrer la stabilité de $\mathcal{F}_{UPP-PA}$ par le minimum n-aire sous de légères conditions. En effet, il faut que les points m et M du Lemme 7.5 existent.

Nous sommes maintenant intéressé à la convolution de deux fonctions UPP-PA. Comme dans [BT08], nous allons utiliser la propriété qui permet de développer la convolution :

$$\forall f, g, h \in \mathcal{F}, \min(f, g) * h = \min(f * h, g * h).$$

Il se trouve que cette propriété à déjà été traduite en Coq au cours de la partie Instances $(\mathcal{F}, \min, *)$: le *dioïde* des fonctions *(min, +)*, dans le Chapitre 5.

Or, toute fonction UPP-PA peut être décomposée comme le minimum de fonctions élémentaires dont la convolution est plus facile à calculer. Dans la suite de cette partie, nous donnons une telle décomposition.

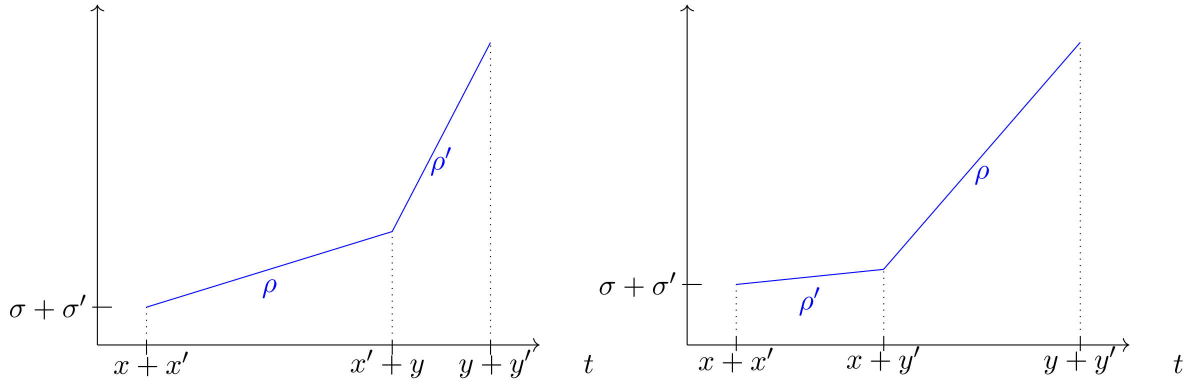


FIGURE 7.5 – Convolution de deux segments. Soit f et f' deux fonctions qui sont respectivement (ρ, σ)-*affine* sur $[x; y[$ et (ρ', σ')-*affine* sur $[x'; y'[$ et valent $+\infty$ ailleurs. Nous avons tracé les deux cas de $f * f'$ sur $[x + x', y + y'[$: à gauche pour $\rho < \rho'$ et à droite pour le cas $\rho' < \rho$.

Définition 7.10 (Opération de découpe, `cutting_operator`) Soit $f \in \mathcal{F}$, $a \in JS(f)$ et $i \in \mathbb{N}$ tel que $i < \#a - 1$, nous définissons l'opération de *découpe* comme

$$(f \downarrow a)_i \triangleq t \mapsto \begin{cases} f(t) & si\ t \in [a_i; a_{i+1}[\\ +\infty & sinon\ . \end{cases}$$

La traduction en Coq nous donne :

```
Definition cutting_operator (f : F) a i : F := fun t ⇒
  if i.+ 1 < size a && (nth 0 a i ⩽ t < nth 0 a i.+ 1) then f t else +∞ .
```

La convolution de deux segments $(f \downarrow a)_i$ et $(f' \downarrow a')_j$ est donc calculée à partir d'une disjonction de cas comme nous l'avons fait dans la Figure 7.5. Cependant, nous allons considérer les discontinuités que peuvent avoir ces fonctions et cela va nous mener à devoir traiter plus de deux cas.

Nous avons besoin d'une dernière définition pour spécifier la convolution.

Définition 7.11 (`cutting_below`) Soit $f \in \mathcal{F}$ et $l \in \mathbb{R}_+$, nous notons $f_{<l}$ la fonction qui est égale à f jusqu'à l et vaut $+\infty$ après.

Lemme 7.9 (`cutting_operator_spec`) Soit $f \in \mathcal{F}$ et $a \in JS(f)$, nous avons $f_{<last(a)} = \min_{i<\#a-1} (f \downarrow a)_i$.

Nous pouvons maintenant donner la décomposition de la convolution en utilisant ces opérations.

Lemme 7.10 (Convolution de fonctions PA, `PA_conv`) Soit $f, f' \in \mathcal{F}$ avec $a \in JS(f)$ et $a' \in JS(f')$ et soit l tel que $l = last(a) = last(a')$. Nous avons

$$(f * f')_{<l} = \left(\min_{i,j} \left((f \downarrow a)_i * (f' \downarrow a')_j\right)\right)_{<l} . \tag{7.17}$$

7.5 Vérification des opérations sur $\mathcal{F}_{UPP-PA}$

Nous allons dans cette partie donner les critères que nous avons développé pour vérifier le calcul d'opérations *(min, +)*. Dans un premier temps, nous donnons des critères d'égalité fini. En effet, nous sommes capables de vérifier l'égalité sur les opérations d'addition, de minimum et de convolution dans $\mathcal{F}_{UPP-PA}$.

Dans une seconde sous partie, nous allons expliquer comment nous allons vérifier les opérations sur lesquelles nous n'avons pas de critère d'égalité. En effet, nous avons pour les opérations de *(min, +)* manquantes, des critères majorants ces opérations.

7.5.1 Critère d'égalité fini sur $\mathcal{F}_{UPP-PA}$

Dans les section 7.2.1 à 7.4, nous avons prouvé en Coq quelques (extensions de) résultats de la littérature. Nous allons ici étudier l'objectif de ce chapitre : trouver des critères de test pour vérifier des égalités.

Cela commence par donner un critère d'égalité commun.

Définition 7.12 (Égalité sur un segment, `eq_segment`) Pour tout $a \in JS, i \in \mathbb{N}$ et $f, g \in \mathcal{F}$, nous définissons $eq_segment(a, i, f, g)$, la propriété suivante :

$$f(a_i) = g(a_i) \land \exists x, y \in \left]a_i; a_{i+1}\right[, x \neq y \land f(x) = g(x) \land f(y) = g(y). \tag{7.18}$$

Que nous avons traduit par :

```
Definition eq_segment (a : JS) i (f g : F) :=
  f (a i) = g (a i)
  ∧ ∃ x y : R+ , a i < x < a i.+ 1 ∧ a i < y < a i.+ 1 ∧ x ≠ y ∧ f x = g x ∧ f y = g y.
```

Cette définition est utile pour vérifier l'égalité sur un intervalle. En effet, pour deux fonctions f et g toutes les deux *affines* sur $]a_i; a_{i+1}[$, $eq_segment(a, i, f, g)$ permet d'assurer que $f = g$ sur $[a_i; a_{i+1}[$.

En combinant les résultats de ce chapitre, nous pouvons donner un critère d'égalité sur l'addition n-aire.

Proposition 7.1 (`UPP_PA_n_add`) Soit $n \in \mathbb{N}^\star$, $f \in \mathcal{F}^n_{UPP-PA}$, $f' \in \mathcal{F}_{UPP-PA}$ avec comme segments initiaux $T \in \mathbb{Q}^n_+$ et $T' \in \mathbb{Q}_+$, des périodes $d \in (\mathbb{Q}^\star_+)^n$ et $d' \in \mathbb{Q}^\star_+$, et des incréments $c \in \mathbb{Q}^n$ et $c' \in \mathbb{Q}$ respectivement. Définissons $l \triangleq \max\left(\max_i\{T_i\}, T'\right) + lcm_{\mathbb{Q}^\star_+}\left(\underset{i}{lcm_{\mathbb{Q}^\star_+}}(d_i), d'\right)$, et $u \triangleq \left(\bigcup_i a_i\right) \cup a'$. Si pour tout i, $a_i \in JS(f_i)$ et $last(a_i) = l$, $a' \in JS\left(f'\right)$ et $last(a') = l$ et si $\sum_i \left(\frac{c_i}{d_i}\right) = \frac{c'}{d'}$ alors

$$\forall i < \#u - 1, eq_segment\left(u, i, \sum_j f_j, f'\right) \tag{7.19}$$

est une condition suffisante pour $\sum_i f_i = f'$.

Il se trouve que cette condition est aussi nécessaire, ce qui est trivial à prouver mais ne présente aucun intérêt dans notre cas. En effet, connaissant que les fonctions sont égales en tout point, elles le sont en particulier pour chaque point vérifié par *eq_segment*.

> **Remarque 7.3** (*Temps de calcul fini*) Ce critère peut être calculé en un temps fini. Les a_i et a' peuvent être obtenues en utilisant le Lemme 7.2. Pour vérifier $eq_segment(a, i, f, f')$, nous pouvons prendre par exemple les points $x = \frac{a_i + a_{i+1}}{2}$ et $y = \frac{a_i + x}{2}$.

Nous avons des critères similaires pour le minimum et la convolution.

Proposition 7.2 (`UPP_PA_n_min`) Soit $n \in \mathbb{N}^\star$ et $f \in \mathcal{F}^n_{UPP-PA}$. Pour tout $f' \in \mathcal{F}_{UPP-PA}$ avec comme segment initial $T' \in \mathbb{Q}_+$, comme période $d' \in \mathbb{Q}^\star_+$ et comme incrément $c' \in \mathbb{Q}$, supposons que nous avons M et m satisfaisant les hypothèses du Lemme 7.5 et définissons $\tilde{T}, \tilde{d}$ et $\tilde{c}$ comme dans le Lemme 7.5. Nous définissons $l \triangleq \max(\tilde{T}, T') + lcm_{\mathbb{Q}^\star_+}(\tilde{d}, d')$ et $u \triangleq \left(\bigcup_{i,j} \cup_{\min}\left((f_i)_{<l}, (f_j)_{<l}, a_i, a_j\right)\right) \cup a'$, où pour tout i, $a_i \in JS\left(f_i\right)$ et $last(a_i) = l$, $a' \in JS\left(f'\right)$ et $last(a') = l$. Si $\frac{\tilde{c}}{\tilde{d}} = \frac{c'}{d'}$, alors :

$$\forall i < \#u - 1, eq_segment\left(u, i, \min_j(f_j), f'\right) \tag{7.20}$$

est une condition suffisante pour $\min_i(f_i) = f'$.

Proposition 7.3 (`F_UPP_conv`) Soit $f, f' \in \mathcal{F}_{UPP-PA}$. Pour tout $f'' \in \mathcal{F}_{UPP-PA}$ avec comme segment initial $T'' \in \mathbb{Q}_+$, une période $d'' \in \mathbb{Q}_+$ et un incrément $c'' \in \mathbb{Q}$, supposons que nous avons M, M', m et $m' \in \mathbb{Q}$ satisfaisant les hypothèses du Lemme 7.6 et définissons $\tilde{T}, \tilde{d}, \tilde{c}$ comme dans le Lemme 7.6. Définissons $l \triangleq \max(\tilde{T}, T'') + lcm_{\mathbb{Q}_+^*}(\tilde{d}, d'')$. Supposons que nous avons $a \in JS(f)$, $last(a) = l$, $a' \in JS(f')$ et $last(a') = l$, $a'' \in JS(f'')$ et $last(a'') = l$ et définissons $k \triangleq \#a - 1$ et $k' \triangleq \#a' - 1$. Soit $\tilde{a} \in JS^{\{0,\dots,k-1\}\times\{0,\dots,k'-1\}}$ tel que pour tout i, i', $\tilde{a}_{i,i'} \in JS\left((f \downarrow a)_i * (f' \downarrow a')_{i'}\right)$ et $last\left(\tilde{a}_{i,i'}\right) = l$, définissons

$$u \triangleq \left(\bigcup_{i,i',j,j'} \cup_{\min} \left((f \downarrow a)_i * \left(f' \downarrow a'\right)_{i'}, (f \downarrow a)_j * \left(f' \downarrow a'\right)_{j'}, \tilde{a}_{i,i'}, \tilde{a}_{j,j'} \right) \right) \cup a''. \tag{7.21}$$

Si $\frac{\tilde{c}}{\tilde{d}} = \frac{c''}{d''}$ alors

$$\forall j < \#u - 1, eq_segment\left(u, i, \min_{i,j}\left((f \downarrow a)_i * \left(f' \downarrow a'\right)_j\right), f''\right) \tag{7.22}$$

est une condition suffisante pour $f * f' = f''$.

Tout comme la Proposition 7.1, ces critères suffisants peuvent être vérifiés en temps fini.

7.5.2 Critère majorant

Il nous sera utile aussi de pouvoir vérifier les déviations horizontales et verticales entre deux fonctions. Dans ce cas là, nous n'avons pas besoin de donner un critère d'égalité : il est suffisant de donner une borne majorant la déviation. Ainsi, nous avons les deux prochaines propriétés.

Lemme 7.11 (`hDev_suff`) Pour tout $A \in \mathcal{F}^{\uparrow}$ et $D \in \mathcal{F}$, nous avons :

$$\forall d \in \mathbb{R}_+, A * \delta_d \leqslant D \implies \mathrm{hDev}(A, D) \leqslant d.$$

Lemme 7.12 (`vDev_suff`) Pour tout A et $D \in \mathcal{F}$ tel que pour tout t, $A(t) \neq -\infty$ et pour tout t, $D(t) \neq +\infty$, nous avons, pour tout $b \in \mathbb{R}_+$:

$$\forall t \in \mathbb{R}_+, A(t) \leqslant D(t) + b \implies \mathrm{vDev}(A, D) \leqslant b.$$

Pour la déconvolution, en pratique, le *Calcul réseau* nécessite, pour deux fonctions f et g, une fonction h tel que $h \geqslant f \oslash g$. Il est alors suffisant de vérifier que $h * g \geqslant f$ (cf. Lemme 5.33 page 44), qui est $\min(f, h * g) = f$. Ainsi, nous pouvons établir un critère de vérification que la fonction h est un majorant de la déconvolution entre f et g n'impliquant que le minimum et la convolution. Une telle propriété a été décrite dans le fichier `UPP_PA_refinement.v`.

7.6 Implémentation

L'implémentation de cette partie est constituée de 6.3k lignes de Coq. Elle utilise les nombres rationnels définis dans la librairie `MathComp` [MT21] et les nombres réels de la librairie standard de Coq [Sem20]. Ces nombres réels sont liés à la structure algébrique de `MathComp` avec le fichier `Rstruct .v` de la librairie `MathComp Analysis` [Rou19]. Ainsi, nous pouvons utiliser la librairie `MathComp` que nous avons déjà utilisée sur les opérations d'ensembles, amenée par [Ber+08].

Lors d'un premier développement, nous avions utilisé pour cette partie les réels étendus $\overline{\mathbb{R}}$ et d'autres définitions sur les réels de la librairie `Coquelicot` [BLM15]. Il a été possible par la suite de remplacer les nombres réels de la librairie standard et `Coquelicot` par ceux utilisés dans la librairie de `MathComp Analysis`. Cette librairie était en cours de développement lorsque ces travaux ont commencé. Les nombres réels de cette librairie sont cependant plus explicites grâce aux notations fortement présente et ils permettent de faciliter le passage entre les différentes librairies de `MathComp`.

Pour obtenir un programme Coq exécutable, nous avons fait certains ajustements à ces travaux. En effet, nous avons par exemple inclu les ρ et σ de la Definition 7.5 dans la Séquence de saut. Ainsi, elle devient une liste contenant non seulement les séquences de saut mais aussi les valeurs des offset σ et des pentes ρ. Ceci nous permet de ne plus avoir à réévaluer ces valeurs lors de la formalisation de certaine preuves. La version du code finalement exécutable est présente dans la branche `master` de notre code, dans le dossier portant le nom du projet : `minerve` :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/master/minerve`

Elle est constituée de 9k lignes de code Coq. Elle utilise la formalisation précédente ainsi que le raffinement des nombres rationnels de la librairie `bignums` [GT06] fourni par la librairie `CoqEAl` [CDM13].

7.7 Exemple sur le cas d'étude

Nous allons vérifier à l'aide du développement présenté dans ce chapitre les calculs du cas d'étude du Chapitre 6, dans la Section 6.4, et au début de ce chapitre, dans la Section 7.1.2. Le fichier correspondant à cette partie est dans le même dossier que le cas d'étude, `examples/case_study/` et se nomme `case.v` accessible par :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/blob/phd-lucien/`

Pour pouvoir exécuter ce code, il est nécessaire d'installer la version finale du développement. Il s'agit donc dans cette partie du sous dossier `minerve`. Toutes les commandes et librairies nécessaire sont expliquées en Annexe A (page 112).

Dans cette partie, nous avons commencé par déclarer la fonction `alpha1` avec la Commande 1. Cette fonction, comme définie dans l'équation (7.1), a un seul segment. Elle a un point de discontinuité en 0 valant 0 puis une pente de 20000 avec un offset de 8000. C'est donc une fonction de $\mathcal{F}_{UPP-PA}$ qui n'a pas besoin d'être décrite avec un préfixe et un motif répété. Nous disons familièrement que c'est une fonction de PA.

Pour décrire des fonctions de $\mathcal{F}_{UPP-PA}$ en Coq, nous avons développé la fonction `F_of_sequpp`. Cette fonction permet de donner les éléments décrivant une fonction de cet ensemble :

— `sequpp_T` pour le segment initial
— `sequpp_d` pour la période
— `sequpp_c` pour l'incrément
— `sequpp_js` pour la liste des segments

Ce dernier élément correspond à une extension de la description de séquence de sauts de la Définition 7.5. Maintenant, cette séquence est constitué de 4 valeurs par éléments. Ils permettent de décrire un point de discontinuité et un segment. Ce point de discontinuité est exprimé par une abscisse et une ordonnée. Le segment est décrit par une pente $\rho \in \mathbb{Q}$ et un offset $\sigma \in \overline{\mathbb{Q}}$. Ainsi, la fonction sur le segment prend la même valeur que dans la Définition 7.3 mais peut aussi valoir $+\infty$ ou $-\infty$.

Ainsi, nous décrivons la fonction `alpha1` avec la commande suivante :

```
Definition alpha1 := F_of_sequpp {|
  sequpp_T := 2;
  sequpp_d := 1;
  sequpp_c := 20_000;
  sequpp_js := (* list of (x, (y, (rho, sigma))) *)
            [:: (0, (0%:E, (20_000, 8_000%:E)))] |}%bigQ.
```

En Coq, nous pouvons utiliser un *underscore* (_) pour séparer les nombres. Nous notons 20_000 pour 20000, améliorant ainsi la lisibilité.

Le segment initial est quelconque : nous savons que la fonction est directement périodique et n'a donc pas de séquence initiale hors de ce cycle. Nous avons fixé une période de 1s pour avoir le même incrément que la pente de la fonction. Le segment est décri avec un point de discontinuité 0 en 0 puis une pente de 20000bit/s et un offset 8000bit.

Nous sommes dans le cas particulier où chaque courbe d'arrivée est identique donc nous décrivons ensuite :

```
Definition alpha2 := alpha1.
Definition alpha3 := alpha1.
Definition alpha4 := alpha1.
Definition alpha5 := alpha1.
```

Pour décrire la fonction `beta1` comme dans la Commande 2, nous avons :

```
Definition beta1 := F_of_sequpp {|
  sequpp_T := 0;
  sequpp_d := 1;
  sequpp_c := 100_000;
  sequpp_js := [:: (0, (0%:E, (100_000, 0%:E)))] |}%bigQ.
```

qui correspond à décrire une fonction PA avec une pente de 100000 et un offset nul. De même que pour les courbes d'arrivée, les courbes de service `beta2`, `beta3`, `beta4` et `beta5` sont égales à `beta1`.

Dans un premier temps, nous cherchons à calculer la fonction α_{S1} correspondant à la somme des arrivées sur le serveur S_1. Lors de l'exécution de la Commande 3, l'interpréteur a retourné la valeur suivante :

```
» uaf([(0,0)]](0,24000)60000(+Infinity,+Infinity)[)
```

Donc, la fonction `alpha_S1` est une fonction avec un segment avec un point de discontinuité 0 en 0, un offset de 24000, une pente de 60000, une période de 1s et un incrément de 60000bit. Nous la formalisons de la manière suivante :

```
Definition alpha_S1 := F_of_sequpp {|
  sequpp_T := 1;
  sequpp_d := 1;
  sequpp_c := 60_000;
  sequpp_js := [:: (0, (0%:E, (60_000, 24_000%:E)))] |}%bigQ.
```

Cette fonction a été obtenue par un calcul de l'interpréteur. Nous allons donc vérifier que ce résultat, donné par l'interpréteur, correspond bien à notre attente, c'est à dire :

```
* Goals *
Goal alpha_S1 = alpha1 + alpha2 + alpha.3
Proof.
                    ____________________________________
                    alpha_S1 = alpha1 + alpha2 + alpha3
```

La commande `Goal` permet de créer des propriétés que nous ne pourrons pas utiliser par la suite.

Pour utiliser notre vérificateur de calcul *(min, +)*, nous donnons une tactique `nccoq`. Celle ci va directement appliquer les propriétés nécessaires pour vérifier le calcul. Cette tactique s'applique directement de la manière suivante :

```
* Goals *
Goal alpha_S1 = alpha1 + alpha2 + alpha.3
Proof.
nccoq.
Qed.
```

Le but est alors résolu automatiquement est nous pouvons mettre un terme à la preuve avec `Qed`.

Nous continuons à vérifier les calculs en vérifiant que la déviation horizontale entre α_{S1} et β_1, calculée dans la Commande 4, est bien bornée par la valeur $d_{\beta_{S1}} = 6/25$.

Pour vérifier les déviations horizontales et verticales, nous allons utiliser de nouvelles fonctions : `hDev_bounded` et `vDev_bouded` respectivement, obtenues en étendant les lemmes 7.11 et 7.12. Ces dernières permettent de donner un critère suffisant pour vérifier qu'un élément est un majorant de la déviation. Dans notre cas, nous souhaitons borner la déviation horizontale par un élément d : le critère que nous avons développé utilise le produit de convolution par la fonction δ appliquée sur cet élément d. En effet, cette opération permet de déplacer une fonction vers la droite si d est positif (cf. Lemme 6.12).

Pour écrire ce critère, nous avons donné la fonction `hDev_bounded`. Elle prend trois arguments : les deux premiers sont les deux fonctions dans $\mathcal{F}$ dont nous cherchons à borner la déviation et le troisième est cette borne. Donc, pour vérifier que la déviation entre `alpha_S1` et `beta1` est bornée par 6/25, la valeur donnée par l'oracle, nous formalisons en Coq :

```
Goal hDev_bounded alpha_S1 beta1 (6/25).
Proof. nccoq. Qed.
```

qui est résolu par notre tactique.

Une fois la valeur de $d_{\beta_{S1}}$ vérifiée, nous pouvons introduire la fonction β_{S1} comme nous l'avons fait dans la Commande 5. Nous avions alors obtenue une fonction PA à deux segments que nous formalisons ainsi :

```
Definition beta_S1 := F_of_sequpp {|
  sequpp_T := 1;
  sequpp_d := 1;
  sequpp_c := 0;
  sequpp_js := [:: (0,  (0%:E, (0,  0%:E)));
                (6/25,  (0%:E, (0,  +∞ %E))) ] |}%bigQ.
```

Dans la Commande 7, nous avons cherché à calculer la déconvolution entre la fonction `alpha_S1` et `beta_S1` pour déterminer une contrainte sur la sortie de ce premier serveur. Nous avons obtenue la fonction `alpha1_S1`' qui, une fois formalisée, donne :

```
Definition alpha1_S1' := F_of_sequpp {|
  sequpp_T := 0;
  sequpp_d := 1;
  sequpp_c := 20_000;
  sequpp_js := [:: (0,  (12_800%:E, (20_000, 12_800%:E))) ] |}%bigQ.
```

dont nous vérifions qu'il s'agit bien d'un majorant pour la déconvolution entre `alpha_S1` et `beta_S1` avec :

```
Goal alpha1 / beta_S1 ⩽ alpha1_S1'.
Proof. nccoq. Qed.
```

Pour les raisons expliquées dans le développement du cas d'étude, nous calculons `alpha1_S1` dans la Commande 8 qui est le minimum de `alpha1_S1`' et δ_0. Nous introduisons `alpha1_S1` et δ_0 de la manière suivante :

```
Definition alpha1_S1 := F_of_sequpp {|
  sequpp_T := 1;
  sequpp_d := 1;
  sequpp_c := 20_000;
  sequpp_js := [:: (0,  (0%:E, (20_000, 12_800%:E))) ] |}%bigQ.

Definition d0 := F_of_sequpp {|
  sequpp_T := 1;
  sequpp_d := 1;
  sequpp_c := 0;
  sequpp_js := [:: (0,  (0%:E, (0,  +∞ %E))) ] |}%bigQ.
```

pour vérifier :

```
Goal F_min alpha1_S1' d0 = alpha1_S1.
Proof. nccoq. Qed.
```

Pour terminer la vérification des calculs du flux f_1, il faut vérifier tous les calculs qui y sont associés. Ainsi, tous les calculs du fichier `CaseStudy.nc` sont vérifiés de la même manière que ceux présentés ci dessus dans le fichier `case.v`.

Les derniers calculs concernent les Commandes 9 et 10. Dans la première commande, nous avons demandé le calcul de la convolution de chaque service résiduel reçu par f_1. Nous avons alors obtenu la fonction $\beta_{1_{E2E}}$ qui est formalisée avec :

```
Definition beta1_E2E := F_of_sequpp {|
  sequpp_T := 2;
  sequpp_d := 1;
  sequpp_c := 0;
  sequpp_js := [:: (0,  (0%:E, (0,  0%:E)));
                   (3612/3125, (0%:E, (0,  +∞ %E)))] |}%bigQ.
```

et vérifiée avec :

```
Goal beta1_E2E = (beta_S1 * beta_S2_1 * beta_S3 * beta_S5_1).
Proof. nccoq. Qed.
```

Cette dernière fonction nous permet de vérifier le calcul de la borne de temps de traversé du flux f_1 avec :

```
Goal hDev_bounded alpha1 beta1_E2E (3612/3125).
Proof. nccoq. Qed.
```

Ainsi, nous avons pu vérifier qu'avec les entrées du problème que nous avions, le temps maximal de traversé d'un message passant par le flux f_1 ne dépasse pas $3612/3125 = 1{,}15584$s. Une telle borne a aussi été déterminée pour les autres flux de notre cas d'étude et leurs vérification se trouve dans le fichier `case.v`.

7.8 Conclusion

Dans ce chapitre, nous avons montré comment les calculs pouvaient être effectués en *Calcul réseau*. Nous avons aussi montré quelle étaient les classes de fonctions utilisées dans de tels calculs, notamment dans les algorithmes présentés par Bouillard et Thierry dans [BT08].

Nous avons ensuite formalisé en Coq les ensembles de fonctions présentés dans [BT08]. En particulier nous avons formalisé l'ensemble de fonctions $\mathcal{F}_{UPP-PA}$. Nous avons donné des généralisations

de certaines propriétés de stabilité présentées dans ce même article. Cette formalisation a nécessité d'expliciter certaines propriétés ou définition de [BT08]. Par exemple, la valeur de $\tilde{T}$ du Lemme 7.6 pour déterminer la période d'une convolution de deux fonctions de $\mathcal{F}_{UPP}$. En effet, il est toujours nécessaire de détailler entièrement une propriété formalisée en Coq et donc certains détails qui paraissent évidents dans [BT08] doivent être expliqués et décris dans tout assistant de preuve.

De plus, nous avons obtenu des critères pour vérifier les opérations *(min, +)* effectuées par un oracle. Certains de ces critères sont des conditions suffisantes pour vérifier que la fonction retournée par l'oracle est exactement la fonction attendue. Ils ont pu être développés pour les opérations d'addition, de minimum et de convolution. D'autres critères permettent de vérifier que le résultat de l'oracle est un majorant du résultat d'une opération. Ces opérations, que sont les déviations horizontales et verticales et la déconvolution, ne nécessitent que d'un tel critère dans l'usage qui en est fait en général dans le *Calcul réseau*.

Nous avons aussi été capables de tester notre formalisation sur le cas d'étude du chapitre précédent. Ainsi, nous avons dans cette partie vérifié en Coq la correcte application de valeur concrètes sur notre cas d'étude. Cet ajout permet d'avoir un cas d'étude totalement formalisé en Coq, puisque nous pouvons vérifier la théorie de celui ci grâce au chapitre précédent et nous pouvons vérifier les calculs sur cette théorie grâce à ce chapitre.

Nous avons cependant fait évoluer le code que nous avons présenté dans la partie principale de ce chapitre, en Section 7.2, 7.3 et 7.4 pour pouvoir décrire le cas d'étude de la Section 7.7. Pour pouvoir calculer des valeurs concrètes avec nos critères, nous avons repris la formalisation des fonctions PA. Il a été bien plus pratique d'enregistrer par exemple un point de discontinuité (x et y) suivi d'une pente ρ et d'un offset σ. Ainsi, un segment débute à un point de discontinuité puis est décris par une pente et un offset. Il s'arrête au prochain point de discontinuité, c'est à dire aux points x et y du prochain élément de la liste. Cette modification a facilité les preuves pour aboutir à la vérification du cas d'étude. De plus, nous avons besoin d'utiliser des valeurs rationnelles de `MathComp`. Pour cette raison, nous avons préféré utiliser les nombres réels étendus de `MathComp Analysis`. Ainsi, les structures algébriques sont déjà liées et le passage de l'une à l'autre est assez simple. De plus, la lisibilité dans cette librairie est très explicite et permet de connaître dans quelles structures se situe chaque élément.

Chapitre 8

Conclusion

Cette thèse avait pour objectif la **formalisation en Coq le *Calcul réseau***. Nous allons dans ce dernier chapitre conclure ces travaux en redonnant le raisonnement que nous avons suivi ainsi que les résultats auxquels nous avons abouti. Nous discuterons ensuite d'extensions envisageables à ces travaux.

Pour parvenir à notre objectif, nous avons séparé le raisonnement en trois parties. Afin de pouvoir utiliser la structure algébrique sur laquelle est décrite le *Calcul réseau*, nous avons formalisé en Coq cette structure. Cette formalisation a permis de créer une librairie Coq utilisable qui décrit les structures algébriques de *dioïde* et de *dioïde complet*. Elle est disponible à l'adresse :

`https://github.com/math-comp/dioid`

Cette même librairie nous permet de formaliser les instances de ces structures algébriques utilisées dans la théorie que décris le *Calcul réseau* et d'ainsi créer une base mathématique au reste de nos travaux.

Nous avons donc utilisé ces instances dans la description du *Calcul réseau*. Au cours de cette description, nous avons formalisé les définitions existantes de cette méthode ainsi que les preuves associées aux théorèmes et propriétés mathématiques. Cette contribution se trouve sur :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien`

Cela nous a permis de formaliser complètement un cas d'étude que nous avons publié dans [RBR19]. Il est disponible dans les fichiers du lien précédent, dans :

`examples/case_study/`

Ce cas d'étude a été plus explicitement présenté dans ce manuscrit ainsi que le détail de son développement.

Afin de continuer l'apport de confiance dans l'utilisation du *Calcul réseau*, nous avons ensuite repris des travaux sur le calcul de valeurs effectives pour le *Calcul réseau*. Ces travaux élaborent des algorithmes pour pouvoir calculer des valeurs sur des fonctions *(min, +)*. Nous avons donc formalisé en Coq de telles fonctions. Pour pouvoir vérifier de tels calculs, nous avons développé autour de ceux ci des critères de vérification. Ces critères nécessitent de connaître préalablement un résultat de l'opération que l'on souhaite vérifier et donnent un nombre de points finis à vérifier. La présentation que nous avons donnée dans ce manuscrit a été publiée dans [RRB21]. Le développement de cette partie se trouve au lien :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien/minerve`

Par ailleurs, nous avons pu étendre ainsi le précédent cas d'étude : nous avons utilisé des algorithmes existants pour calculer des valeurs concrètes que nous avons vérifiées à l'aide de nos critères en Coq. L'usage de ce vérificateur sur des cas représentatifs de réseaux industriels est actuellement à l'étude. Les premiers résultats sont encourageants avec près de 2000 opérations vérifiées par Coq en moins de 5 minutes sur un ordinateur portable.

Pour compiler les trois librairies que nous avons présentées (*dioïde*, la librairie formalisant le *Calcul réseau* et le cas d'étude et le vérificateur de calculs `minerve`), il ne faut pas plus de 5 minutes pour chaque librairie. Tout les détails de compilation se trouve en Annexe, page 111.

Les contributions en code Coq se séparent en plusieurs parties. La partie développé sur la construction algébrique d'un *dioïde* et d'un *dioïde complet* est intégrée au projet `MathComp` et donc obtient les mêmes droits que celle-ci. La partie sur la formalisation du *Calcul réseau* et la vérification de calcul *(min, +)*, respectivement appelé `NCCoq` et `minerve`, est sous licence protégé. Chacune sont utilisable pour des fins académiques comme l'est le projet `Comcert` (`https://compcert.org/`). La page de licence se trouve en dans l'Annexe B.

La formalisation avec un assistant de preuve est un développement nécessitant beaucoup de détails. Ces détails apparaissent au niveau des preuves puisqu'il s'agit des formules que nous tenons à vérifier

particulièrement avec l'assistant de preuve. Le niveau de détail apparaît aussi lors de la formalisation des notions et définitions. En effet, nous avons vu dans ce manuscrit qu'il a souvent été nécessaire d'ajouter des éléments implicites dans les définitions originales de l'ouvrage de référence [BBLC18]. Par exemple, dans le Chapitre 5, nous avons dû donner une structure non détaillée dans les définitions originales de *dioïde complet*. La structure de *treillis complet* a ainsi été ajoutée pour formaliser en Coq la notion de « somme infinie ».

Dans cette étude, nous avons appliqué notre formalisation sur la preuve d'un cas d'étude. Pour y parvenir, nous avons décris les étapes de preuves Coq à la main. Une telle description est longue et l'appliquer à un cas avec plus de preuves semble être compliqué. En effet, chaque étape pour déterminer une contrainte à appliquer doit être décrite en Coq. De plus, une modification du réseau donne une modification importante des descriptions Coq. Il serait donc intéressant de pouvoir automatiser la rédaction de preuve en Coq pour faciliter cette partie.

Donc, pour automatiser la vérification avec un assistant de preuve des propriétés apportées par le *Calcul réseau* dans un réseau embarqué, nous avons vu que plusieurs approches étaient possibles. Une d'entre elles revient à donner une formalisation complète. Ainsi, nous avons donné une formalisation capable de vérifier que la théorie utilisée par l'outil est correcte.

En effet, les travaux montrés dans le Chapitre 6 explique comment nous avons développer suffisamment de propriétés pour les utiliser sur un cas d'étude. Ce cas d'étude permet d'affirmer que nous pourrions, dans des travaux futur, élaborer un moyen de générer une telle preuve à partir d'un outil de *Calcul réseau*. Cet outil, prenons par exemple PEGASE [Boy+10], pourrait écrire des preuves Coq pour déterminer une borne de temps de traversé. Cet ajout à l'outil demanderait d'inclure directement dans le code de PEGASE la possibilité de sortir une preuve Coq. Cette approche demande donc de modifier PEGASE. Une variation serait de ne pas inclure dans l'outil PEGASE mais plutôt reprendre ses traces. A partir de celles ci, il serait possible de déduire en Coq directement une preuve comme celle de notre cas d'étude.

Une autre approche possible est de développer directement en Coq un nouvel outil du *Calcul réseau* avec une preuve, aussi développer en Coq, que cet outil donne des résultats toujours corrects. La preuve apportée en plus autour de cet outil permettrait d'avoir encore plus confiance dans l'outil qui serait développé.

Chapitre A

Compilation du Code Coq

Pour compiler le code Coq de ce manuscrit, il est nécessaire d'avoir une version du gestionnaire de paquets `opam` disponible à :

`https://opam.ocaml.org/`

Pour exécuter tout le code Coq présenté dans ce manuscrit, il faut :

```
opam repo add coq-released https://coq.inria.fr/opam/released
opam repo add coq-extra-dev https://coq.inria.fr/opam/extra-dev
opam install coq.8.13.1 coq-mathcomp-algebra.1.12.0 coq-mathcomp-analysis.0.3.9
             coq-hierarchy-builder.1.0.0 coq-mathcomp-dioid.dev
             coq-coqeal.dev
```

A.1 Compilation de la librairie *dioïde* du Chapitre 5

Pour compiler la librairie présenté dans le Chapitre 5, il faut avoir `opam` et les paquets suivants :

- `coq-mathcomp-ssreflect`.1.12.0
- `coq-hierarchy-builder`.1.0.0
- `coq-mathcomp-analysis`.0.3.5

La librairie *dioïde* est accessible sur sur le lien :

`https://github.com/math-comp/dioid.git`

Pour l'installer et la compiler, il existe plusieurs possibilités. L'une d'entre elle permet de pouvoir aussi disposer des fichiers dans un dossier et ainsi pouvoir lire les descriptions des définitions (pour un lecteur coq avancé). Il faut alors :

```
git clone https://github.com/math-comp/dioid.git
cd dioid
git checkout 0.2
make
```

Cette compilation permet suivre les définitions des différentes structure algébrique présentées au Chapitre 5. Il est possible d'installer manuellement cette librairie et ainsi l'utiliser, notamment pour le reste du manuscrit. Pour cela, il faut lancer :

```
make install
```

Un autre moyen d'installer cette librairie et que nous utiliserons plus généralement dans le manuscrit est de lancer :

```
opam install coq-mathcomp-dioid.dev
```

A.2 Compilation de la librairie de formalisation du *Calcul réseau*

Pour observer pas à pas la formalisation du *Calcul réseau* donné au Chapitre 6, il est nécessaire d'avoir installer avec `opam`

- `coq`.8.13.1 `coq-mathcomp-algebra`.1.12.0

— `coq-mathcomp-analysis`.0.3.9
— `coq-hierarchy-builder`.1.0.0
— `coq-mathcomp-dioid.dev`

Le code de la formalisation se situe sur le lien suivant :

`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien`

Pour pouvoir suivre l'évolution, il est nécessaire de télécharger l'ensemble des fichiers de cette branche dans un dossier, de l'ouvrir et de lancer la commande :

```
git clone https://gitlab.rts.mpi-sws.org/proux/nc-coq/
cd nc-coq
git checkout phd-lucien
make
```

L'installation de cette librairie se fait avec :

```
make install
```

A.3 Compilation de MinErVe

Les fichiers nécessaire pour exécuter la vérification du cas d'étude présentée dans le Chapitre 7 (Vérification de calculs de fonctions *(min, +)*) sont disponibles sur le lien suivant :
`https://gitlab.rts.mpi-sws.org/proux/nc-coq/-/tree/phd-lucien`

Pour pouvoir exécuter le cas d'étude, il faut installer la librairie dédié au *Calcul réseau* comme c'est expliqué dans la partie précédente. Une fois cette partie terminée, il faut exécuter les commandes :

```
cd minerve
make
make install
```

qui va permettre d'installer la librairie `minerve` et ainsi rendre le cas d'étude exécutable.

A.4 Compilation du fichier de vérification du cas d'étude

```
coqc examples/case_study/case.v
```

Chapitre B

Licence NC-Coq

```
NON-COMMERCIAL LICENSE AGREEMENT FOR NC-COQ SOFTWARE

PREAMBLE

NC-Coq permits formal proofs of network calculus and proposes examples
using those results.

NC-Coq is and remains property of ONERA (www.onera.fr) that solely
holds the economic rights over it.

ONERA distributed NC-Coq under the terms of a Non-Commercial License
Agreement that grants the right to use the software for educational,
academic research or test and evaluation purposes only, and prohibits
any commercial use. Any use of the software, other than as authorised
under this licence agreement is prohibited (to the extent such use is
covered by a right of the copyright holder of the software).

For commercial use, please contact the authors of the Software.

DEFINITIONS

* "Licensee" means the Software user having accepted the Agreement.

* "Licensor" means ONERA

* "Agreement" means this license agreement and it shall not be
  modified or amended except by an instrument in writing signed by
  both parties hereto.

* "Software" means NC-COQ software in its Object Code and/or Source
  Code form and, where applicable, its documentation.

* "Derivative Work" means a work that is a modification of,
  enhancement to, derived from, or based upon the Software.

TERMS AND CONDITIONS

1. Scope of rights granted

On use and modification : Licensor hereby grants to the Licensee, who
accepts, a revocable, non-exclusive, non-transferable, royalty-free
and worldwide right to use the Software and prepare Derivative Works
solely for educational, academic and non-commercial research
endeavors, or test and evaluation purposes. Any other use is
considered as commercial use and is prohibited. In particular,
Licensee may not use the Software in connection with any activities
```

```
which purpose is to procure a commercial gain to him or others.

On distribution : The Licensee is authorized to distribute the
Software or any Derivative Works of the Software only under the same
terms and conditions as in this License, no other rights to the
Software or Derivative Works that are different from those provided by
this License can be granted. A copy of this Agreement must accompany
the distribution of the Software or any derivative works of the
Software.

2. Acceptance, Duration and Termination

The Licensee shall be deemed as having accepted the terms and
conditions of this Agreement as soon as he will have access to the
software and exercises any of the rights granted hereunder for the
first time.

The Agreement shall remain in force for the entire legal term of
protection of the economic rights over the Software.

The Agreement and the rights granted hereunder will terminate
automatically upon any breach by the Licensee of the terms of the
Agreement. Upon termination of this Agreement, Licensee shall
immediately discontinue all use of the Software.

Such a termination will not terminate the licence agreements of any
person who has received the Software or any Derivative Works of the
Software from the Licensee, provided such persons remain in full
compliance with the Agreement.

3. Publication Credit

Licensee agrees to mention Licensor and the name of the Software with
appropriate citations in any publication, presentation, document
containing results obtained in whole or in part through the use of the
Software.

4. Disclaimer of Warranty

The Software is a research and development Software, it is not an
industrial Software and may therefore contain errors, 'bugs' or a lack
of comments inherent to this type of development. For this reason, the
Software is provided "AS IS" without warranties of any kind concerning
the Software including, but not limited to, merchantability, fitness
for a particular purpose, absence of defects or errors, accuracy,
non-infringement of intellectual property rights.

5. Disclaimer of Liability

Licensor will in no event be liable for any direct or indirect,
material or moral, damages of any kind, arising out of the Agreement
or of the use of the Software, including, but not limited to, damages
for loss of goodwill, work stoppage, computer failure or malfunction,
```

```
loss of data or any commercial damage, even if the Licensor has been
advised of the possibility of such damage.

The Licensee uses the Software at its own risk and expense.

6. Additional Services

Support or issues updates to the software are not provided with the
software and not included in the Agreement.

7. Applicable law and settling of disputes

This Agreement shall be governed and construed in accordance with
French law.

The Parties agree to attempt to settle amicably any controversy or
claim arising under this Agreement or a breach of this
Agreement. Thereafter, both parties agree that all disputes between
them arising out of or relating to this Agreement, shall be submitted
to non-binding mediation unless the parties mutually agree
otherwise. All parties agree to exercise their best effort in good
faith to resolve all disputes in mediation. If not, disputes shall be
brought before the competent courts of France.

8. Third party libraries

The Software makes use of third-party libraries which had not been
modified. These third-party components are made available under a
number of different standard free software/open source licenses, and
are mentioned below :
* Coq standard library, LGPL-2.1, https://github.com/coq/coq
* MathComp, CECILL-B, https://github.com/math-comp/math-comp
* MathComp Analysis, CECILL-C, https://github.com/math-comp/analysis
* MathComp Dioid, CECILL-C, https://github.com/math-comp/dioid
* CoqEAL, MIT, https://github.com/CoqEAL/CoqEAL
```

Bibliographie

Contributions

[RBR19] Lucien Rakotomalala, Marc Boyer et Pierre Roux. « Formal Verification of Real-time Networks ». In : *JRWRTC 2019, Junior Workshop RTNS 2019*. TOULOUSE, France, nov. 2019.

[RRB21] Lucien Rakotomalala, Pierre Roux et Marc Boyer. « Verifying Min-Plus Computations with Coq ». In : *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*. Sous la dir. d'Aaron Dutle et al. T. 12673. Lecture Notes in Computer Science. Springer, 2021, p. 287-303.

Livres

[BBLC18] Anne Bouillard, Marc Boyer et Euriell Le Corronc. *Deterministic Network Calculus : From Theory to Practical Implementation*. John Wiley & Sons, Ltd, oct. 2018.

[LBT01] Jean-Yves Le Boudec et Patrick Thiran. *Network Calculus : A Theory of Deterministic Queuing Systems for the Internet*. T. 2050. Lecture Notes in Computer Science. Springer, 2001.

[MG08] Michel Minoux et Michel Gondran. *Graphs, Dioids and Semirings. New Models and Algorithms*. T. 41. Operations Research/Computer Science Interfaces Series. Springer, 2008.

[MT21] Assia Mahboubi et Enrico Tassi. *Mathematical Components*. Zenodo, jan. 2021.

Sources en ligne

[Rea10] RealTime-at-Work, éd. *RealTime-at-Work online Min-Plus Interpreter for Network Calculus*. `https://www.realtimeatwork.com/minplus-playground`. Consulté le 18 Juillet 2021. 2010.

[WT06] Ernesto Wandeler et Lothar Thiele. *Real-Time Calculus (RTC) Toolbox*. `http://www.mpa.ethz.ch/Rtctoolbox`. 2006.

Bibliographie

[AB14] Joan Adrià Ruiz De Azua et Marc Boyer. « Complete modelling of AVB in Network Calculus Framework ». In : *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*. Sous la dir. de Mathieu Jan et al. ACM, 2014, p. 55.

[ACR18] Reynald Affeldt, Cyril Cohen et Damien Rouhling. « Formalization Techniques for Asymptotic Reasoning in Classical Analysis ». In : *J. Formaliz. Reason.* 11.1 (2018), p. 43-76.

[Adn+10] Muhammad Adnan et al. « Model for worst case delay analysis of an AFDX network using timed automata ». In : *Proceedings of 15th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2010, September 13-16, 2010, Bilbao, Spain*. IEEE, 2010, p. 1-4.

[Adn+12] Muhammad ADNAN et al. « An improved timed automata approach for computing exact worst-case delays of AFDX sporadic flows ». In : *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012, Krakow, Poland, September 17-21, 2012*. IEEE, 2012, p. 1-8.

[Aff+20] Reynald AFFELDT et al. « Competing inheritance paths in dependent type theory : a case study in functional analysis ». In : *IJCAR 2020 - International Joint Conference on Automated Reasoning*. Paris, France, juin 2020, p. 1-19.

[Bau11] Henri BAUER. « Analyse pire cas de flux hétérogènes dans un réseau embarqué avion ». Thèse de doctorat dirigée par Fraboul, Christian Résaux, Télécommunications, Systèmes et Architecture (RTSA) Toulouse, INPT 2011. Thèse de doct. 2011.

[BD19] Marc BOYER et Hugo DAIGMORTE. « Impact on credit freeze before gate closing in CBS and GCL integration into TSN ». In : *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS 2019, Toulouse, France, November 06-08, 2019*. Sous la dir. de Jérôme ERMONT, Ye-Qiong SONG et Christopher D. GILL. ACM, 2019, p. 80-89.

[Ber06] Yves BERTOT. « Coq in a Hurry ». In : *CoRR* abs/cs/0603118 (2006). arXiv : `cs/0603118`.

[Ber+08] Yves BERTOT et al. « Canonical Big Operators ». In : *Theorem Proving in Higher Order Logics*. Sous la dir. d'Otmane Ait MOHAMED, César MUÑOZ et Sofiène TAHAR. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 86-101.

[BFG11] Anne BOUILLARD, Nadir FARHI et Bruno GAUJAL. *Packetization and Aggregate Scheduling*. Research Report RR-7685. INRIA, juil. 2011, p. 24.

[Bis+10] Luca BISTI et al. « DEBORAH : A Tool for Worst-Case Analysis of FIFO Tandems ». In : *Leveraging Applications of Formal Methods, Verification, and Validation*. Sous la dir. de Tiziana MARGARIA et Bernhard STEFFEN. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 152-168.

[BJ11] Anne BOUILLARD et Aurore JUNIER. « Worst-case delay bounds with fixed priorities using network calculus ». In : *5th International ICST Conference on Performance Evaluation Methodologies and Tools Communications, VALUETOOLS '11, Paris, France, May 16-20, 2011*. Sous la dir. de Samson LASAULCE et al. ICST/ACM, 2011, p. 381-390.

[BLM15] Sylvie BOLDO, Catherine LELAY et Guillaume MELQUIOND. « Coquelicot : A User-Friendly Library of Real Analysis for Coq ». In : *Math. Comput. Sci.* 9.1 (2015), p. 41-62.

[Boy10] Marc BOYER. « NC-Maude : A Rewriting Tool to Play with Network Calculus ». In : *Leveraging Applications of Formal Methods, Verification, and Validation*. Sous la dir. de Tiziana MARGARIA et Bernhard STEFFEN. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 137-151.

[Boy+10] Marc BOYER et al. « The PEGASE project : precise and scalable temporal analysis for aerospace communication systems with Network Calculus ». In : *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation - ISOLA 2010*. Heraclion, Greece, oct. 2010.

[BS13] Michael A. BECK et Jens SCHMITT. « The DISCO Stochastic Network Calculator Version 1.0 : When Waiting Comes to an End ». In : *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*. ValueTools '13. Torino, Italy : ICST (Institute for Computer Sciences, Social-Informatics et Telecommunications Engineering), 2013, 282–285.

[BT08] Anne BOUILLARD et Eric THIERRY. « An Algorithmic Toolbox for Network Calculus ». In : *Discret. Event Dyn. Syst.* 18.1 (2008), p. 3-49.

[CDM13] Cyril COHEN, Maxime DÉNÈS et Anders MÖRTBERG. « Refinements for Free ! » In : *Certified Programs and Proofs*. Sous la dir. de Georges GONTHIER et Michael NORRISH. T. 8307. Springer, 2013, p. 147-162.

[Cha+06] Hussein CHARARA et al. « Methods for bounding end-to-end delays on an AFDX network ». In : *18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings*. IEEE Computer Society, 2006, p. 193-202.

[Cru91] Rene L. CRUZ. « A calculus for network delay, Part I : Network elements in isolation ». In : *IEEE Trans. Inf. Theory* 37.1 (1991), p. 114-131.

[CSB16] Felipe CERQUEIRA, Felix STUTZ et Björn B. BRANDENBURG. « PROSA : A Case for Readable Mechanized Schedulability Analysis ». In : *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*. 2016, p. 273-284.

[CST20] Cyril COHEN, Kazuhiko SAKAGUCHI et Enrico TASSI. « Hierarchy Builder : Algebraic hierarchies Made Easy in Coq with Elpi (System Description) ». In : *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Sous la dir. de Zena M. ARIOLA. T. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 34 :1-34 :21.

[DAE12] J. DIEMER, Philip AXER et R. ERNST. « Compositional Performance Analysis in Python with pyCPA ». In : 2012.

[Dav+07] Robert I. DAVIS et al. « Controller Area Network (CAN) schedulability analysis : Refuted, revisited and revised ». In : *Real Time Syst.* 35.3 (2007), p. 239-272.

[DBZ18] Hugo DAIGMORTE, Marc BOYER et Luxi ZHAO. « Modelling in network calculus a TSN architecture mixing Time-Triggered, Credit Based Shaper and Best-Effort queues ». working paper or preprint. Juin 2018.

[DG00] Raymond R. DEVILLERS et Joël GOOSSENS. « Liu and Layland's schedulability test revisited ». In : *Inf. Process. Lett.* 73.5-6 (2000), p. 157-161.

[DR12] Daniel DE RAUGLAUDRE. « Vérification formelle de conditions d'ordonnancabilité de tâches temps réel périodiques strictes ». In : *JFLA - Journées Francophones des Langages Applicatifs - 2012*. Carnac, France, fév. 2012.

[Dut00] Bruno DUTERTRE. « Formal Analysis of the Priority Ceiling Protocol ». In : *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, 27-30 November 2000*. IEEE Computer Society, 2000, p. 151-160.

[Fra+18] Pascal FRADET et al. « A Generic Coq Proof of Typical Worst-Case Analysis ». In : *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*. IEEE Computer Society, 2018, p. 218-229.

[Fra+19] Pascal FRADET et al. « CertiCAN : A Tool for the Coq Certification of CAN Analysis Results ». In : *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*. Sous la dir. de Björn B. BRANDENBURG. IEEE, 2019, p. 182-191.

[Gon07] Georges GONTHIER. « The Four Colour Theorem : Engineering of a Formal Proof ». In : *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*. Sous la dir. de Deepak KAPUR. T. 5081. Lecture Notes in Computer Science. Springer, 2007, p. 333.

[Gre93] Klaus GRESSER. « An Event Model for Deadline Verification of Hard Real-Time Systems ». In : *Fifth Euromicro Workshop on Real-Time Systems, RTS 1993, Oulu, Finland, June 22-24, 1993. Proceedings*. IEEE, 1993, p. 118-123.

[Gri04] Jérôme GRIEU. « Analyse et évaluation de techniques de commutation ethernet pour l'interconnexion des systèmes avioniques ». Thèse de doctorat dirigée par Fraboul, Christian et Frances, Fabrice Informatique et télécommunications Toulouse, INPT 2004. Thèse de doct. 2004, 1 vol. (X-134 p.)

[GT06] Benjamin GRÉGOIRE et Laurent THÉRY. « A Purely Functional Library for Modular Arithmetic and Its Application to Certifying Large Prime Numbers ». In : *IJCAR*. 2006, p. 423-437.

[Gu+16] Ronghui Gu et al. « CertiKOS : An Extensible Architecture for Building Certified Concurrent OS Kernels ». In : *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Sous la dir. de Kimberly Keeton et Timothy Roscoe. USENIX Association, 2016, p. 653-669.

[Guo+19] Xiaojie Guo et al. « Integrating Formal Schedulability Analysis into a Verified OS Kernel ». In : *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Sous la dir. d'Isil Dillig et Serdar Tasiran. T. 11562. Lecture Notes in Computer Science. Springer, 2019, p. 496-514.

[Hen+05] Rafik Henia et al. « System level performance analysis - The SymTA/S approach ». In : *Computers and Digital Techniques, IEE Proceedings* - 152 (avr. 2005), p. 148 -166.

[JP86] Mathai Joseph et Paritosh K. Pandya. « Finding Response Times in a Real-Time System ». In : *Comput. J.* 29.5 (1986), p. 390-395.

[Kem+13a] Georges Kemayo et al. « Optimism due to serialization in the trajectory approach for switched Ethernet networks ». In : *Proc. of the 7th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2013)*. Sophia Antipolis, France, 2013.

[Kem+13b] Georges Kemayo et al. « Optimistic problems in the trajectory approach in FIFO context ». In : *Proc. of the 18th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2013)*. 2013.

[Lau+10] Michaël Lauer et al. « Analyzing End-to-End Functional Delays on an IMA Platform ». In : *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*. Sous la dir. de Tiziana Margaria et Bernhard Steffen. T. 6415. Lecture Notes in Computer Science. Springer, 2010, p. 243-257.

[Leh90] John P. Lehoczky. « Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines ». In : *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*. IEEE Computer Society, 1990, p. 201-209.

[Ler14] Xavier Leroy. *The CompCert C verified compiler : Documentation and user's manual*. Intern report. Inria, sept. 2014.

[LL73] C. L. Liu et James W. Layland. « Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment ». In : *J. ACM* 20.1 (1973), p. 46-61.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson et Wang Yi. « UPPAAL in a Nutshell ». In : *Int. J. Softw. Tools Technol. Transf.* 1.1-2 (1997), p. 134-152.

[MA10] Ahlem Mifdaoui et Hamdi Ayed. « WOPANets : a tool for WOrst case performance analysis of embedded networks ». In : *International Workshop on Computer-Aided Modeling Analysis and Design of Communication Links and Networks (CAMAD)*. Miami, United States, 2010.

[MA14] Alexandre Mouradian et Isabelle Augé-Blum. « Formal Verification of Real-Time Wireless Sensor Networks Protocols : Scaling Up ». In : *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*. IEEE Computer Society, 2014, p. 41-50.

[Mab+13a] Etienne Mabille et al. « Certifying Network Calculus in a Proof Assistant ». In : *EUCASS - 5th European Conference for Aeronautics and Space Sciences*. Astrium and Technische Universität München. Munich, Germany, juil. 2013.

[Mab+13b] Etienne Mabille et al. « Towards Certifying Network Calculus ». In : *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Sous la dir. de Sandrine Blazy, Christine Paulin-Mohring et David Pichardie. T. 7998. Lecture Notes in Computer Science. Springer, 2013, p. 484-489.

[MC97] Aloysius K. Mok et Deji Chen. « A Multiframe Model for Real-Time Tasks ». In : *IEEE Trans. Software Eng.* 23.10 (1997), p. 635-645.

[MM06] Steven Martin et Pascale Minet. « Schedulability analysis of flows scheduled with FIFO : application to the expedited forwarding class ». In : *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece.* IEEE, 2006.

[MMG04] Steven Martin, Pascale Minet et Laurent George. « The Trajectory Approach for the End-to-End Response Times with Non-preemptive FP/EDF ». In : *Software Engineering Research, Management and Applications, Second International Conference, SERA 2004, Los Angeles, CA, USA, May 5-7, 2004, Selected Revised Papers.* Sous la dir. de Walter Dosch, Roger Y. Lee et Chisu Wu. T. 3647. Lecture Notes in Computer Science. Springer, 2004, p. 229-247.

[MT13] Assia Mahboubi et Enrico Tassi. « Canonical Structures for the working Coq user ». In : *ITP 2013, 4th Conference on Interactive Theorem Proving.* Sous la dir. de Sandrine Blazy, Christine Paulin et David Pichardie. T. 7998. LNCS. Rennes, France : Springer, juil. 2013, p. 19-34.

[Nel+15] Geoffrey Nelissen et al. « Timing Analysis of Fixed Priority Self-Suspending Sporadic Tasks ». In : *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015.* IEEE Computer Society, 2015, p. 80-89.

[Owr+92] S. Owre et al. « PVS : A Prototype Verification System ». In : *11th International Conference on Automated Deduction (CADE).* Sous la dir. de Deepak Kapur. T. 607. Lecture Notes in Artificial Intelligence. Saratoga, NY : Springer-Verlag, 1992, p. 748-752.

[RC18] Antonin Riffard et Felipe Cerqueira. « Towards Synchronization in Prosa ». In : 2018.

[Ric05] Kai Richter. « Compositional scheduling analysis using standard event models : the SymTA/S approach ». Thèse de doct. University of Braunschweig - Institute of Technology, 2005.

[Rou19] Damien Rouhling. « Formalisation Tools for Classical Analysis - A Case Study in Control Theory ». Thèse de doct. University of Côte d'Azur, Nice, France, 2019.

[SD11] Wilfried Steiner et Bruno Dutertre. « Automated Formal Verification of the *TTEthernet* Synchronization Quality ». In : *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings.* Sous la dir. de Mihaela Gheorghiu Bobaru et al. T. 6617. Lecture Notes in Computer Science. Springer, 2011, p. 375-390.

[Sem20] Vincent Semeria. « Nombres réels dans Coq ». In : *JFLA*. 2020, p. 104-111.

[SRL90] Lui Sha, Ragunathan Rajkumar et John P. Lehoczky. « Priority Inheritance Protocols : An Approach to Real-Time Synchronization ». In : *IEEE Trans. Computers* 39.9 (1990), p. 1175-1185.

[SSH07] Henrik Schioler, Hans P. Schwefel et Martin B. Hansen. « CyNC : A MATLAB/-SimuLink Toolbox for Network Calculus ». In : *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools.* ValueTools '07. Nantes, France : ICST (Institute for Computer Sciences, Social-Informatics et Telecommunications Engineering), 2007.

[SZ06] Jens B. Schmitt et Frank A. Zdarsky. « The DISCO Network Calculator : A Toolbox for Worst Case Analysis ». In : *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools.* valuetools '06. Pisa, Italy : Association for Computing Machinery, 2006, 8–es.

[TCN00] Lothar Thiele, Samarjit Chakraborty et Martin Naedele. « Real-time calculus for scheduling hard real-time systems ». In : *IEEE International Symposium on Circuits and Systems, ISCAS 2000, Emerging Technologies for the 21st Century, Geneva, Switzerland, 28-31 May 2000, Proceedings.* IEEE, 2000, p. 101-104.

[THW94] Ken Tindell, H. Hanssmon et Andy J. Wellings. « Analysing Real-Time Communications : Controller Area Network (CAN) ». In : *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994*. IEEE Computer Society, 1994, p. 259-263.

[Wan06] Ernesto Wandeler. « Modular performance analysis and interface based design for embedded real time systems ». Thèse de doct. ETH Zurich, 2006.

[Wil98] Matthew Wilding. « A Machine-Checked Proof of the Optimality of a Real-Time Scheduling Policy ». In : *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*. Sous la dir. d'Alan J. Hu et Moshe Y. Vardi. T. 1427. Lecture Notes in Computer Science. Springer, 1998, p. 369-378.

[ZB09] Fengxiang Zhang et Alan Burns. « Schedulability Analysis for Real-Time Systems with EDF Scheduling ». In : *IEEE Trans. Computers* 58.9 (2009), p. 1250-1258.