Vérification Formelle dans le Modèle Polyédrique

Katell Morin-Allory

N° d'ordre: 2910

THÈSE

Présentée devant

devant l'université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Katell MORIN-ALLORY

Équipe d'accueil : LANDE
École doctorale : MATISSE
Composante universitaire : IFSIC/IRISA

Titre de la thèse :

Vérification Formelle dans le Modèle Polyédrique

soutenue le 27 octobre 2004 devant la commission d'examen

M. :	Philippe	CLAUSS	Président, Rapporteur
M. :	Nicolas	HALBWACHS	Rapporteur
MM. :	David	CACHERA	Examinateurs
	Sanjay	RAJOPADHYE	
	Jean-Pierre	TALPIN	
	Patrice	QUINTON	

Aucune règle n'existe, les exemples ne viennent qu'au secours des règles en peine d'exister.

André Breton, *Le surréalisme et la peinture*

Remerciements

Je tiens tout d'abord à remercier les personnes qui ont permis la soutenance de cette thèse :

- Philippe CLAUSS et Nicolas HALBWACHS pour avoir bien voulu être mes rapporteurs malgré leur charge de travail. Je les remercie pour toutes les remarques faites sur ce travail. Merci à Philippe Clauss pour ses remarques sur la bibliothèque polyèdrique. Elles montrent que le travail présenté ici mérite d'être poursuivi.

- Jean-Pierre TALPIN qui a accepté de participer à ce jury.

- Sanjay RAJOPADHYE pour m'avoir initialement accueillie au sein de son équipe et qui a accepté de venir du Colorado pour être membre de ce jury.

- Patrice QUINTON qui a accepté de diriger ma thèse. Je le remercie pour toutes les discussions que nous avons pu avoir.

- Enfin, David CACHERA qui a eu la lourde tâche de m'encadrer. Je le remercie pour ses conseils, critiques toujours constructives et la patience dont il a fait preuve.

Je remercie les membres des équipes R2D2 et Lande dans lesquelles j'ai été accueillie durant ma thèse.

Je remercie l'équipe pédagogique de l'université de Nice et l'équipe Sports de l'I3S pour leur accueil chaleureux. Un merci particulier à Laurence Pierre pour m'avoir permis de ne commencer mon enseignement qu'au second semestre et pour m'avoir acceuillie dans son équipe, à Jean-Pierre Lips pour m'avoir fait découvrir la cuisine niçoise de "la petite dame" et à Olivier Dalle grâce à qui je me débrouille maintenant en administration système...

Je remercie M. et M^{me} Beaucé pour m'avoir si gentiment hébergé pendant six mois à Nice.

Un grand merci aux gens du 2VB et plus particulièrement à Sylvain pour les soirées passées ensembles. Merci pour la soirée au festival de Cannes...

Merci aux magistériens, à leurs conjoints et aux petits nouveaux : Julien, Katell, Elouan, Solen, Damien, Karine, Joaquin, Françoise, Cédric, Thibault, Gwéno, Kristen pour leur amitié et pour avoir toujours été présents quand il le fallait.

Merci aux doctorants, non-doctorants et anciens doctorants de l'Irisa : Madeleine, Eric, Michel, Philippe, Karine, Elena pour tous les repas et pauses pris à la cafeteria.

Merci à la famille Cervera, ma famille d'adoption rennaise pour leur amitié, leurs encouragements, nos soirées du mercredi, et pour tous les livres qu'ils m'ont fait découvrir !

Merci à toute ma famille pour leur soutien affectif, pour avoir cru en moi et m'avoir toujours encouragée.

Et bien sûr, merci à Olivier...

Introduction

Contexte général

Systèmes enfouis : La notion de système embarqué est apparue lors de la conquête de l'espace, quand il a fallu placer un système informatique à bord des vaisseaux spatiaux. De cette époque vient le terme anglais « embedded system » qui a été traduit par « système embarqué ». Depuis, l'utilisation de tels systèmes s'est largement répandue, et l'on en trouve actuellement jusque dans les réfrigérateurs. Ainsi, le terme « système embarqué » n'étant plus adapté, on lui préfère le terme de « système enfoui ».

Un système enfoui est un système logiciel ou matériel qui forme un composant d'un système plus complexe. Ces systèmes doivent fonctionner sans intervention humaine. Ils sont spécialisés pour effectuer certaines opérations. Ces systèmes sont présents dans les télécommunications et les transports, mais aussi dans de nombreuses applications plus discrètes mais d'une importance critique comme les systèmes de contrôle aérien et les centrales nucléaires. Ces systèmes associent intimement les développements logiciel et matériel, informatique et électronique. Ils sont constitués de circuits intégrés. Ces derniers peuvent contenir plusieurs millions de transistors.

La conception des systèmes enfouis est un processus difficile : la complexité croissante des applications, l'évolution rapide des technologies de réalisation des circuits intégrés, nécessitent la conception de nouvelles architectures de systèmes numériques. Les méthodologies de conception associées doivent être capables de tenir compte de nombreuses contraintes : temps réel, fiabilité, complexité, consommation. La conception des systèmes enfouis nécessite un niveau de fiabilité important, du fait de l'aspect « critique » de ces systèmes tout comme de leur complexité croissante et de leur importance grandissante dans le monde actuel. Les méthodes de conception et de vérification y sont souvent moins développées que pour la conception de logiciel. Néanmoins, le besoin en fiabilité et en complexité réclame l'utilisation de méthodes et modèles de haut niveau pour garantir la sécurité des développements.

Pour concevoir ces systèmes, des méthodes de « codesign » sont utilisées : les parties matérielles et logicielles sont développées simultanément. Les techniques de conception consistent en des raffinements successifs d'une description haut niveau d'abstraction (spécification) jusqu'à une description bas niveau (implémentation) en incluant des techniques d'optimisation. Pour effectuer cette synthèse, le concepteur utilise des outils automatiques. Cependant, il est rarement prouvé que ces outils préservent la sémantique de la description initiale. De plus de nombreuses optimisations peuvent être introduites manuellement, et peuvent donc générer des erreurs. Il est donc important de vérifier que le système obtenu reste correct vis-à-vis de sa spécification initiale ; les techniques de vérification formelle sont indispensables lors de la conception de systèmes enfouis.

Vérification La vérification formelle des systèmes enfouis est fondamentale dans la conception de tels circuits. Elle permet d'assurer que les systèmes respectent bien leur spécification. Afin de limiter les coûts et le temps de conception, les systèmes sont généralement décrits de manière générique, à l'aide de paramètres. Ils sont ainsi facilement réutilisables. Les propriétés sur ces systèmes doivent donc être prouvées pour toutes les valeurs des paramètres. Apt et Kozen [4] ont montré que prouver que des architectures paramétrées étaient conformes à leur spécification n'était pas décidable. Cependant

de nombreuses heuristiques ont été développées (nous en présenterons quelques-unes dans le chapitre sur l'état de l'art). Pour vérifier les propriétés et les spécifications, il existe actuellement deux types de méthodes : la vérification utilisant des prouveurs de théorèmes et la vérification à l'aide de modèle. Selon le problème étudié, l'une ou l'autre des méthodes est utilisée.

- Prouveur de théorèmes [41, 54]. L'utilisation de prouveur de théorème permet d'effectuer une vérification hiérarchique au cours de la synthèse. À chaque étape de raffinement, on vérifie que le système raffiné est équivalent au système précédent, c'est-à-dire que les signaux de sorties des deux systèmes sont égaux si les signaux d'entrées le sont. Pour cela, l'utilisation d'un prouveur de théorèmes est particulièrement adapté. En effet, les prouveurs de théorèmes permettent de modéliser un circuit à différents niveaux d'abstraction. Le circuit initial et le circuit raffiné sont traduits dans le formalisme associé au prouveur de théorèmes et les théorèmes nécessaires pour prouver l'équivalence sont automatiquement générés. La vérification hiérarchique pose le problème suivant : les circuits doivent respecter un format particulier pour pouvoir être automatiquement traduits et pour que les théorèmes puissent être générés. Cette méthode n'est pas toujours compatible avec les outils de synthèse automatique qui ne respectent pas forcément les formats nécessaires à la preuve. De plus, l'utilisation de prouveurs de théorèmes nécessite très souvent l'intervention de l'utilisateur qui doit préciser les tactiques à utiliser.
- Vérification à l'aide de modèle [70]. L'idée ici est de définir un niveau d'abstraction et d'écrire un modèle, c'est-à-dire une description formelle du circuit à ce niveau d'abstraction. Il faut aussi définir les propriétés exprimant la correction du modèle pour ce niveau d'abstraction. Ces propriétés sont généralement écrites dans une logique temporelle. Puis, on vérifie que toutes les exécutions du modèle vérifient les propriétés. L'idée est de vérifier systématiquement toutes les situations possibles qui peuvent apparaître au cours d'une exécution. Le principal inconvénient de cette approche est l'explosion combinatoire du nombre d'états par rapport à la taille du modèle. Tant que les systèmes ne sont pas trop gros et qu'ils ont un nombre fini d'états, on peut utiliser les techniques de vérification de modèle « model checking ». Cependant, dans le cas qui nous intéresse, celui des architectures régulières paramétrées, ces méthodes ne sont plus utilisables directement. Pour prouver de tels systèmes, l'idée est d'utiliser des techniques d'abstraction qui permettent de représenter de manière finie un nombre d'états infinis, et de générer automatiquement des invariants. Ces méthodes sont assez difficiles à mettre en place et nécessitent généralement une intervention importante de l'utilisateur qui doit ajuster des paramètres.

Les méthodes formelles de vérification, indispensables à la conception des systèmes réguliers paramétrés, sont justement difficiles à mettre en place dans ce cadre et nécessitent une intervention importante de l'utilisateur.

Problématique

Dans ce mémoire, nous ne nous intéresserons qu'à un certain type de systèmes enfouis. Les systèmes que nous étudierons présenteront une certaine régularité, c'est-à-dire qu'ils peuvent être décomposés en un nombre restreint de cellules distinctes et simples. L'ensemble des cellules (qui peuvent être très nombreuses) est connecté de manière régulière en suivant des structures géométriques (anneau, hypercubes, ...). Ces systèmes permettent d'effectuer des calculs très complexes appliqués à un grand nombre de données à partir d'un petit nombre d'opérations de base. Pour de tels systèmes, nous parlons alors de circuits réguliers ou d'architectures régulières.

Pour exploiter ces modèles réguliers, des techniques ont été développées depuis les années quatre-vingts, fondées sur le formalisme des équations récurrentes. Ce formalisme permet la description d'un algorithme à différents niveaux d'abstraction : on peut tout autant d'écrire une architecture à un haut niveau d'abstraction qu'à un niveau proche de l'implémentation. Nous nous intéresserons ici plus particulièrement aux équations récurrentes affines sur des domaines polyédriques. Ce formalisme est appelé

modèle polyédrique [97].

Le modèle polyédrique se fonde donc sur la combinaison d'Équations Récurrentes Affines (ERA) et de polyèdres entiers (c'est-à-dire l'intersection de $\mathbb{Z}^n$ avec un sous-espace de $\mathbb{Q}^n$ délimité par un nombre fini d'hyperplans). Une variable polyédrique est définie sur un domaine polyédrique (une union finie de polyèdres) par une ou plusieurs équations récurrentes affines. Une variable polyédrique est une application d'un domaine polyédrique dans un espace de valeurs ($\mathbb{R}, \mathbb{Z}$, booléens), elle peut être vue comme une généralisation de la notion de tableau multidimensionnel : à chaque point du domaine de la variable polyédrique correspond une variable scalaire que l'on appelle ici instance. Une instance de la variable est définie de manière unique par rapport aux autres instances par une équation (la nature équationnelle du modèle fait que les instances d'une variable ne peuvent avoir qu'une valeur définie possible). Une ERA permet de définir les valeurs de toutes les instances. Pour connaître la définition d'une instance en un point x, il suffit de restreindre le domaine de l'ERA au point x. Nous appelons système un ensemble d'équations récurrentes affines. Ce modèle permet de représenter des systèmes paramétrés, en utilisant des paramètres symboliques.

Le modèle polyédrique permet de représenter facilement des calculs réguliers. Il permet de spécifier, de raisonner, d'analyser, de transformer, et de paralléliser des équations récurrentes affines et ainsi de générer une architecture parallèle régulière. Ce modèle a été à l'origine développé pour la synthèse de réseaux systoliques [59]. Il est aussi utilisé pour l'analyse et la transformation de nids de boucles à bornes et à références tableaux affines [36, 26, 37].

Pour exprimer des systèmes dans le modèle polyédrique, l'équipe R2D2 de l'IRISA a développé le langage ALPHA. Un programme ALPHA est un système d'équations récurrentes affines définies sur un domaine polyédrique. Le langage ALPHA est un langage à assignation unique : on ne peut assigner qu'une seule valeur à chaque instance de variable.

Ce modèle permet d'obtenir rapidement une architecture régulière. Tout d'abord le système est décrit dans le modèle polyédrique à un haut niveau d'abstraction. Pour obtenir la description bas niveau, on procède par raffinements successifs d'une spécification algorithmique vers son implémentation. Ces techniques permettent donc d'obtenir rapidement une architecture régulière. Dans le cadre du langage ALPHA, nous utilisons l'environnement MMALPHA qui permet de manipuler et de transformer des descriptions haut niveau en un système régulier.

Jusqu'ici, la correction des systèmes obtenus reposait sur deux bases : (i) l'utilisation, au sein d'un formalisme unique tout au long de la dérivation, de transformations préservant la sémantique des systèmes ; (ii) l'interfaçage avec un démonstrateur de théorèmes lorsqu'il était nécessaire de prouver des propriétés « complexes ». Toutefois, les méthodes implémentées jusqu'ici ne permettaient que de prouver des propriétés fonctionnelles (des prédicats reliant les entrées aux sorties), et ne tiraient pas profit du modèle polyédrique.

> Le but de ce mémoire est d'automatiser la vérification de propriétés de sûreté sur des systèmes décrits dans le modèle polyédrique avec des paramètres formels. De manière informelle, une propriété de sûreté assure que quelque chose ne se produira jamais. Dans ce travail, nous ne nous intéresserons qu'à des propriétés de sûreté sur des signaux de contrôle. Les signaux de contrôles sont représentés par des variables booléennes (les termes variable et signal seront utilisés de manière interchangeable). Ces signaux sont des signaux qui n'interviennent pas directement dans le calcul du résultat fonctionnel du système. Ils sont soit générés automatiquement soit introduits manuellement dans la phase finale de la synthèse et servent par exemple à contrôler les accès aux registres.

La démarche Le problème que nous venons d'énoncer étant vaste, nous avons commencé par restreindre notre champ d'investigation. Nous avons ainsi restreint la forme des propriétés étudiées. Dans

un premier temps, nous devrons prouver qu'un signal vaut, sur tout un domaine, une valeur constante. Notre problème est donc le suivant :

Prouver que pour un signal X et sur un domaine $\mathcal{D}$:
- *soit chacune des instances de X définies sur $\mathcal{D}$ vaut la constante vrai,*
- *soit chacune des instances de X définies sur $\mathcal{D}$ vaut la constante faux.*

Pour construire ces preuves, nous avons développé différentes techniques de preuve :

- **Substitution par constante.** Comme les signaux sont définis récursivement, le raisonnement le plus adapté est la récurrence. Nous avons automatisé ce raisonnement sous forme d'une technique de substitution par constante :

  Cette première technique s'applique à des propriétés s'exprimant à l'aide d'une variable définie par récurrence. L'idée est de prouver la propriété récursivement en utilisant une induction induite par l'ordonnancement[1]*. Ainsi, nous supposons connaître les valeurs des instances de la variable jusqu'à un instant donné t, et dans les équations des instances définies à $t+1$, nous substituons les instances définies avant $t+1$ par leurs valeurs. Grâce à la forme générale des équations récurrentes affines, qui permettent de définir simultanément les valeurs d'un ensemble (potentiellement infini) d'instances d'une variable, nous mettons en œuvre une seule substitution syntaxique pour toutes les instances. La récurrence est ainsi « cachée » derrière cette substitution : elle n'apparaît que dans la preuve de la correction de la règle de substitution par constante. Si l'expression obtenue après substitution est trivialement vraie sur tout le domaine considéré, la preuve est terminée. La présence d'instances dont la valeur est fausse permet en revanche de fournir un contre-exemple.*

  Cette technique nous a permis de prouver une propriété de contrôle sur un filtre adaptatif. Cependant, cette technique n'est pas suffisante : après avoir utilisé cette technique, nous n'obtenons pas toujours une constante, nous pouvons obtenir une expression e. Il faut prouver que cette expression polyédrique vaut vrai (ou faux) sur tout un domaine. Nous avons donc recherché de nouvelles techniques de preuve.
- **Recherche d'auto-dépendance.** Si les variables ne sont pas définies récursivement (c'est-à-dire, en considérant une variable X, la définition de X ne fait pas intervenir d'occurrence de X) nous ne pouvons pas faire de substitution par constante. L'idée a donc été de transformer ces variables en variables récursives (définies à l'aide d'occurrences d'elles-mêmes, ou autrement dit à l'aide d'auto-dépendances) afin de pouvoir ensuite utiliser la technique de substitution par constante. Nous avons appelé cette technique recherche d'auto-dépendance.

  La recherche d'auto-dépendance s'applique à des variables qui ne sont pas définies récursivement. L'idée est de transformer ces variables en variables récursives. Nous utilisons des principes de réécritures et des techniques proches de l'unification.

  Cette technique de recherche d'auto-dépendance ne peut s'appliquer que dans certains cas que nous avons caractérisés. Cependant, en généralisant cette technique, nous obtenons des résultats intéressants en pratique. Cette généralisation est appelé recherche de motifs.
- **Recherche de motifs.** Lors de la construction des preuves nous avons constaté que pour certaines expressions nous n'arrivions pas à construire de preuve et que ni la recherche d'auto-dépendances ni la substitution par constante ne donnaient de résultats. Ces expressions sont composées de sous-expressions qui servaient à définir des variables. L'idée a donc été de remplacer tout d'abord ces sous-expressions par la variable définie par ces sous-expressions, puis, si nous connaissions la valeur de ces variables d'utiliser le principe de substitution par constante. Nous avons appelé cette technique recherche de motifs.

[1] Nous travaillons sur des systèmes ordonnancés, c'est-à-dire qu'un des indices (généralement le premier) de chaque variable représente l'instant où les instances de cette variable seront calculées.

La recherche de motifs permet de simplifier les expressions des variables. L'idée est de substituer une expression par une constante. Comme dans la recherche d'auto-dépendances, nous utilisons des principes de réécritures et des techniques proches de l'unification. La recherche d'auto-dépendance est une application de la recherche de motifs.

Les trois techniques que nous venons de présenter ne sont cependant pas suffisantes pour construire une preuve. Dans certains cas, nous avons été amenés à prouver des propriétés sur des domaines plus grands que ceux fournis initialement.

– **Agrandissements et Pseudo-pipelines.** Nous avons donc trouvé des heuristiques pour agrandir automatiquement les domaines. Nous avons constaté que pour certaines variables, l'agrandissement du domaine où devait être construit la preuve était très simple à obtenir. Nous nous sommes donc intéressés à ces variables que nous avons appelées pseudo-pipelines car elles correspondent à la généralisation de la notion de pipeline. Ces variables ont la particularité de bien propager les valeurs, c'est-à-dire qu'en connaissant la valeur d'une instance en un point, nous pouvons facilement en déduire la valeur d'autres instances sur tout un domaine. En effectuant une recherche systématique de ces variables et en propageant leurs valeurs, nous avons réussi à proposer des heuristiques d'agrandissement de domaine donnant en pratique de bons résultats.
– **Paramètres.** Jusqu'à présent, nous nous sommes restreints à un type très précis de propriété : prouver qu'un signal vaut une valeur constante sur tout un domaine. Pour lever cette contrainte, nous avons enrichi nos techniques de spécification des propriétés. Pour cela nous ajoutons des paramètres. L'ajout de paramètre modifie légèrement la manière d'interpréter la notion de construction de preuve : nous obtenons des preuves paramétrées, pour certaines valeurs des paramètres nous aurons construit une preuve et pour d'autres nous aurons construit des contre-exemples. Cette notion nous a ouvert d'autres voies. Nous avons ainsi développé des tactiques spécifiques aux preuves paramétrées comme des démonstrations par l'absurde, des preuves par récurrence sur les paramètres[2]. En appliquant ces différentes tactiques, nous sommes en mesure de prouver (pour le moment semi-automatiquement) des propriétés d'exclusion mutuelle et de priorité sur un arbitre matériel.

Contributions

Notre travail a permis de développer un cadre de preuve formelle lié au modèle polyédrique.

– Nous avons tout d'abord défini un premier cadre formel de spécification, puis nous avons défini formellement un ensemble de règles de preuve. Trois de ces règles, la substitution par constante, la recherche d'auto-dépendance et la recherche de motif sont indissociables du modèle polyédrique. La règle de substitution par constante permet de substituer simultanément un ensemble d'instances par une constante ; la règle de recherche d'auto-dépendance permet d'introduire automatiquement des auto-dépendances et la recherche de motif permet de simplifier l'expression du système. Nous avons montré que ces règles étaient correctes vis-à-vis de la sémantique.
– Nous avons ensuite défini des techniques d'automatisation de preuve : nous avons caractérisé un certain nombre de cas où la terminaison de l'itération des règles de preuve était décidable et d'autres où la terminaison de l'itération était indécidable. Ceci nous a permis de fournir un algorithme capable de construire soit une preuve ou un contre-exemple de la propriété, soit de nouveaux sous-buts à prouver.
– Pour accélérer la construction de la preuve, nous avons développé des heuristiques d'agrandissement de polyèdres. Ces heuristiques donnent des résultats intéressants dans la mesure où les

[2]La substitution par constante correspond quant à elle à une récurrence sur l'indice représentant le temps. Cependant, cette récurrence est cachée : elle n'est utilisée que pour prouver de la correction de la règle. En pratique, quand nous utilisons la règle de substitution par constante, nous n'effectuons donc aucune récurrence.

variables sur lesquelles portent la preuve respectent une certaine structure. Nous appelons ces variables pseudo-pipelines. Ces variables sont largement utilisées dans les systèmes sur lesquels nous travaillons.
- Enfin, nous avons enrichi le cadre de spécification de propriétés dans le modèle polyédrique. Nous ne présenterons ici que la syntaxe de nos propriétés. Nous avons ensuite développé des tactiques de preuve liées à ces formules. Ces tactiques s'appuient sur l'ajout de paramètres et sur des calculs de contraintes. Nous nous contenterons ici de présenter l'intuition de ces tactiques, la formalisation n'est pas complètement achevée.

Ces travaux ont été en partie implémentés dans l'environnement MMALPHA :
- toutes les règles de preuve,
- l'algorithme d'automatisation,
- l'ajout des paramètres,
- le calcul des contraintes.

Cette implémentation a permis de prouver des spécifications sur trois systèmes matériels :
- un filtre adaptatif (preuve automatique),
- un produit matrice-vecteur (preuve actuellement semi-automatique, il manque l'implémentation des heuristiques d'agrandissement),
- un arbitre matériel distribué (preuve semi-automatique).

Publications Ces travaux ont mené à un rapport de recherche, une publication dans une conférence et une publication dans une revue.
- La publication [18] dans la conférence internationale MEMOCODE'03 reprend principalement le chapitre 4 et aborde le chapitre 5. Les détails techniques de cette publication sont présentés dans le rapport de recherche [19].
- La seconde publication [20] paraîtra en décembre 2004 dans la revue Transaction on Embedded Computer Systems, ACM. Elle reprend les chapitres 4 et 5, et aborde le chapitre 6.

Ces deux publications ne prennent pas en compte la formalisation sous forme de système de preuve présenté dans ce document.

D'autres articles sont en cours de rédaction sur l'opérateur d'agrandissement, et sur la représentation des quantifications à l'aide de paramètres.

Plan

La première partie de ce travail sera consacrée à un état de l'art et à la présentation de la sémantique de notre modèle. Dans la seconde partie, nous présenterons les travaux effectués au cours de cette thèse. Le chapitre 3 illustrera sur un exemple notre processus de preuve. Le chapitre 4 nous permettra de présenter notre système de preuve, et nous fournirons pour cela une méthode de spécification des propriétés. Le chapitre 5 nous permettra d'aborder les problèmes liés à l'itération des règles de preuve, et nous montrerons quels sont les cas pour lesquels l'itération est finie. Dans le chapitre 6, nous nous intéresserons à l'automatisation de notre processus de preuve, et nous fournirons un algorithme de construction de preuve. Dans le chapitre 7, nous nous intéresserons aux techniques d'accélération et d'agrandissement de domaines. Le chapitre 8 sera consacré à l'étude d'application et dans le chapitre 9, nous nous intéresserons à l'exemple de l'arbitre et nous présenterons les différents travaux en cours de formalisation.

Première partie

Contexte

Chapitre 1

État de l'art

La conception de systèmes matériels complexes est de plus en plus difficile. Les pressions économiques et l'évolution rapide du marché demandent des temps de conception de plus en plus courts. Il est donc important qu'une fois un système ou un circuit obtenu, il puisse être réutilisé. Pour cela, les systèmes sont décrits de manière générique à l'aide de paramètres symboliques dont la valeur n'est pas connue a priori. Ces circuits sont conçus à partir d'une synthèse de haut niveau : les systèmes sont tout d'abord spécifiés à un très haut niveau en termes de diagrammes, de tableaux ou de textes informels. Puis, par des techniques de synthèse, les systèmes sont affinés jusqu'à obtenir une description physique. Ces méthodes de synthèse sont actuellement plus ou moins automatisées, mais nécessitent toujours une intervention humaine pour obtenir l'architecture la plus optimisée possible. Ce cycle de production est long et le temps de test laissé à la fin de la conception en est d'autant raccourci. Une erreur à un niveau quelconque de la conception peut s'avérer très coûteuse, et pour certains systèmes embarqués les erreurs sont inenvisageables. Les méthodes formelles sont une solution possible pour repérer les erreurs le plus tôt possible.

On appelle méthode formelle toute utilisation d'un raisonnement mathématique dans le développement d'un système logiciel et/ou matériel afin d'améliorer la qualité de ce système en le spécifiant et en le vérifiant. Le rôle de ces méthodes a été longuement discuté dans [9, 61, 25] et semble s'imposer lors de la conception de systèmes. Les méthodes formelles peuvent être utilisées à tous les niveaux de la conception et permettent ainsi de repérer très tôt des erreurs dans la conception. Nous nous intéresserons plus particulièrement aux méthodes formelles de vérification dans le cas des systèmes matériels [41, 54] et plus précisément, aux méthodes formelles de vérification des systèmes paramétrés.

Deux approches sont possibles pour la spécification et la vérification de matériel.

- La première approche consiste à spécifier les propriétés souhaitées du système (sûreté, vivacité, ...). Le système est valide s'il vérifie les propriétés. La spécification de telles propriétés passe généralement par l'utilisation d'une logique temporelle. La vérification se fait par vérification de modèle (*cf.* 1.1).
- La seconde approche consiste à spécifier le comportement du système à l'aide d'un modèle haut-niveau. La vérification consiste alors à vérifier que tous les comportements de l'implémentation sont cohérents avec les comportements du modèle. Dans ce cas, on utilise une approche déductive (*cf.* 1.2).

En résumé, la conception de système matériel commence donc généralement par une spécification haut niveau, puis à chaque étape de la conception, la description s'affine et se précise jusqu'à obtenir un niveau de description matérielle. Pour vérifier la correction du système obtenu, on utilise les deux approches de vérification : on commence par vérifier des propriétés sur le modèle haut niveau, puis à chaque étape de raffinement, on vérifie que le nouveau modèle est correct par rapport à la spécification précédente, le nouveau modèle servant de spécification pour le modèle suivant obtenu après une nouvelle étape de raffinement (Fig. 1.1). Ainsi, les propriétés du système sont préservées à chaque étape de

raffinement.

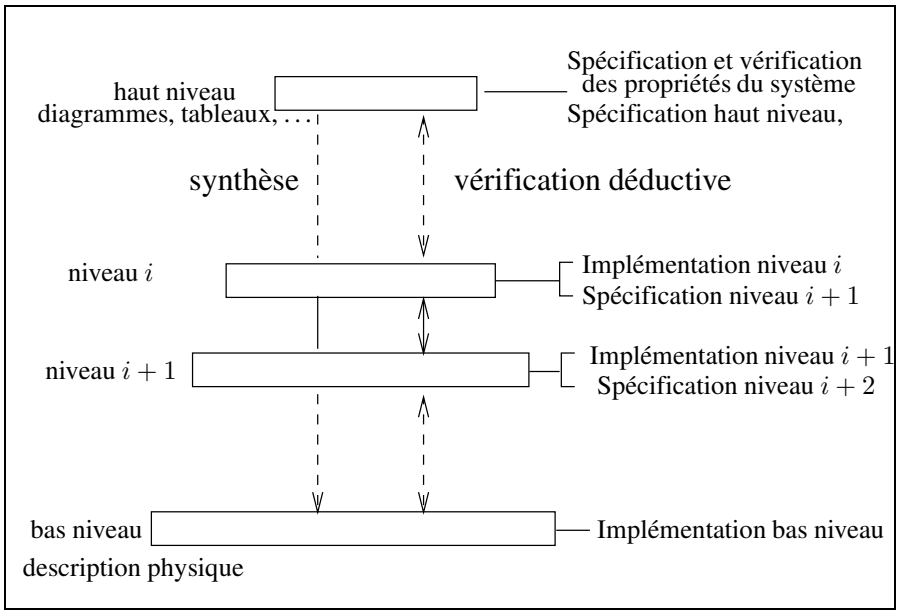


FIG. 1.1 – Vérification hiérarchique

Le sujet de cette thèse étant l'utilisation du modèle polyédrique dans les méthodes formelles de vérification, nous nous intéresserons donc à ces dernières dans cet état de l'art.

La première section de ce chapitre sera consacrée aux méthodes formelles de vérification. Nous présenterons les techniques les plus courantes. Dans la section 2, nous nous attacherons à une classe plus particulière d'architectures, les réseaux systoliques, et nous verrons quelques techniques de spécification et de vérification sont utilisées. La section 3 permettra de nous rapprocher du sujet de cette thèse, en présentant les différents travaux de vérification liés au modèle polyédrique. Enfin, dans la section 4, nous présenterons les objectifs de ce travail.

1 Méthodes formelles de vérification

Dans ce chapitre, nous présenterons les méthodes classiques de vérification. Nous verrons tout d'abord, en section 1.1, la technique de vérification de modèle, puis en 1.2 les méthodes de vérification déductive et nous comparerons les deux techniques en 1.3. Nous présenterons alors, en section 1.4, des techniques spécifiques aux systèmes paramétrés. Enfin, dans la section 1.5 nous parlerons de vérification de description VHDL.

1.1 La vérification de modèle

Cette méthode est utilisée pour montrer qu'un système vérifie certaines propriétés temporelles. Le système est modélisé par une structure M, généralement sous la forme d'un automate ; les propriétés, quant à elles, sont exprimées dans une logique temporelle. Notons ϕ une telle propriété. L'algorithme de vérification de modèle (*"model checker"*) est une procédure qui décide si la structure M est un modèle pour la formule ϕ, ou autrement dit, si M satisfait ϕ ($M \models \phi$) [30].

Modélisation du système La modélisation du système peut se faire de différentes manières. Nous parlerons ici des modélisations avec des structures de Kripke, des systèmes de transitions étiquetées, ou une combinaison des deux.

– Les structures de Kripke [50] sont des automates dont les états sont étiquetés par des propositions atomiques. Ces propositions atomiques représentent des propriétés de chaque état de l'automate. Une transition entre deux états signifie que l'un des états est atteignable depuis l'autre.
– Les systèmes de transitions étiquetées sont des automates dont les transitions sont étiquetées par des actions. Pour passer d'un état dans un autre, il faut effectuer une action. Ces systèmes de transitions permettent de représenter des systèmes concurrents.
– Les systèmes de transitions de Kripke représentent une combinaison des deux précédentes structures. Ils permettent de représenter aussi bien les propriétés statiques d'états en étiquetant les nœuds par des propositions atomiques, que le dynamisme du système en étiquetant les transitions par des actions.

Spécification Dans la vérification de modèle, les propriétés sont spécifiées dans une logique temporelle [30]. Cette logique est une spécialisation de la logique modale qui donne le cadre général d'une logique permettant d'exprimer le changement. La logique temporelle est constituée d'une logique classique du premier ordre à laquelle ont été ajoutés des opérateurs quantifiant le temps. Ces opérateurs se forment à partir des deux opérateurs de base suivants : l'opérateur toujours, noté $\Box$ et l'opérateur inévitablement, noté $\Diamond$. Les formules en logique temporelle sont interprétées par rapport au modèle choisi (structure de Kripke, système de transitions étiquetées, etc.). Les transitions dans de telles structures représentent une évolution du temps. De manière schématique, une formule en logique temporelle représente un ensemble d'états de la structure.

Nous utilisons ici la classification usuelle des propriétés temporelles [57].
– Propriétés de sûreté : ces propriétés affirment que rien de mauvais ne peut se produire. Elles se représentent avec l'opérateur $\Box$. Exemples : exclusion mutuelle, absence de blocage, invariants globaux.
– Propriétés de vivacité : elles affirment que quelque chose de bon peut inévitablement se produire. Elles se représentent à l'aide de l'opérateur $\Diamond$. Exemples : accessibilité, absence de famine (un processus en attente d'un service peut l'obtenir un jour).

Ce sont les deux grandes classes de propriétés temporelles. À cela, on peut ajouter des propriétés de précédence, des propriétés d'équité, des propriétés d'atteignabilité. On peut aussi trouver de nouvelles méthodes de classification des propriétés [72] : ces propriétés sont classées en fonction du comportement du système (fini ou infini) et du comportement que la propriété spécifie (fini ou infini). Il existe de nombreux types de logiques temporelles que l'on peut diviser en deux catégories : la logique temporelle linéaire (comme LTL [68], le μ-calcul [58, 16]) et la logique temporelle avec branchement (comme CTL [23], CTL* [31], le μ-calcul etc.).

Vérification de modèle L'algorithme de vérification de modèle est intrinsèquement lié à la logique choisie pour la spécification. Il y a différentes manières d'implémenter une procédure de vérification : l'approche sémantique, l'approche théorie des automates et l'approche tableau.
– L'idée derrière l'approche sémantique est de calculer itérativement la sémantique d'une formule à partir du modèle du système. Cette technique fonctionne bien pour les logiques temporelles avec branchement.
– Pour les logiques temporelles linéaires, l'approche par automate est préférée. Le problème se réduit à tester l'inclusion de deux automates.
– La méthode par tableau permet de savoir si un état du système vérifie une propriété ; elle construit un arbre de preuve en générant des sous-buts. Si l'arbre ne peut être construit, nous obtenons alors la preuve que l'état ne vérifie pas la propriété.

La limitation de la vérification de modèle vient de l'explosion du nombre d'états [16] : la structure modélisant le système augmente exponentiellement avec le nombre de composants en parallèle. Pour pallier ce problème, la solution est d'utiliser des techniques symboliques comme les diagrammes de

décision binaire (BDD) [15]. Avec de telles techniques, l'ensemble des états est représenté de manière symbolique et non plus explicite. Nous verrons dans la section 1.4 comment utiliser les techniques de vérification de modèle dans le cas des systèmes paramétrés. Le lecteur intéressé par la vérification de modèle trouvera une étude approfondie dans [70].

1.2 Les prouveurs de théorèmes

Vérifier qu'une implémentation est conforme à sa spécification revient à montrer un théorème dans une certaine logique. La spécification est faite dans une logique de premier ordre ou d'ordre supérieur. Le comportement d'un dispositif peut se modéliser à l'aide d'un prédicat par rapport aux entrées de ce dispositif. La spécification et l'implémentation sont toutes deux décrites dans la même logique. L'utilisation d'une logique comme langage de description de la spécification et de l'implémentation permet d'associer immédiatement à cette description une interprétation fondée sur la sémantique de la logique. Cependant, une description dans une logique demande de la part de l'utilisateur une très grande discipline, et une compréhension très claire de son système. Un prouveur de théorème permet de construire (semi-)automatiquement des preuves.

Un système de preuve dans une logique $\mathcal{L}$ est un ensemble d'axiomes et de règles d'inférence. Les axiomes sont des formules dans la logique $\mathcal{L}$ et sont élémentaires dans le sens où ils représentent les propriétés de base des opérateurs de la logique. Une règle d'inférence a la forme suivante :

$$\frac{\alpha_1, \ldots, \alpha_k}{\beta}$$

Les formules $\alpha_1, \ldots, \alpha_k$ sont appelées prémisses de la règle et β est appelée conclusion. Les axiomes et les règles sont en fait des "modèles d'axiomes et de règles", dans le sens où l'on peut substituer les variables les composant par des formules et générer ainsi une infinité d'axiomes. Une déduction dans un système de preuve est une suite de formules obtenues en appliquant des règles d'inférences. La dernière formule est généralement appelée théorème.

L'automatisation peut être faite de différentes manières.

- Le prouveur vérifie automatiquement qu'une série de formules est bien une déduction.
- Il aide à la construction d'une preuve.
- Il permet d'utiliser des procédures de décision. Les procédures de décision permettent de décider de manière algorithmique la validité d'une formule et ainsi d'alléger les preuves.

Pour automatiser au mieux de tels systèmes, l'utilisateur peut développer des stratégies de preuve qu'il pourra réutiliser par la suite. Dans le cadre de la vérification de matériel, de telles méthodes sont particulièrement bien adaptées car on utilise généralement le même schéma de preuve.

Une logique est consistante si toute formule que l'on peut prouver dans la logique est sémantiquement vraie ; une logique est complète si toute formule sémantiquement vraie peut être prouvée dans la logique. Un système logique qui n'est pas consistant est dangereux car il peut permettre de prouver des formules sémantiquement fausses. En revanche, un système non complet reste utile car il permet de construire des preuves à partir d'un ensemble de formules sémantiquement vraies.

Il existe de très nombreux prouveurs de théorèmes. Beaucoup d'entre eux sont utilisés pour la vérification d'architectures (HOL [40], Nqthm et ACL2 [14, 52], PVS [92], Coq [96]). Ces prouveurs se distinguent par la logique mathématique employée, le style de preuve, la manière dont les procédures de décision automatique sont intégrées au système.

1.3 Vérification de modèle *vs.* vérification déductive

Ces deux approches sont fondamentalement différentes. Nous allons les comparer sur différents points.

- **Champ d'application.** Le champ d'application de la vérification déductive est très large. On peut aussi bien formaliser des systèmes matériels complexes que les propriétés que l'on veut vérifier sur ces systèmes. La preuve de théorème fournit un cadre de travail unifié en permettant de définir et raisonner sur des théories appropriées au problème et en permettant d'accomplir de nombreux types de vérification. Le champ d'application de la vérification de modèle est plus modeste, il ne s'applique que sur la logique et le type de modèle pour lequel il a été conçu, ainsi selon les propriétés que l'on veut prouver il faut choisir avec soin la logique temporelle employée.
- **Automatisation.** Comme son champ d'application est très large, la vérification déductive ne peut être automatisée que jusqu'à un certain degré ; elle nécessite généralement une intervention humaine pour guider le processus de déduction. En revanche, pour la vérification de modèle, on peut généralement fournir un algorithme entièrement automatique.
- **Hiérarchisation.** Dans le cas de vérification de matériel, on effectue une vérification hiérarchique. Les techniques de vérification déductive s'adaptent bien à la vérification hiérarchique car elles utilisent un mécanisme d'abstraction. Dans le cas de la vérification de modèle, la vérification hiérarchique est rendue beaucoup plus difficile, car la description du système se fait sur les états, d'une manière non hiérarchique. Il faut donc remodéliser le système à chaque niveau d'abstraction.
- **Description matérielle.** Les méthodes de vérification déductive et de vérification de modèle ne représentent pas le matériel de la même manière. L'approche par vérification déductive représente le matériel de manière structurelle. Elle permet de raisonner sur une spécification fonctionnelle et de raisonner sur des descriptions paramétrées. Pour la vérification de modèle, la description de matériel est fondée sur l'état, elle permet de représenter le comportement plutôt que la structure. La modélisation des systèmes qu'elle induit permet de représenter plus facilement des formalisations concernant des propriétés de concurrence, d'équité, de communication et de synchronisation ; elle permet de raisonner à propos des aspects contrôle du système.

1.4 Systèmes paramétrés

Un des problèmes actuels les plus importants dans le domaine de la vérification est celui de la vérification des systèmes paramétrés. Ces systèmes sont définis de manière générique en utilisant des paramètres pour représenter le nombre de processus ou la manière dont sont connectés les processus entre eux. Le problème de la vérification de systèmes paramétrés se place dans le cadre plus large des systèmes à états infinis. La vérification de tels systèmes s'est longtemps faite en utilisant des méthodes déductives. Cependant, depuis quelques années, des approches algorithmiques ont été développées qui fournissent des procédures de vérification ou plus généralement des semi-algorithmes demandant une contribution de l'utilisateur plus limitée que dans le cas des méthodes déductives.

Ces procédures permettent de calculer des ensembles infinis d'états atteignables à travers une représentation symbolique finie. La vérification de systèmes paramétrés est en général indécidable [4]. Pour pallier ce problème, deux catégories de méthodes se présentent :

- Soit on se restreint aux familles des systèmes paramétrés pour lesquelles le problème est décidable.
- Soit, on conçoit des méthodes consistantes mais incomplètes en espérant qu'une de ces méthodes s'appliquera au problème étudié.

Nous allons présenter ici quelques travaux relevant de l'une ou l'autre des catégories.

Construction de procédures de vérification pour certaines classes d'architecture. Pour la première catégorie de méthodes, nous citerons les travaux de Emerson et Namjoshi [33, 32] et ceux de German et Sistla [38]. Ces deux équipes ont montré qu'il était possible d'avoir un algorithme de vérification pour certaines familles de systèmes paramétrés. Dans [38], German et Sistla s'intéressent aux systèmes

paramétrés concurrents synchrones et à la preuve de propriétés locales aux processus. Ils étudient deux modèles. Le premier est constitué d'un processus de contrôle et de N processus utilisateurs identiques. Pour ce modèle, German et Sistla fournissent un algorithme pour prouver des propriétés temporelles sur le processus de contrôle. Le second modèle est constitué de N processus identiques. Dans ce cas, German et Sistla donnent un algorithme permettant de prouver une même propriété sur tous les processus du système (d'où le terme de propriété locale au processus). Certaines propriétés plus globales comme l'exclusion mutuelle peuvent aussi être vérifiées par leurs méthodes. Les travaux de Emerson et Namjoshi sont similaires. Ils s'intéressent eux aussi aux propriétés locales aux processus et élargissent leur étude à la preuve de propriété pour toutes paires de processus (sous certaines hypothèses de symétrie). Emerson et Namjoshi fournissent une procédure de décision pour des réseaux sous forme d'anneaux [32] et donnent un algorithme dans le cas des processus synchrones. Les différences entre ces travaux viennent principalement des méthodes de modélisation choisies.

Constructions de procédures de vérification semi-décidables. Les méthodes relevant de la seconde catégorie sont plus nombreuses. Kesten *et al.* développent des techniques de vérification de modèle régulier. Ces travaux sont exposés dans [55, 56]. Le langage d'assertion sous-jacent à la vérification de modèle régulier permet non seulement de représenter des assertions booléennes, mais surtout des expressions régulières. Ces expressions permettent de représenter l'état global d'un système, indépendamment de la valeur des paramètres. Par exemple, considérons un réseau linéaire paramétré $S(N)$ constitué de N processus se partageant un jeton. Nous supposons que l'état global de chaque processus est représenté par 1 si le processus a le jeton et 0 sinon. L'état initial global (ou configuration) du réseau se représente alors par l'expression régulière $I = 10^*$, ce qui signifie que le premier processus a le jeton et qu'aucun autre ne l'a. Dans ce modèle, les transitions sont vues comme des règles de réécriture. Ainsi, dans cet exemple, la transition peut-être $10 \rightarrow 01$. Kesten *et al.* appliquent ensuite les techniques de vérification de modèle symbolique.

Un défaut de la vérification de modèle régulier est qu'elle ne converge par toujours, même quand l'ensemble des états atteignables se représente par un langage régulier. Dans [78], Pnueli et Shahar étudient les causes fréquentes de cette absence de convergence : les relations de transition ne permettent d'effectuer qu'une action sur un seul processus à la fois. Ils fournissent des méthodes qui permettent d'accélérer les relations de transitions, en autorisant un nombre non borné de processus de changer simultanément d'état ou d'interagir avec leur voisin. Cependant ces méthodes demandent une intervention de l'utilisateur qui doit choisir quelle méthode utiliser pour son problème.

Induction et génération d'invariants. D'autres techniques de vérification s'appuient sur le principe d'induction et la génération d'invariant de réseau [60, 101, 44, 24, 64, 65]. Soit F une famille de processus, et soit ϕ une propriété à prouver pour tout processus de cette famille, soit $\preceq$ une relation de préordre sur les processus (préordre de simulation, préordre sur les traces, etc.) telle que :

$$P \preceq Q \wedge (Q \models \phi \implies P \models \phi)$$

où $Q \models \phi$ signifie que Q est un modèle pour ϕ. Il faut alors définir un processus I appelé invariant de réseau tel que pour tout processus P_i engendrant la famille F de processus

$$P_i \preceq I$$

ce qui permet d'obtenir

$$I \models \phi \implies \forall P \in F, P \models \phi$$

Cette technique a été utilisée entre autres par Halbwachs *et al.* dans [44] pour prouver des propriétés de sûreté sur des réseaux paramétrés. Nous reviendrons sur ce travail dans la partie 9. Cette technique de vérification soulève deux problèmes :

– comment exprimer la propriété ϕ de manière générale (sans tenir compte du nombre de composants),
– comment trouver des invariants de réseaux adaptés, s'ils existent.

Dans [24], Clarke *et al.* utilisent des techniques fondées sur les grammaires de réseaux et des techniques d'abstraction pour résoudre partiellement ces problèmes. Leurs réseaux sont définis à l'aide de grammaires de réseaux. Les propriétés sont spécifiées en CTL*, les assertions de base étant écrites dans un langage régulier L. L'état global des processus est spécifié à l'aide de mots, comme dans les travaux de Kesten *et al.*, qui peuvent ou non appartenir au langage régulier L. On peut ainsi partitionner les états globaux en classes d'équivalence selon les propriétés qu'ils possèdent : tous les états globaux possédant les mêmes propriétés sont dans la même classe. Ces classes représentent alors des états abstraits, et il ne reste qu'à abstraire les transitions. Clarke *et al.* obtiennent ainsi un système abstrait de transitions-états qui est plus grand pour le pré-ordre de simulation que tout système de la famille de processus. Ainsi pour toute formule CTL* vraie pour le système abstrait, la formule est vraie pour tout système de la famille de processus. Une fois le système abstrait calculé, les techniques classiques de vérification de modèle s'appliquent.

Les travaux de Lesens *et al.* [64, 65] proposent une autre approche de ces problèmes. Ils s'intéressent à un ensemble plus restreint de familles de systèmes, à savoir les réseaux linéaires et les réseaux sous forme d'arbre. Ils proposent une méthode de spécification différente fondée sur les observateurs synchrones. Un observateur synchrone est un processus capable d'observer le comportement d'un processus qui lui est associé sans changer son propre comportement. Une propriété de sûreté s'exprime en fournissant à chaque processus un observateur qui prend en entrée les entrées/sorties de son processus associé, ainsi que les observations fournies par l'observateur voisin. La construction de l'invariant se fait ici par rapport à un pré-ordre de traces. La synthèse de l'invariant revient à calculer le point fixe d'une équation. Lesens *et al.* proposent un ensemble d'heuristiques pour effectuer le calcul des points fixes. Cependant le calcul peut ne pas terminer et nécessite souvent l'intervention de l'utilisateur pour ajuster certains paramètres de calculs.

Vérification inductive. Nous présentons maintenant les travaux de Arons *et al.* [77, 5]. Cette équipe de recherche travaille aussi sur la vérification inductive, elle développe une méthode appelée vérification par invariant invisible. Arons *et al.* sont partis du constat suivant : dans de nombreux cas, il est possible de fixer la valeur du paramètre et si la propriété est prouvée pour cette valeur, elle l'est alors pour toutes les valeurs des paramètres. Ils génèrent automatiquement (et de manière transparente pour l'utilisateur) des assertions inductives ; c'est-à-dire des assertions vérifiées pour l'état initial et restant stables par transition. Ces assertions sont calculées en appliquant des généralisations simples et en effectuant des abstractions sur l'ensemble des états atteignables calculé pour la valeur fixée du paramètre. Pour montrer que les assertions sont inductives ils utilisent la vérification de modèle. Enfin, ils montrent que les assertions calculées impliquent la propriété étudiée. Ces travaux sont présentés en détail dans [77, 5]. Ces travaux se fondent sur des hypothèses de symétrie. D'autres travaux [34] utilisent la symétrie pour empêcher l'explosion du nombre d'états. Cependant ces techniques ne sont pas suffisamment puissantes (et donc ne réduisent pas suffisamment le nombre d'états) lorsqu'il y a des hypothèses d'équité.

Les efforts de recherche dans ce domaine sont maintenant tournées vers la production d'heuristiques qui s'appliquent à des classes spécifiques de système (processus connectés en anneau, en arbre, etc.).

Bien que la vérification de systèmes paramétrés soit indécidable, il est donc possible, pour certaines classes d'architectures, de construire des procédures de vérification ou plus généralement des semi-algorithmes. Si l'on se place dans un cadre plus général, les différentes méthodes proposées consistent à générer des invariants, et à utiliser des techniques d'abstraction et d'accélération. De nombreuses heuristiques de construction d'invariants sont proposées, mais l'utilisateur a toujours un rôle important pour ajuster la valeur de certains paramètres ou pour choisir des techniques à utiliser. Nos travaux se

placent dans le contexte des systèmes paramétrés, nous utiliserons aussi des invariants et des techniques d'accélération.

Nos travaux sont cependant plus ciblés : nous travaillons sur des systèmes bas niveau proches de l'implémentation. Ces systèmes sont décrits dans le modèle polyédrique, et notre preuve se fera sur cette description. Nous allons donc nous intéresser à des vérifications d'architectures. Plus précisément, nous allons maintenant nous tourner vers la vérification de description VHDL à l'aide du prouveur de Boyer et Moore.

1.5 Vérification de description VHDL à l'aide du prouveur de théorème de Boyer et Moore

Le système Prevail [13] développé par Borrione, Pierre et Salem est un environnement de preuve qui inclut un ensemble d'outils de vérification. Il prend en entrée deux descriptions VHDL [1] et vérifie l'équivalence (ou l'implication) de deux architectures différentes de la même entité (par exemple deux descriptions données à des niveaux d'abstractions différents). L'outil de vérification le plus adapté est ensuite choisi par l'utilisateur et les descriptions sont automatiquement traduites dans le formalisme du prouveur de théorème. Parmi ces outils de preuve, on trouve le prouveur de théorème de Boyer et Moore : Nqthm [14].

Le prouveur de théorème Nqthm se fonde sur une logique du premier ordre avec égalité. Les trois principes de base de ce prouveur sont les suivants :

- le principe de « shell » : ce principe permet de construire des types abstraits récursifs à partir d'un objet de base, d'un constructeur et d'un ou plusieurs déconstructeurs, de plus une fonction booléenne permet d'identifier les éléments du type.
- Le principe de définition : avant d'accepter la définition d'une fonction récursive, le système vérifie qu'il existe une mesure qui décroît par rapport à une relation d'ordre bien fondé.
- Le principe d'induction : ce principe permet de prouver des théorèmes inductifs sur des expressions récursives. Les schémas d'induction sont générés automatiquement par rapport aux définitions des fonctions récursives utilisées pour la preuve du théorème.

Pour automatiser les preuves d'équivalence de deux descriptions VHDL dans Nqthm, l'idée est de générer des schémas de preuve. Ces schémas dépendent des architectures décrites et il est donc nécessaire de fournir des modèles pour chaque classe d'architecture. Ces modèles permettront de générer le schéma de preuve adapté. Nous allons présenter ici un type de modèle qui concerne les architectures synchrones. Les modèles sont des fonctions. Ils doivent être tels qu'il est possible de générer automatiquement les fonctions à partir des descriptions VHDL étudiées.

Dans [76], Pierre a développé deux modèles pour les architectures synchrones et prouvé l'équivalence de ces deux modèles. Les deux modèles sont récursifs. Les circuits sont caractérisés par :

- Un vecteur d'entrées primaires, $I \subset \mathcal{I}$
- un vecteur de sorties primaires, $O \subset \mathcal{O}$
- un vecteur de variables d'états (vecteur d'éléments de mémoire), $S \subset \mathcal{S}$
- une fonction vectorielle d'états qui associe à un vecteur d'entrée primaire et un vecteur d'états, un autre vecteur d'état. C'est une fonction successeur.

$$\mathcal{I} \times \mathcal{S} \rightarrow \mathcal{S}$$

- une fonction vectorielle de sorties, qui associe à un vecteur d'entrée primaire et un vecteur d'états, un vecteur de sorties primaires.

$$\mathcal{I} \times \mathcal{S} \rightarrow \mathcal{O}$$

Le temps est un temps discret qui correspond aux fronts montants de l'horloge du circuit.

- Le premier modèle caractérise les entrées primaires comme des fonctions temporelles. Les variables d'état et les sorties primaires sont vues comme des fonctions d'ordre supérieur :

- domaine des vecteurs d'entrées primaires : $\mathbb{N} \rightarrow \mathcal{I}$
- domaine des vecteurs de variables d'état : $(\mathbb{N} \rightarrow \mathcal{I}) \rightarrow (\mathbb{N} \rightarrow \mathcal{S})$
- domaine des vecteurs de sorties primaires : $(\mathbb{N} \rightarrow \mathcal{I}) \rightarrow (\mathbb{N} \rightarrow \mathcal{O})$

  Comme Nqthm n'accepte pas la logique d'ordre supérieur, il faut le supprimer : les variables d'entrées ne sont plus considérées comme des variables temporelles, mais comme une liste de valeurs représentant l'histoire de ces variables. La récursion apparaît si le circuit contient des boucles structurelles (par exemple, variable dépendant d'elle même à un instant précédent).

– Dans le deuxième modèle, il y a une unique fonction pseudo-itérative, elle prend en entrée les entrées du système primaire et les variables d'états. Les variables d'entrées sont représentées par leur histoire. Le domaine de cette fonction est le suivant :

$$\cup_{k=1}^{\infty} \mathcal{I}^k \times \mathcal{S} \rightarrow \mathcal{S} \times \mathcal{O}$$

Pierre a montré à l'aide de Nqthm que ces deux modèles étaient équivalents, mais que le modèle pseudo-itératif était plus adapté tant du point de vue de l'efficacité que du point de vue de la traduction.

Dans [75], Pierre propose un modèle pour les architectures répliquées et parallèles. Une architecture répliquée est constituée d'un motif architectural qui est répété. Ces architectures sont génériques, c'est-à-dire que le nombre de répétitions du motif est représenté par un ou plusieurs paramètres. Pierre propose un modèle pour les architectures de dimension 1 et un modèle pour les architectures de dimension 2. Elle fournit ensuite un schéma pour générer des preuves à partir de ces modèles.

Cette modélisation a permis ainsi de prouver des circuits répliqués génériques.

En conclusion, les schémas de preuve ne peuvent être générés que pour certains types d'architecture. Les architectures doivent donc respecter un certain modèle. À chaque modèle correspond donc un schéma de preuve. Cependant si l'architecture ne respecte pas complètement un modèle, il n'est pas possible de générer un schéma de preuve. Les techniques de preuve que nous proposons dans ce mémoire sont totalement indépendantes de l'architecture choisie.

Nous allons maintenant nous tourner vers la vérification de réseaux systoliques qui sont des systèmes paramétrés particuliers. Les méthodes de vérification mises en œuvre dans ce cas sont très spécifiques.

2 Vérification d'architecture : les réseaux systoliques

Le but de cette section est donc d'étudier les différentes techniques de vérification des réseaux systoliques. Nous commencerons par définir les réseaux systoliques, puis dans la section suivante nous définirons quelques notions indispensables à la description de ces réseaux : les équations récurrentes. Les sections 2.3 et 2.4 seront consacrées à la vérification de tels réseaux. Nous présenterons tout d'abord les travaux d'Abdulla [2], puis ceux de Ling et Bayoumi [66]. Dans la dernière section, nous présenterons quelques travaux supplémentaires sur ce sujet.

2.1 Réseaux systoliques

La notion de circuit systolique a été introduite par Kung [59] au début des années quatre-vingts. Ces réseaux sont constitués de cellules toutes identiques qui ne peuvent réaliser qu'un nombre restreint de calculs. Les cellules sont reliées entre elles par un réseau géométrique régulier, elles ne peuvent communiquer qu'avec leurs voisines. Ainsi, une cellule reçoit des données d'une de ses voisines, fait un calcul et transmet le résultat à une voisine. Seules les cellules situées à la frontière du réseau communiquent avec l'extérieur. Les cellules évoluent en parallèle, sous le contrôle d'une horloge globale : pendant un temps de cycle, plusieurs calculs peuvent être effectués simultanément sur le réseau. Ces réseaux sont donc synchrones.

Ces circuits se prêtent à la conception modulaire. Ils sont appelés systoliques car le flot de données est régulier et uniforme.

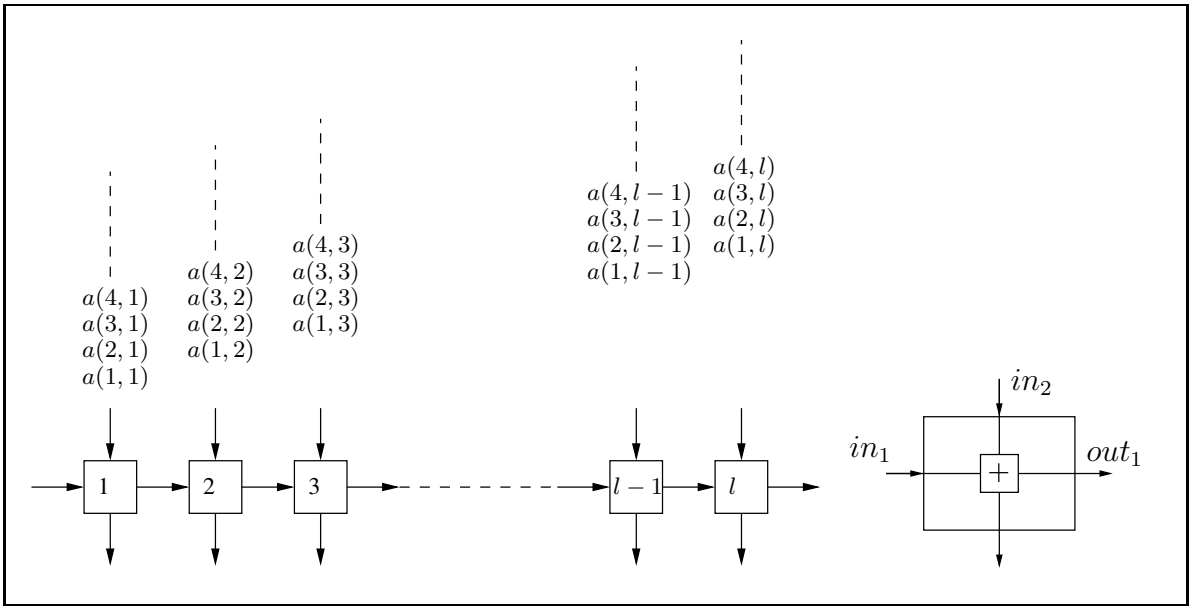


FIG. 1.2 – Réseau d'addition des colonnes d'une matrice ; une cellule de ce réseau

Plutôt que de donner une définition formelle [86] de ce qu'est un réseau systolique, nous préférons donner un exemple. Imaginons un système qui additionne les colonnes d'une matrice entre elles. Ce circuit reçoit en entrée les éléments d'une matrice ayant l colonnes et une infinité de lignes et en sortie le résultat de l'addition des colonnes entre elles. Le circuit est composé de l cellules, chacune ayant deux entrées et une sortie. L'élément $a(i,j)$ est reçu par la première entrée de la cellule j à l'instant $i+j-2$. À chaque temps de cycle (appelé top), toutes les cellules additionnent les valeurs reçues par leurs deux entrées et renvoient le résultat par leur sortie vers la cellule suivante. La deuxième entrée de la première cellule reçoit en continu la valeur 0. Le résultat de l'addition de la ligne i est donné par la sortie de la dernière cellule au bout de $l+i-1$ tops. Le système est représenté par le réseau de la figure 1.2.

Pour décrire ces systèmes nous pouvons utiliser ce que l'on appelle des équations récurrentes.

2.2 Préliminaires

Dans cette section, nous allons définir un certain nombre de notions qui nous seront utiles par la suite. Nous commencerons par définir la notion d'équation récurrente, puis nous présenterons l'arithmétique de Presburger

Équations récurrentes Soit l'équation suivante :

$$X(i) = \begin{cases} i = 0 : A(j \to j)(i) \\ i > 0 : X(j \to j-1)(i) + A(j \to j)(i) \end{cases} \tag{1.1}$$

Cette équation est une équation récurrente. Les variables A et X sont dites variables fonctionnelles, elles sont définies de $\mathbb{N} \to \mathbb{N}$, c'est-à dire que pour tout i appartenant à $\mathbb{N}$, $A(i)$ et $S(i)$ sont des entiers. L'équation 1.1 est dite récurrente car la variable fonctionnelle X est définie en fonction d'elle-même. Cette équation signifie simplement que pour tout point i la valeur de la variable fonctionnelle X au point i vaut

- si $i = 0$, la valeur de A au point image de i par la fonction $(j \to j)$, c'est-à-dire qu'elle vaut $A(0)$,
- sinon elle est définie comme étant la somme de la valeur de S au point image de i par la fonction $(j \to j-1)$ et de la valeur de A au point image de i par la fonction $(j \to j)$. Elle vaut donc $S(i-1) + A(i)$.

Les fonctions $(j \to j-1)$ et $(j \to j)$ peuvent être représentées à l'aide de matrices Nous donnons maintenant une première définition formelle d'une équation récurrente. Cette définition ne sera pas celle que nous utiliserons dans les chapitres suivants, nous donnerons la définition utilisée en section 3.

Équations récurrentes *Une* équation récurrente *définissant une variable fonctionnelle X de dimension n (une fonction de $Z^n \to T$ où T est un type) est une équation de la forme :*

$$X(\bar{z}) = \begin{cases} p_1(\bar{z}) : g_1(\dots, K(d_1(\bar{z})), \dots) \\ \vdots \\ p_m(\bar{z}) : g_m(\dots, K(d_m(\bar{z})), \dots) \end{cases}$$

où

- *$\bar{z}$ est un vecteur d'indices de dimension n ;*
- *d_i est une fonction de dépendance $d_i : \mathbb{Z}^n \to \mathbb{Z}^m$ de la forme $d_i(z) = A_i z + a_i$ où A_i est une matrice de dimension $n \times m$ à coefficients rationnels et a_i est un vecteur de dimension n à coefficients rationnels ;*
- *g_i est une fonction stricte*[1]*. Ces opérandes sont des variables fonctionnelles ou des variables de flots de la forme $K(d_i(\bar{z}))$;*
- *$p_i(\bar{z}))$ est un prédicat (ou garde) sur les entiers, tel que $g_i(\dots, X(d_i(\bar{z})), \dots)$ est bien défini : c'est-à-dire que $d_i(\bar{z})$ est un entier pour tout $\bar{z}$ vérifiant le prédicat p_i.*

On suppose que les prédicats sont tous disjoints. Les expressions de la forme $p_i(\bar{z}) : g_i(\dots, K(d_i(\bar{z})), \dots)$ seront appelées expressions gardées.

Un système d'équations récurrentes est un ensemble d'équations récurrentes :

$$\begin{aligned} X_1(\bar{z}) &= eq_1 \\ &\vdots \\ X_n(\bar{z}) &= eq_n \end{aligned}$$

Les seules variables fonctionnelles utilisées dans les équations récurrentes $eq_1, \dots eq_n$ sont les variables $X_1, \dots, X_n$.

Les gardes des équations sont des prédicats sur les entiers. Plus précisément, ces prédicats sont définis dans l'arithmétique de Presburger.

Arithmétique de Presburger L'arithmétique de Presburger est une logique du premier ordre sur les entiers munis de l'addition. Les formules de Presburger sont des formules construites en combinant des contraintes affines sur des variables entières avec les opérations logiques $\neg$, $\vee$, $\wedge$ et les quantificateurs $\forall$ et $\exists$. Les contraintes affines peuvent être soit des égalités, soit des inégalités. Le premier algorithme de décidabilité est dû à Presburger en 1929 dans [80, 81]. Il a prouvé que cette arithmétique était consistante et complète. De nombreux autres algorithmes permettent de calculer la valeur des formules de Presburger [82]. La borne supérieure de la complexité de ces algorithmes est $2^{2^{2^n}}$ où n représente la longueur de l'expression [74].

La bibliothèque Omega [53] permet de calculer des ensembles d'entiers et des relations définis à l'aide de formules de Presburger.

Nous présenterons ici deux travaux représentatifs de la vérification des circuits systoliques. Le premier de Abdulla [2] concerne les problèmes de décision, le second de Ling et de Bayoumi [66] fournit des procédures de vérification.

[1] Le résultat d'une fonction stricte n'est défini que si tous les opérandes de la fonction sont définis.

2.3 Problèmes de décision dans la vérification d'architectures systoliques

Ce travail a été réalisé par Abdulla au cours de sa thèse [2] et concerne les problèmes de décision dans la vérification des circuits systoliques. La première étape de ce travail est de fournir un modèle pour la description formelle de l'implémentation et la spécification des systèmes. Ce modèle se compose de systèmes d'équations récurrentes.

La description d'un circuit se fait en trois étapes : tout d'abord on décrit la topologie du circuit, puis l'interconnexion des cellules et enfin les calculs effectués dans une cellule. Pour illustrer ce travail, nous reprenons l'exemple d'addition des colonnes d'une matrice donné dans la section 2.1 et dans la figure 1.2.

La signature[2] de cet exemple est l'anneau des entiers.

– **Topologie.** La topologie spécifie comment les cellules sont placées dans le circuit. Elle est définie par un prédicat $top(\bar{x}, \bar{l})$ où $\bar{l}$ est appelé le paramètre du circuit. Pour chaque valeur de $\bar{l}$, on décrit un circuit de la famille de circuits associée au prédicat : à chaque cellule de calcul est associée un point entier $\bar{x}$ satisfaisant le prédicat. Dans notre exemple, le prédicat est

$$sommetop(x, l) = 1 \leq x \leq l$$

– **Interconnexion.** L'interconnexion spécifie comment les entrées et les sorties des cellules sont connectées. Une cellule a le même nombre d'entrées et de sorties. Les entrées et les sorties sont numérotées (dans notre exemple de 1 à 2). L'entrée i de la cellule placée au point $\bar{x}$ est connectée à la sortie i d'une autre cellule donnée par la fonction de connexion $\Delta(\bar{x}, \bar{l}) + \bar{x}$. Pour les cellules à la frontière, une (au moins) des entrées n'est pas connectée à la sortie d'une cellule. De telles entrées sont définies par une fonction d'entrée, qui associe à un entier correspondant à l'emplacement de la cellule une variable de flot ou une constante. La fonction d'entrée est définie par une équation récurrente qui peut être définie par plusieurs expressions gardées. Les gardes de ces expressions sont des prédicats disjoints deux à deux définissant la position $\bar{x}$ des cellules et les instants courants t où la fonction est définie par l'expression. Dans notre exemple, nous n'avons qu'une fonction de connexion, elle correspond à l'entrée 1.

$$\Delta_1(x, l) = -1$$

Notons $top'_i(\bar{x}, \bar{l})$ le prédicat qui définit la position des cellules dont la i-ème entrée ($Input_i$) est connectée à la i-ème sortie (out_i) d'une cellule voisine, et $top''_i(\bar{x}, \bar{l})$ le prédicat qui définit la position des cellules dont la i-ème entrée n'est pas connectée à la sortie d'une autre cellule. Les interconnexions sont alors définies de la manière suivante :

$$\begin{array}{rcl} top'_i(\bar{x}, \bar{l}) & \longrightarrow & in_i(\bar{x}, \bar{l}, t) = out_i(\Delta_i(\bar{x}, \bar{l}) + \bar{x}, \bar{l}, t) \\ top''_i(\bar{x}, \bar{l}) & \longrightarrow & in_i(\bar{x}, \bar{l}, t) = Input_i(\bar{x}, \bar{l}, t) \end{array}$$

Dans l'exemple étudié, nous obtenons les interconnexions suivantes :

$$\begin{array}{rcl} (2 \leq x \leq l) & \longrightarrow & in_1(x, l, t) = out_1(x - 1, l, t) \\ (x = 1) & \longrightarrow & in_1(x, l, t) = Input_1(x, l, t) \\ (1 \leq x \leq l) & \longrightarrow & in_2(x, l, t) = Input_2(x, l, t) \end{array}$$

Les fonctions d'entrées sont définies comme suit :

$$\begin{array}{rcl} Input_1(x, l, t) & = & 0 \\ Input_2(x, l, t) & = & \{x \leq t + 1\} : a(t - x + 2, x) \end{array}$$

[2]La signature correspond à la structure de l'ensemble des valeurs d'entrée des signaux : elle peut être un anneau sur les entiers, une algèbre, etc.

– **Calcul des cellules.** Ce calcul spécifie la manière dont les sorties et les variables locales d'une cellule dépendent à un certain instant des entrées et des variables locales de cette cellule aux instants précédents. À chaque top, les cellules effectuent un certain nombre de calculs. Certains des résultats de ces calculs sont envoyés aux cellules suivantes, d'autres sont gardés dans la cellule. Les calculs effectués dans une cellule dépendent de la position de la cellule. Ainsi, bien que les cellules soient similaires d'un point de vue architectural, elles peuvent effectuer des calculs différents. De même, une cellule peut effectuer des calculs différents selon l'instant considéré. La notion de délai est introduite pour séparer la phase d'initialisation de la suite des calculs. Un calcul de cellule est défini par une équation récurrente. Dans l'exemple de somme, voici les équations définissant les sorties :

$$\begin{aligned} out_1(x,l,t) &= \{x \leq t\} : in_1(x,l,t-1) + in_2(x,l,t-1) \\ out_2(x,l,t) &= \{x \leq t\} : in_2(x,l,t-1) \end{aligned}$$

Nous n'avons pas fait apparaître les phases d'initialisation, qui correspondent aux instants où les cellules ne contiennent pas encore de coefficients de la matrice.

La spécification d'un circuit décrit le comportement du système que l'on souhaite avoir. Elle décrit quelles sont les valeurs attendues de certaines entrées et sorties de cellules et de certaines variables locales. Cette spécification est une conjonction d'expressions gardées. Les expressions sont des égalités entre un signal et une variable fonctionnelle d'un système d'équations récurrentes. Quand la garde d'une des expressions est vraie, le signal de l'expression doit être égal à la variable fonctionnelle. Dans le cas de notre exemple, la spécification est donnée par :

$$sommespec(x,l,t) =$$
$$(x = l) \wedge (l \leq t) \longrightarrow out_1(x,l,t) = f(t-l+1,l)$$

et f est définie par :

$$f(y_1,y_2) = \begin{cases} \{y_2 = 1\} : a(y_1,y_2) \\ \{1 < y_2 \leq l\} : f(y_1,y_2-1) + a(y_1,y_2) \end{cases}$$

Dans les cas où la signature est une algèbre (*cf.* annexe B), Abdulla a associé à ce formalisme une sémantique (la limitation aux algèbres permet de couvrir les cas les plus courants). Cette sémantique est le plus petit point fixe de l'ensemble des fonctions du système d'équations récurrentes. Abdulla définit formellement ce que signifie : « la description de l'implémentation d'un circuit respecte la spécification du comportement de ce circuit ».

Abdulla a montré que toute valeur d'un signal dans un réseau systolique peut être décrite à l'aide d'équations récurrentes dont la signature est la même que celle du circuit. Les variables fonctionnelles de ce système d'équations correspondent aux variables de flots du réseau. Ainsi, le problème de vérification revient à montrer l'égalité de deux systèmes d'équations récurrentes.

Dans sa thèse, Abdulla étudie plusieurs problèmes de décision. Tout d'abord, il s'intéresse au cas où la signature est un anneau (algébrique) commutatif et étudie le problème de décision qui est de savoir si l'implémentation d'un circuit respecte sa spécification pour tout anneau (algébrique) commutatif. Ce travail s'applique à deux classes particulières de réseaux, les réseaux linéaires, et les réseaux en forme d'anneaux. La spécification de tels réseaux est donnée par des expressions définies sur des anneaux (algébriques). Pour de tels réseaux, la vérification se fait en deux étapes.

– Tout d'abord, Abdulla montre que les valeurs des sorties, des entrées et des variables locales de toute cellule peuvent à tout instant se décrire à l'aide d'expressions gardées définies sur un anneau. Pour cela, il introduit une classe de fonctions récursives et montre, pour toute fonction de cette classe, l'existence d'une expression gardée définie sur un anneau et égale à cette fonction. Puis, il

montre que toutes les valeurs des sorties, entrées et variables locales des cellules sont définies par de telles fonctions. Ainsi, la fonction de spécification est maintenant :

$$spec(\bar{x}) = (p_1(\bar{x}) \rightarrow (e_1(\bar{x}) = r_1(\bar{x}))) \wedge \cdots \wedge (p_m(\bar{x}) \rightarrow (e_m(\bar{x}) = r_m(\bar{x})))$$

Les p_i sont des prédicats, les expressions e_i sont des expressions gardées définies sur un anneau et les r_i des expressions sur des anneaux.

– La deuxième étape de vérification concerne la validité des expressions suivantes :

$$p(\bar{x}) \rightarrow e(\bar{x}) = r(\bar{x})$$

Comme $e(\bar{x})$ est une expression gardée, on montre d'abord que le prédicat p implique l'une des gardes. Comme les gardes sont des expressions définies dans l'arithmétique de Presburger, le problème est décidable. Ensuite, il faut montrer une expression de la forme suivante :

$$p(\bar{x}) \rightarrow r(\bar{x}) = 0$$

Abdulla définit une classe particulière de formules et montre que pour cette classe la vérification de l'expression précédente est décidable. Il montre de plus que pour des anneaux courants (entiers, réels, naturels modulo m), on peut décider si une formule fait partie de cette classe.

Abdulla a étendu ses travaux à d'autres types d'algèbre tout en restreignant la structure des circuits. Pour une classe particulière concernant des algèbres booléennes, il montre ainsi des résultats similaires à ceux sur les classes d'anneaux pour certains types de réseaux systoliques linéaires (correspondant par exemple à des circuits de convolutions).

Ce travail est le premier à présenter des résultats sur des problèmes de décision dans les réseaux systoliques. Le travail que nous présenterons ce placera dans le cas où la signature est l'algèbre des booléens. Cependant, nous nous intéresserons à une classe plus large de circuits. Nos circuits devront s'exprimer à l'aide d'équations récurrentes, et ils devront être ordonnançables (c'est-à-dire que l'on pourra associer une fonction d'ordre sur l'exécution des calculs).

2.4 Spécification et vérification de réseaux systoliques avec STA

Ling et Bayoumi [66] ont développé un formalisme nommé STA (Systolic Temporal Arithmetic) pour spécifier et vérifier des architectures systoliques. Ce formalisme fournit une description complète et précise de la conception des réseaux et une sémantique pour la vérification. Il exploite les caractéristiques des réseaux pour produire des notations et des méthodes de raisonnement adaptées à ces réseaux. Les constructeurs et opérateurs utilisés se fondent sur la logique ITL (Interval Temporal Logic) qui permet des raisonnements à différents niveaux. Ce formalisme permet de manipuler le parallélisme et le pipeline des réseaux. De plus, tout en offrant des raisonnements mathématiques, il peut permettre de simuler, de trouver des erreurs ou de générer des tests. La spécification d'un réseau systolique est la suivante :

– **Spécification de la structure.** Il faut tout d'abord préciser la place des cellules dans le réseau et la manière dont elles sont connectées. Voici par exemple la spécification de la structure du système d'addition des colonnes d'une matrice :

$$\forall 1 \leq i \leq l,\ \textit{Cell}(B_i)$$
$$\forall 1 \leq i \leq l-1,\ \textit{Conn}(\textit{Out}(out_1, B_i), \textit{In}(in_1, B_{i+1}))$$

  – *Out*, et *In* sont des identificateurs : $\textit{Out}(out_1, B_i)$ désigne la sortie out_1 du composant B_i et $\textit{In}(in_1, B_{i+1})$ désigne l'entrée in_1 du composant B_{i+1}
  – *Cell*, *Conn* sont des prédicats. Le prédicat $\textit{Cell}(B_i)$ signifie que B_i est une cellule. Nous avons donc un circuit de l cellules identiques. Le prédicat $\textit{Conn}(\textit{Out}(out_1, B_i), \textit{In}(in_1, B_{i+1}))$ spécifie que la sortie out_1 de la cellule B_i est directement connectée à l'entrée in_1 de la cellule B_{i+1}.

– **Spécification du comportement.** Il faut ensuite spécifier le comportement des cellules, le comportement des entrées, et le comportement des sorties. Voici par exemple, la spécification du comportement des cellules du système d'addition des colonnes d'une matrice :

$$\forall B_i,\ \mathit{Cell}(B_i) \wedge \mathit{Val}(\mathit{In}(in_1, B_i), y_{il}) \wedge \mathit{Val}(\mathit{In}(in_2, B_i), a_{il}) \rightarrow Val(Out(out_1, B_i), \_, (y_{il} + a_{il}))$$

Le prédicat $\mathit{Val}(\mathit{In}(in_1, B_i), y_{il})$ signifie que la valeur de la sortie in_1 de la cellule B_i est y_{il}. Ainsi, la spécification du comportement de la cellule signifie que, si en entrée de la cellule les signaux valent y_{il} et a_{il}, alors le signal de sortie vaudra $(y_{il} + a_{il})$, sauf pour le premier instant où la valeur est inconnue (_).
Nous spécifions maintenant le comportement des entrées. Nous avons deux types d'entrées. Celui correspondant à l'entrée in_1 de la cellule B_1 et celui correspondant à l'entrée in_2 des cellules $B_1, \ldots, B_l$. Il y a donc deux prédicats

$$\mathit{Val}(\mathit{In}(in_1, B_1), \Box 0)$$
$$\forall 1 \leq i \leq l,\ \mathit{Val}(\mathit{In}(in_1, B_i), \wedge_{r=0}^{i-2} \_, \wedge_{r=1}^{k-1} a_{ri}, \Box a_{ki})$$

Le premier prédicat définit l'entrée in_1 de la première cellule, elle vaut toujours 0 ($\Box 0$). Le second prédicat définit l'entrée in_2 de toutes les cellules : pour la cellule i, il y a tout d'abord $i-1$ valeurs quelconques, puis nous avons les coefficients de la colonne i de la matrice. Il y a une infinité de coefficients.
Enfin, nous donnons la spécification des sorties :

$$Val(Out(out_1, B_l), \wedge_{r=0}^{l-1} \_, \wedge_{r=1}^{k-1} \Sigma_{i=1}^{l} a_{ri}, \Box \Sigma_{i=1}^{l} a_{ki})$$

En sortie, nous avons l'addition de tous les coefficients d'une ligne de la matrice. Il y a une infinité de résultats.

La spécification de la structure et du comportement du système d'addition des colonnes d'une matrice est assez proche de celle proposée par Abdulla. Cependant, comme Ling et Bayoumi n'utilisent pas directement la notion d'équations récurrentes, leurs notations sont plus complexes, et ils doivent spécifier les sorties à l'aide d'une somme de l termes. La notion d'équations récurrentes est cependant sous-jacente à leur formalisme.

En plus de la syntaxe et de la sémantique, Ling et Bayoumi ont prouvé des théorèmes permettant de réécrire de manière sémantiquement équivalente les spécifications. Ils possèdent aussi des axiomes et des règles de réécriture.

La vérification formelle d'un réseau au niveau architecture revient alors à prouver la correction de l'implication suivante :

spec. structure$\wedge$spec. comportement cellule$\wedge$spec. comportement entrées $\Rightarrow$ spec. comportement sorties

Ling et Bayoumi ont développé différentes techniques de vérification utilisant les axiomes, les règles et les théorèmes de réécritures :

– **Dérivation et comparaison :** cette méthode est directe, elle exploite la régularité et la localité des réseaux. À partir des cellules du réseau les plus proches des entrées, on fait dériver les sorties des cellules à partir des entrées. Ces sorties deviennent alors des entrées pour les cellules suivantes. On répète le processus jusqu'à arriver aux sorties. Puis, on compare le comportement dérivé au comportement de sortie spécifié. S'ils sont égaux le réseau est correct, sinon il n'est pas correct. Cette méthode peut être appliquée à certains réseaux paramétrés en utilisant les règles, les axiomes et théorèmes prédéfinis. L'utilisation de ces règles, axiomes, et théorèmes permet de cacher une induction, et d'obtenir une dérivation finie.

– **Par induction :** la méthode précédente est trop longue si le réseau est grand. Dans ce cas, on peut définir une hypothèse de récurrence $P(n)$:

  spécification structure $\wedge$ comportement cellule $\wedge$ comportement entrées $\Rightarrow$ comportement sorties

  pour un réseau de taille n. On prouve alors le cas de base pour un réseau de taille n_0, puis on suppose $P(k)$ correct pour $k \geq n_0$ et on montre $P(k+1)$ en utilisant les prédicats définis lors de la spécification et les théorèmes, axiomes et règles prédéfinies.
– **Résolutions d'équations différentielles STA :** les deux méthodes précédentes ne peuvent pas être appliquées dans le cas de réseaux systoliques bidirectionnels. La méthode est la suivante : pour une cellule quelconque, il faut tout d'abord générer des équations reliant les sorties aux variables d'entrée, à l'aide de variables temporelles. Puis, le système est résolu en utilisant les règles de réécritures, ce qui permet d'obtenir une relation entre les entrées et les sorties du réseaux. Il ne reste qu'à substituer des variables d'entrée par leur comportement défini dans la spécification, et à comparer le résultat obtenu au comportement spécifié.

Ces méthodes sont implémentées en Prolog. Elles ont été appliquées avec succès sur des exemples complexes (décomposition LU). Mais Ling et Bayoumi ne donnent pas de cadre précis pour lequel ces méthodes s'appliquent avec succès. De plus, la vérification ne peut être effectuée que sur une description physique du système. La vérification ne se fait donc qu'à la fin de la conception ce qui peut être préjudiciable si l'erreur a lieu au début de la conception. Dans la méthode que nous proposons, la vérification peut avoir lieu à n'importe quel niveau de la conception. De plus nous ne nous intéresserons pas au même type de propriétés, notre but sera de prouver des propriétés sur des signaux de contrôle introduits lors de la phase de synthèse.

2.5 Autres travaux sur les réseaux systoliques

D'autres travaux ont été effectués sur ce sujet. En ce qui concerne la spécification, nous citerons les travaux de Zhou et Burleson [102] qui fournissent une description formelle, une sémantique et une classification de propriétés d'équivalence. Hennessy [47] propose un langage de description pour les réseaux systoliques. Ce langage permet de décrire la spécification et l'implémentation des systèmes. Pour prouver que l'implémentation respecte la spécification, il utilise des transformations syntaxiques et une méthode d'induction.

Les langages proposés pour la spécification sont tous différents, mais ils font généralement appel à la notion d'équations récurrentes. Tous ces travaux sont tournés vers la preuve d'équivalence, et selon le formalisme utilisé, ils peuvent caractériser les systèmes sur lesquels des techniques de vérification peuvent être appliquées.

Dans tous les travaux que nous avons présentés le principe est le même. De manière schématique, le point de départ est une description d'architectures et de la spécification qui est un ensemble d'équations récurrentes. Il faut ensuite dériver l'architecture pour obtenir des équations récurrentes. Les travaux que nous allons maintenant présenter se placent dans une optique différente. La description de l'architecture et la spécification (dans le cas de vérification hiérarchique) se font directement à l'aide d'équations récurrentes. Pour obtenir le circuit on synthétise la description qui a été vérifiée. La phase de synthèse conserve généralement la sémantique. Si des transformations ne préservant pas la sémantique sont effectuées, on applique alors les techniques de vérification hiérarchiques. Il faut pour cela un formalisme permettant de décrire des équations récurrentes et de synthétiser facilement une architecture. Ce formalisme est le modèle polyédrique.

3 Le modèle polyédrique

Nous nous plaçons maintenant dans un cadre plus général que celui des réseaux systoliques. Nous allons nous intéresser à la vérification de systèmes réguliers et paramétrés décrits dans le modèle polyédrique.

3.1 Présentation

Le modèle polyédrique est un formalisme qui permet de représenter des calculs massivement parallèles en fournissant une représentation compacte. Il permet l'utilisation de paramètres symboliques. Grâce à ce modèle, nous pouvons donner une description haut niveau (proche du niveau algorithmique) et par une série de transformations, nous pouvons effectuer une synthèse et obtenir une description bas niveau décrivant une architecture VLSI régulière. Ce modèle est aussi utilisé dans le cadre de la parallélisation de boucles imbriquées.

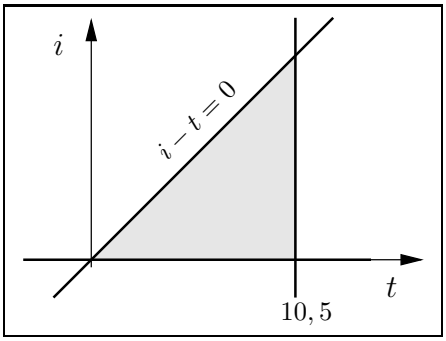
Ce modèle se fonde sur les systèmes d'équations récurrentes affines définies sur des domaines polyédriques. Les systèmes d'équations récurrentes uniformes ont été introduits en 1967 par Karp, Miller, et Winograd [51] pour dériver des architectures parallèles. Nous présenterons ici une extension aux équations récurrentes affines [97].

Soit $\{t, i \in \mathbb{Q} \mid 0 \leq i \leq t \leq 10,5\}$ un ensemble (*cf.* fig 1.3). Cet ensemble est délimité par trois hyperplans (en l'occurrence simplement des droites car nous sommes en dimension 2) :

- la droite $\{t, i \mid t = 10,5\}$,
- la droite $\{t, i \mid i - t = 0\}$ et
- la droite $\{t, i \mid -i = 0\}$.

C'est ce que nous appelons un polyèdre. Ce polyèdre peut être représenté de manière plus compacte par l'ensemble :

$$\{t, i \in \mathbb{Q} \mid \begin{pmatrix} 1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} t \\ i \end{pmatrix} \leq \begin{pmatrix} 10,5 \\ 0 \\ 0 \end{pmatrix}$$



$\mathcal{P} = \{t, i \in \mathbb{Q} \mid 0 \leq i \leq t \leq 10,5\}$

FIG. 1.3 – Un polyèdre $\mathcal{P}$

Définition 1.1 (Polyèdre) *Un* polyèdre *de dimension n est un sous espace $\mathcal{P}$ de $\mathbb{Q}^n$ borné par un nombre fini d'hyperplans. On peut le définir implicitement par :*

$$\mathcal{P} = \{x \in \mathbb{Q}^n | A.x \leq b\}$$

où A est une matrice (m, n) et b un vecteur de dimension m de rationnels.

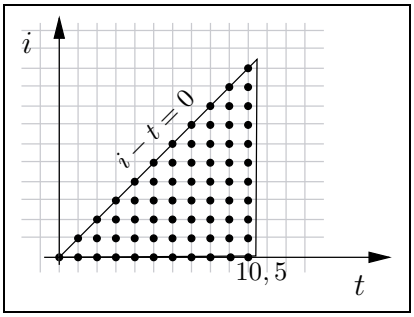


FIG. 1.4 – Un polyèdre entier

Nous nous intéresserons simplement aux points du polyèdre appartenant à $\mathbb{Z}^n$. C'est ce que nous appellerons un polyèdre à points entiers. Sur la figure 1.4, nous avons représenté les points du polyèdre $\mathcal{P}$ appartenant à $\mathbb{Z}^n$.

Définition 1.2 (Polyèdre à points entiers entiers) *On appelle* polyèdre à points entiers entiers *de dimension n un ensemble $\mathcal{D}$ défini par :*

$$\mathcal{D} = \{x \in \mathbb{Z}^n, z \in \mathcal{P}\} = \mathbb{Z}^n \cap \mathcal{P}$$

où $\mathcal{P}$ est un polyèdre de dimension n.

Définition 1.3 (Domaine) *On appelle* domaine polyédrique *(ou simplement* domaine*) de dimension n une union finie de polyèdres à points entiers de dimension n.*

La notion de variable polyédrique ou plus simplement de variable est similaire à celle de variable fonctionnelle vue dans la section 2.2. Les différences portent principalement sur les gardes, qui ne sont plus quelconques mais définies par un polyèdre.

Définition 1.4 (Variable) *Une* variable *X est une application d'un domaine dans un ensemble de valeurs. Ces valeurs peuvent être de type entier, booléen ou réel. On appelle instance de la variable X la restriction de X à un point de son domaine. Si z est un point du domaine de X, on notera $X[z]$, l'instance correspondant à ce point. On peut l'assimiler à une variable scalaire. Les variables constantes sont associées au domaine $\mathbb{Z}^0$.*

Par exemple, l'expression suivante

$$\forall (t,i) \in \{t,i \mid 0 \leq i \leq t\}, Y[t,i] \in \mathbb{B}$$

définit une variable Y et un ensemble d'instances entières $Y[t,i]$.

Nous pouvons définir la variable Y de la manière suivante :

$$Y = \begin{cases} \{t,i \mid i = 0\} : \textit{True} \\ \{t,i \mid 0 < i \leq t\} : Y.(t,i \rightarrow t-1, i-1) \wedge X.(t,i \rightarrow t-1,i) \end{cases}$$

où X est une variable polyédrique définie sur le domaine $\{t,i \mid 0 \leq i \leq t\}$ Cette équation signifie simplement que :

- si le point (t,i) appartient à $\{t,i \mid i = 0\}$, alors l'instance $Y[t,i]$ est définie par la constante *True*,
- sinon si elle appartient au domaine $\{t,i \mid 0 < i \leq t\}$, l'instance $Y[t,i]$ est définie par la conjonction de l'instance de Y au point $(t-1,i-1)$, point image de (t,i) par l'application $(t,i \rightarrow t-1,i-1)$ et de l'instance de X au point $(t-1,i)$, point image de (t,i) par l'application $(t,i \rightarrow t-1,i)$.

Ces applications sont appelées des dépendances.

Nous donnons maintenant la définition formelle d'une équation récurrente. La différence par rapport à la définition donnée en section 2.2 portent principalement sur les gardes, définies ici par un polyèdre. De pus les dépendances sont à coefficients les entiers et non dans les rationnels.

Définition 1.5 (Équations récurrentes) *Une* équation récurrente *définissant une variable X en tout point z d'un domaine D est une équation de la forme :*

$$X = \mathcal{D}_X : g(\dots, Z.d, \dots)$$

où

- *Z est une variable polyédrique (cela peut être la variable X),*
- *d est une fonction de dépendance $d : \mathbb{Z}^n \to \mathbb{Z}^m$;*
- *g est une fonction stricte. On l'écrit généralement comme étant une expression faisant intervenir des opérandes de la forme $Y.d$, des opérateurs de base et des parenthèses.*
- *$\mathcal{D}_X$ est un ensemble de points de $\mathbb{Z}^n$. C'est le domaine de l'équation, il correspond ici au domaine de la variable X.*

Une variable peut être définie par plusieurs équations définies sur des domaines disjoints. Dans ce cas, le domaine de la variable est l'union des domaines des équations.

La fonction de dépendance utilisée pour définir la variable Y peut s'écrire sous forme matricielle :

$$(t, i \to t-1, i-1)(x, y) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix}$$

Définition 1.6 (Équations récurrentes affines) *Une équation récurrente est dite affine si ses fonctions de dépendances s'écrivent sous la forme $d(z) = Az + a$ où A est une matrice de dimension $n \times m$ à coefficients entiers et a est un vecteur de dimension m à coefficients entiers. Si A est la matrice identité, l'équation est dite récurrente uniforme.*

Définition 1.7 (Système) *Un* système *est vu comme la donnée d'un ensemble de variables d'entrée In, d'un ensemble de variables de sorties Out, d'un ensemble de variables locales et d'un ensemble d'équations récurrentes définissant de manière unique chaque instance des variables locales et des variables de sorties.*

Définition 1.8 (Dépendance) *On dit qu'une instance x dépend directement de y s'il existe une équation telle que $x = g(\dots, y, \dots)$. Nous noterons $x \to y$ le fait que l'instance x dépend de l'instance y.*

En reprenant notre exemple, nous avons les dépendances suivantes :

$$Y[t, i] \to Y[t-1, i-1]$$
$$Y[t, i] \to X[t-1, i]$$

Ces dépendances peuvent être représentées à l'aide d'un graphe de dépendance.

Définition 1.9 (Graphe de dépendance) *On appelle* graphe de dépendance*, le graphe dont les nœuds sont des instances. Il y a un arc entre deux instances si l'une dépend de l'autre.*

La notion de dépendance peut être définie au niveau des variables. Elle se représente à l'aide d'un graphe.

Définition 1.10 (Graphe de dépendance réduit) *On appelle* graphe de dépendance réduit *, le graphe dont les nœuds sont des variables. Il y a un arc de la variable X vers la variable Y s'il existe une instance x de X et une instance y de Y telles que x dépend de y. On notera de même $X \to Y$.*

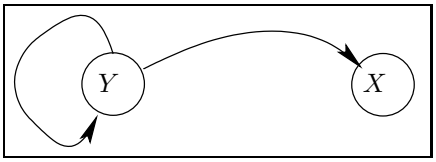


FIG. 1.5 – Graphe de dépendance réduit.

Dans notre exemple la variable Y dépend d'elle même et de la variable X, nous avons le graphe de dépendance représenté dans la figure 1.5 (que l'on note aussi $Y \to Y$ et $Y \to X$).

Nous appelons ce type de dépendance une auto-dépendance.

Définition 1.11 (auto-dépendance) *On appelle auto-dépendance de X une dépendance d telle que $X.d$ est une sous-expression de l'expression définissant X.*

Nous définissons maintenant la notion d'ordonnancement qui permet de préciser l'instant où une instance doit être calculée. Reprenons notre exemple comme nous avons la dépendance suivante $Y[t,i] \to Y[t-1, i-1]$, l'instance $Y[t-1, i-1]$ doit être calculée avant $Y[t,i]$. Nous pouvons associer à la variable Y la fonction qui à tout point (t,i) du domaine de Y associe le point t. Cette fonction est la fonction d'ordonnancement de Y, à l'instant t_0 toutes les instances de Y de la forme $Y[t_0, i]$ seront calculées.

Définition 1.12 (Ordonnancement) *Un* ordonnancement t_X *est une fonction telle que $t_X(z)$ spécifie l'instant où l'instance $X[z]$ est calculée. Généralement, la fonction t_X est définie du domaine de la variable dans $\mathbb{N}$, mais tout ensemble d'arrivée muni d'un ordre total est autorisé (par exemple $\mathbb{N}^k$ muni de l'ordre lexicographique). Dans la suite nous nous limiterons aux ordonnancements à une dimension. Un ordonnancement est valide si pour toutes instances $X[z]$ et $Y[z']$,*

$$X[z] \to Y[z'] \implies t_X(z) \leq t_Y(z')$$

Un système est dit ordonnançable *si son graphe de dépendance est sans cycle et si un ordonnancement valide existe. Déterminer si un Système d'Équation Récurrentes Affines (SERA) est ordonnançable ou non est indécidable dans le cas général [90], mais il existe de nombreuses heuristiques pour trouver des ordonnancements respectant un format particulier.*

Remarque : L'ordonnancement trivial qui à tout point associe la valeur 0 est valide. Un tel ordonnancement conduit à une implémentation de circuit où tous les opérateurs sont combinés en cascade. Les valeurs des variables de sortie sont calculées durant un unique cycle. Naturellement, un tel ordonnancement est invalide dès que l'on ajoute des contraintes sur la longueur maximale d'un chemin critique.

Transformation de système. Le modèle polyédrique est la combinaison de domaines polyédriques et d'équations de récurrences affines. Les propriétés de clôture dans ce modèle (plus précisément par intersection, union finie, et inverse d'application affine) sont la clef de voûte d'un riche ensemble de transformations préservant sémantique. Parmi toutes les transformations que l'on peut effectuer sur un SERA, la réindexation ou changement de base des variables est fondamentale ; cette transformation doit avoir un inverse à gauche pour tout les points du domaine de la variable. Soit X une variable d'un SERA définie de la manière suivante :

$$X_p[z] \quad = \quad \left\{ \begin{array}{ccc} & \vdots & \\ \mathcal{D}_{X_{p,i}} & : & g_i(\ldots X_q[d_i(z)]\ldots) \\ & \vdots & \end{array} \right.$$

Appliquer un changement de base $\mathcal{T}$ (où $\mathcal{T}$ est une application inversible à gauche de $\mathbb{Z}^n$ dans $\mathbb{Z}^n$) à la variable X consiste en :

– remplacer chaque $\mathcal{D}_{X_i}$ par $\mathcal{T}(\mathcal{D}_{X_i})$,
– dans la partie droite de l'équation de X, remplacer chaque dépendance d_i par $d_i \circ \mathcal{T}^{-1}$, la composition de d_i et de $\mathcal{T}^{-1}$,
– pour toutes les occurrences $X[g(z)]$ dans la partie droite de *toutes* les équations du système, remplacer la dépendance g par $\mathcal{T} \circ g$.

Le système ainsi obtenu est sémantiquement équivalent au système original.

Les programmes que nous étudions sont tous ordonnançables[3]. De plus, nous supposons que pour un système donné, un ordonnancement a été déterminé et qu'il a été appliqué, c'est-à-dire qu'un changement de base a été effectué sur tous les domaines et dépendances des variables afin d'identifier un indice particulier (le premier) qui sera appelé indice temporel ou temps. Il sera noté t. Un tel système est appelé *système ordonnancé*.

Nous présenterons dans le chapitre suivant un langage, ALPHA, et sa sémantique, qui permettent d'exprimer de tels programmes. Nous présenterons aussi l'environnement MMALPHA qui permet de manipuler ces programmes, de les transformer, de générer du code ou des architectures.

3.2 Premiers travaux de vérification dans le modèle polyédriques

Les premiers travaux de vérification avec le modèle polyédrique ont été réalisés par Quinton et Dezan [28]. Le but de ce travail est de prouver l'équivalence entre une spécification et une implémentation d'une architecture régulière. Pour cela, Quinton et Dezan utilisent une technique de rencontre au milieu. Le travail est effectué en deux étapes :

– La première étape consiste en une phase descendante à partir de la spécification en utilisant des techniques de synthèse.
– La seconde partie consiste en l'utilisation des techniques ascendantes d'abstraction à partir de l'implémentation. L'idée est de simplifier la description de l'architecture pour obtenir une représentation au même niveau que la représentation obtenue par les techniques de synthèse. Ces techniques utilisent des principes d'induction.

Quand les deux descriptions sont au même niveau, elles sont comparées en utilisant la sémantique fournie par le modèle polyédrique.

Ce travail est intéressant, mais reste limité aux cas où l'implémentation et la spécification se fondent sur le même algorithme. De plus, ce processus est semi-automatique, puisque c'est à l'utilisateur de choisir les transformations qui seront automatiquement appliquées.

3.3 Équivalences de deux spécifications

Dans cette partie, nous présentons des résultats sur l'équivalence de systèmes d'équations affines. Lors de la synthèse d'un système, on peut être amené à appliquer des transformations ne préservant pas obligatoirement la sémantique. Dans ce cas, il est important de vérifier que le système initial et le système transformé sont équivalents. Nous présentons deux résultats sur ce sujet. Le premier est dû à Barthou, Feautrier et Redon [8] et le second à Shashidhar, Bruynooghe, Catthoor et Janssens [93].

3.3.1 Équivalence de deux systèmes d'équations récurrentes affines

Le premier résultat de ce travail [8] montre que le résultat d'équivalence de deux SERAs est indécidable. Les auteurs proposent ensuite un semi-algorithme pour tester l'équivalence de deux SERAs.

Indécidabilité : pour prouver l'indécidabilité, les auteurs se ramènent au dixième problème de Hilbert qui est de décider si une équation diophantienne a une solution positive. Les auteurs se donnent

[3] Les autres systèmes sont inutiles car ils ne peuvent pas être implémentés.

un polynôme diophantien $P(X_1, \ldots, X_n)$. Ce polynôme peut être écrit comme la différence de deux polynômes dont les coefficients sont des entiers positifs :

$$P(X_1, \ldots, X_n) = P_+(X_1, \ldots, X_n) - P_-(X_1, \ldots, X_n)$$

- D'une part, trouver une solution à l'équation diophantienne $P(X_1, \ldots, X_n) = 0$ revient à trouver des valeurs de $X_1, \ldots, X_n$ telles que P_+ et P_- sont égaux. Un problème équivalent est de décider si pour toute valeur positive de $X_1, \ldots, X_n$, P_+ est différent de P_-.
- D'autre part, deux SERAs sont équivalents si pour toute valeur de leur paramètres, leurs sorties sont égales si leurs entrées sont égales.

Les deux problèmes sont reliés de la manière suivante : les auteurs construisent deux SERAs S et S'. Les deux SERAs S et S' ont deux entrées I_1 et I_2 et une seule sortie. Dans S, la sortie dépend seulement de l'entrée I_1. Dans S', la sortie dépend de l'entrée I_2 si, pour une valeur des paramètres $x_1, \ldots, x_n$ des SERAs, $P_+(x_1, \ldots, x_n) = P_-(x_1, \ldots, x_n)$; sinon la sortie dépend de I_1. La construction du deuxième SERAs est faite de la façon suivante : on associe aux polynômes P_+ et P_- deux ensembles de points entiers ; les dépendances alternent itérativement entre les deux ensembles de points, à la fin de l'itération la sortie dépendra soit de I_1 soit de I_2. Les deux SERAs seront équivalents si et seulement si leurs sorties ne dépendent que de I_1, ce qui reviendrait à décider si $P_+(x_1, \ldots, x_n) = P_-(x_1, \ldots, x_n)$.

Algorithme : les auteurs donnent ensuite un semi-algorithme. Un automate à mémoire d'états est associé à chaque paire de SERAs pour lesquels l'équivalence doit être vérifiée, de telle sorte que tester l'équivalence de ces deux SERAs revient à calculer un problème d'atteignabilité dans l'automate correspondant.

Les automates à mémoire d'états (MSA) ont été introduits par Boigelot et Wolper dans [12]. Ce sont des automates à états finis, dont les états sont composés d'un élément appartenant à un ensemble fini et d'un vecteur d'entiers notés $\langle p, v_p \rangle$. Une transition est un triplet $\{p, q, F_{pq}\}$ où p et q sont respectivement l'état de départ et l'état d'arrivée, et F_{pq} la relation de transition entre vecteurs, c'est-à-dire une partie de $\mathbb{Z}^{n_p} \times \mathbb{Z}^{n_q}$, où n_p (resp. n_q) est la dimension du vecteur v_p (resp. v_q). Il y a une transition entre l'état $\langle p, v_p \rangle$ et $\langle q, v_q \rangle$ si $\langle v_p, v_q \rangle$ est dans F_{pq}. Un état $\langle p, v_p \rangle$ est accessible s'il existe une séquence finie de transitions depuis l'état initial aboutissant dans cet état. L'ensemble des états accessibles de p, noté A_p est l'ensemble des vecteurs v_p tels que $\langle p, v_p \rangle$ est accessible. L'atteignabilité dans un MSA est indécidable mais, pour certains cas, il existe un algorithme.

Le MSA associé à la paire de SERA dont on veut connaître l'équivalence est construit de la manière suivante. Chaque état est étiqueté par une équation $e(i) = e'(i')$ où e (respectivement e') est une sous-expression du SERA de droite (respectivement de gauche) et i et i' sont des vecteurs d'itérations distincts ; le vecteur d'états est noté $\langle i, i' \rangle$. Les étiquettes sont des équations qui peuvent être vraies ou fausses quand les vecteurs sont remplacés par leur valeur. Les auteurs construisent un MSA de sorte que l'étiquette soit vraie pour tous les vecteurs dans l'ensemble accessible. Inversement, si une étiquette est fausse, son ensemble accessible doit être vide.

Les auteurs décrivent par la suite comment calculer des relations d'atteignabilité quand il est possible de le faire. Les clôtures transitives n'apparaissent que s'il y a des cycles dans le MSA. La difficulté du calcul d'atteignabilité dépend seulement de la structure du MSA. Dans le cas où il n'y a pas de cycle, on peut calculer les relations d'atteignabilité. Pour les cycles simples (des chemins commençant et finissant sur le même nœud et ne visitant qu'une fois chaque nœud du chemin), Boigelot [11] a montré une condition suffisante pour calculer la relation d'atteignabilité. Dans le cas de cycle complexe, de nombreuses heuristiques peuvent être appliquées. Un prototype implémentant le semi-algorithme a été construit. Il utilise la Polylib [99] et Omega [53].

Ce travail offre un cadre formel très riche et très puissant. Il permet ainsi de vérifier que des transformations faites à la main sont correctes. Les transformations autorisées sont nombreuses : toutes les

transformations qui concernent l'espace d'itérations sont autorisées, de plus on peut renommer des variables (à condition que les entrées et les sorties soient clairement identifiées). Cependant, l'un des points faibles actuels de cette méthode est qu'elle ne prend pas en compte la sémantique des opérateurs. La méthode de preuve que nous proposons ne pose pas ce problème puisqu'elle s'appuie entièrement sur la sémantique de notre modèle.

3.3.2 Model Checking Géométrique : Techniques de vérification automatique pour les transformations de boucle et la réutilisation de données

Dans [93], Shashidhar *et al.* veulent prouver l'équivalence de deux systèmes dont l'un est obtenu à partir de l'autre par une transformation[4]. Les deux systèmes sont spécifiés dans ce que les auteurs appellent le modèle géométrique, une généralisation du modèle polyédrique. Les domaines des variables s'expriment à l'aide de l'arithmétique de Presburger. En revanche, les transformations appliquées au système sont limitées. Les transformations ne peuvent qu'introduire des registres ou réorganiser la structure des boucles, sans modifier les fonctions appliquées aux éléments : seuls la modification des expressions indexées et le remplacement des variables par des registres sont autorisés.

Grâce aux restrictions faites sur les transformations, la technique de vérification mise en œuvre est relativement simple. L'idée est la suivante : soient deux équations issues l'une du programme initial et l'autre du programme transformé, qui définissent la même variable et sont définies par la même fonction. Les opérandes différents entre les deux équations correspondent à l'introduction de registres dans l'équation transformée. Pour chacun de ces registres, on construit alors à partir du système transformé une chaîne de registres dont le dernier élément correspond à la variable présente à la place du registre dans le programme initial. Cette chaîne permet de construire une suite de dépendances que l'on compose. Si la composition correspond à la dépendance de la variable du programme initial alors on a prouvé l'équivalence des deux équations. Il faut ensuite vérifier que l'on a autant de points dans le système transformé que dans le système initial. Pour cela, on vérifie que les variables sont définies sur les mêmes domaines.

Cette technique permet de diagnostiquer des erreurs. Les techniques mises en œuvre ici sont plus simples que dans le travail précédent. Les résultats et les limitations des deux techniques semblent en pratique assez similaires, cependant les travaux de Barthou *et al.* seront plus facile à étendre.

3.4 Propriétés fonctionnelles

Nous allons ici nous intéresser à une classe plus large de propriétés que les propriétés d'équivalence : les propriétés fonctionnelles. On appelle propriétés fonctionnelles des prédicats qui relient les entrées aux sorties. On peut les voir comme des propriétés ne dépendant pas de l'ordre d'évaluation des instances (restriction d'une variable à un point de son domaine).Nous allons présenter ici deux travaux. L'un spécifiant les propriétés et les programmes dans un langage d'assertions, l'autre alliant le langage ALPHA et le prouveur de théorème PVS.

3.4.1 Utilisation d'un langage d'assertions

Le langage ALPHA ne permet pas de spécifier toutes sortes de propriétés. Ainsi, dans sa thèse de doctorat [17], Cachera propose de définir une sémantique par assertions proche de la logique de Hoare. Pour cela, il définit un langage d'assertions. Ce langage se fonde sur une logique de premier ordre. Il est constitué :

[4]Il existe de nombreux travaux sur la vérification de transformations de boucles ne se situant pas dans le modèle polyédrique [73, 79, 88]. Nous nous contenterons ici de citer ceux de GoldBerg *et al.*[39] et ceux de Zuch *et al.*[103] qui ont développé tout d'abord des règles de preuve utilisées lors de transformations préservant la structure des boucles et des règles de permutations dans les autres cas.

– des variables logiques,
– des expressions arithmétiques,
– des expressions booléennes,
– des expressions sur les indices des domaines,
– un prédicat $\delta(X, \bar{\imath}, \alpha)$ qui vaut vrai si l'instance au point $\bar{\imath}$ de la variable polyédrique X est définie et a la valeur de la variable logique α.

Une spécification d'un programme S est un triplet $\{P\}S\{Q\}$ où P et Q sont des formules du langage d'assertions correspondant respectivement à des préconditions et postconditions. La précondition P ne fait intervenir que des variables d'entrées. Cachera introduit alors la notion de validité d'une spécification puis la notion de prouvabilité. La prouvabilité est donnée par l'existence d'un invariant, c'est-à-dire l'existence d'un prédicat tel que s'il est vrai dans un état initial, il reste vrai le long de toute suite de transitions partant de l'état initial.

Une fois ces définitions posées, il est possible de proposer une méthodologie de preuve. L'idée est d'utiliser la notion d'invariants. Il introduit deux classes d'invariants :
– les invariants canoniques qui sont une traduction du comportement opérationnel du programme dans le langage d'assertion,
– les invariants univoques qui permettent de décrire les sorties d'un programme en fonction des entrées ; c'est une généralisation de la notion d'invariant canonique. Les invariants univoques permettent de décrire les valeurs des sorties et correspondent donc à la notion de propriété fonctionnelle.

L'introduction de ces deux classes permet de prouver des propriétés partielles en trouvant des invariants qui peuvent être déduits des invariants canoniques. On peut aussi prouver des propriétés d'équivalence, en prouvant que les deux invariants canoniques du programme impliquent le même invariant univoque ou que l'un des invariants canoniques implique l'autre invariant canonique.

L'approche proposée « oublie » la structure syntaxique des programmes ALPHA, dans le sens où la sémantique opérationnelle sur laquelle elle se fonde, considère séparément chaque instance de variable. Elle ne peut donc tirer parti de la régularité intrinsèque du modèle polyédrique.

3.4.2 L'utilisation d'un prouveur de théorèmes

Le travail de Cachera *et al.* [22], allie le langage ALPHA et un prouveur de théorème PVS [92] pour prouver des propriétés fonctionnelles. Deux méthodes sont employées selon les propriétés à prouver.
– Les propriétés simples sont exprimées dans le modèle polyédrique à l'aide d'expressions booléennes. Il faut vérifier si l'expression est trivialement vraie sur tout son domaine. La preuve est séparée en deux et effectuée par différents outils. La première partie consiste à construire des tautologies à l'aide de réécritures syntaxiques ; ces tautologies sont alors transmises à PVS qui les vérifie. Les réécritures syntaxiques utilisées sont différentes selon que la propriété à vérifier est définie récursivement ou non.
  – Si les variables ne sont pas définies récursivement, il suffit de faire un certain nombre de substitutions, de normaliser l'expression et de la transmettre à PVS.
  – Si les variables sont définies récursivement, il faut alors introduire une preuve par induction. La première étape est une étape de réécriture qui fait « disparaître » la récursion. Les expressions réécrites sont alors transmises à PVS. La correction de la preuve est obtenue grâce à un méta-argument qui s'appuie sur la sémantique de ALPHA et des hypothèses d'ordonnancement.
– L'utilisation du langage ALPHA pour exprimer les propriétés est très restrictif : de nombreuses propriétés ne sont donc pas exprimables. Dans le cas de propriétés trop complexes à exprimer dans ce modèle, les propriétés sont spécifiées directement à l'aide de PVS. Les auteurs ont donc développé une interface pour traduire les programmes polyédriques et les ensembles de propriétés dans des théories PVS et ils ont développé des stratégies pour que PVS effectue les preuves spécifiques.

Un travail similaire a été réalisé par Cachera et Pichardie dans [21], mais cette fois ci le langage ALPHA est interfacé avec le prouveur de théorème Coq [96]. Ce travail a nécessité de développer des schémas de récurrence adaptés à la structure récursive des variables polyédriques, ce qui a permis d'automatiser la preuve.

4 Objectifs

Le but de ce travail est comme nous l'avons dit en introduction de prouver des propriétés de sûreté sur des systèmes paramétrés réguliers dans le modèle polyédrique et plus précisément prouver des propriétés sur des signaux de contrôle. Les signaux de contrôle sont des signaux qui ne sont pas directement utilisés pour calculer le résultat fonctionnel du système. Ils sont soit introduits manuellement soit générés automatiquement vers la fin de la synthèse. Nous nous plaçons dans le cadre de la synthèse de haut niveau, qui vise à produire une implémentation par raffinements successifs à partir d'une description faite à un haut niveau d'abstraction. D'une part, il est rarement prouvé que les étapes de raffinement préservent la sémantique des systèmes produits, d'autre part, de nombreuses optimisations peuvent être introduites manuellement par le concepteur, en particulier lors de la phase de génération des signaux de contrôle de l'architecture. Il apparaît alors nécessaire de vérifier que les systèmes obtenus restent corrects vis-à-vis de leur spécification initiale.

Ce problème de vérification formelle n'est pas décidable dans le cadre de systèmes paramétrés [4]. Les techniques que nous proposons ici sont donc des heuristiques, dans le sens où elles pourront fournir des preuves pour un sous-ensemble de systèmes et de propriétés mais pas pour le cas général.

Dans ce chapitre, nous avons vu que pour prouver des propriétés de sûreté la vérification de modèle était adaptée dans le cas de systèmes finis (*cf.* section 1.1). Cependant comme nos systèmes sont paramétrés et potentiellement infinis, la vérification de modèle devient difficile à effectuer(*cf.* section 1.4). On peut alors :

- soit fournir des algorithmes de vérification pour des classes restreintes d'architectures [33, 32, 38],
- soit fournir des semis-algorithmes. Il faut alors utiliser des techniques d'abstraction, générer des invariants, utiliser des techniques d'agrandissement. L'utilisateur est souvent amené à ajuster des paramètres ou à fournir des invariants.[55, 56, 44, 24, 64, 65].

L'autre approche possible dans le cadre de la conception de systèmes est l'utilisation de prouveur de théorèmes 1.2. L'idée est de travailler par équivalence pour vérifier qu'à chaque étape de raffinement le système est équivalent au système raffiné. De nombreux travaux ont été effectués sur ce sujet. Cependant les résultats sont limités : soit il faut que les systèmes aient une architecture très particulière afin de pouvoir générer automatiquement un schéma de preuve [76, 75, 22, 21], soit les transformations de raffinement entre les deux systèmes sont très limitées [8, 93].

L'approche que nous proposons ici est différente. Nous prouvons des propriétés en utilisant les caractéristiques du modèle polyédrique. Nous avons ainsi développé une « logique polyédrique » qui nous permet de spécifier des propriétés dans le modèle polyédrique. Cette logique s'appuie sur la sémantique du modèle polyédrique, nous avons ainsi développé des règles de preuve intrinsèquement liées au modèle polyédrique. Les règles les plus simples correspondent à la sémantique du modèle polyédrique, les autres correspondent à des règles de réécriture et de manipulation de domaines polyédriques. Puis, nous avons développé un algorithme de preuve qui nous permet de prouver des propriétés simples. Dans un second temps, nous avons décidé d'enrichir notre « logique » afin de pouvoir spécifier puis prouver des propriétés plus complexes. Ce travail ne permettra pas de fournir un algorithme de preuve aussi systématique que dans le cas de propriétés simples. Nous nous orientons ici vers la création de tactiques de preuve comme il peut exister dans les prouveurs de théorèmes. Cette deuxième partie est en cours de formalisation, nous nous contenterons ici dans présenter les grandes lignes.

Dans le chapitre suivant, nous présenterons le langage ALPHA qui se fonde sur le modèle poly-

édrique et nous présenterons sa sémantique. Nous nous intéresserons aussi à l'environnement MMALPHA qui permet de manipuler des systèmes ALPHA. C'est dans cet environnement que nous avons implémenté notre outil de preuve.

La seconde partie de ce document nous permettra de présenter dans le chapitre 4 notre système de preuve puis nous nous intéresserons dans le chapitre 5 aux problèmes d'itérations des différentes règles de preuve. Le chapitre 6 sera consacré à l'automatisation du processus de preuve et nous fournirons un algorithme. Puis nous présenterons dans le chapitre 7 des techniques d'accélération pour construire les preuves. Nous étudierons alors deux applications dans le chapitre 8. Le chapitre 9 nous permettra de présenter en enrichissement de notre système de preuve. Le dernier chapitre sera consacré à la conclusion.

Chapitre 2

Le langage ALPHA et l'environnement MMALPHA

Dans ce chapitre, nous allons présenter le langage ALPHA qui va nous servir à spécifier nos programmes et les propriétés que les programmes doivent vérifier. Tout d'abord, nous présenterons dans la section 1 ce langage, sa syntaxe et sa sémantique. Puis, nous présenterons dans la section 2 l'environnement MMALPHA, conçu pour manipuler des programmes ALPHA, et la bibliothèque polyédrique que cet environnement utilise.

1 Le langage ALPHA

Le langage ALPHA a été conçu par Mauras [69]. Ce langage, qui se fonde sur le formalisme des systèmes d'équations récurrentes, est un langage fonctionnel à parallélisme de données. Initialement conçu pour la synthèse des réseaux systoliques, il est utilisé pour la dérivation automatique de circuits réguliers et pour la génération automatique de code parallèle.

Nous ne présenterons ici qu'un sous-ensemble du langage ALPHA pertinent pour ce travail. Nous ne présenterons pas l'opérateur de réduction [63] qui permet d'exprimer les opérations de type somme (Σ) sur des indices bornés par des paramètres et nous ne parlerons pas de la structuration des systèmes [29]. Le lecteur intéressé trouvera une présentation détaillée du langage dans [99].

1.1 Fondements du langage, syntaxe

Contrairement à la plupart des langages fonctionnels, la structure de données de base dans ALPHA n'est pas la liste mais le tableau de forme polyédrique. Un tel tableau est une application d'un polyèdre entier vers un espace de valeurs scalaires (entiers, réels, booléens). Une variable ALPHA correspond donc à la notion de variable du modèle polyédrique (*cf.* chapitre 1). Nous emploierons de manière équivalente les termes de variable ALPHA et de variable polyédrique.

```
system <nom de système>:<déclaration des paramètres>
   (<déclaration des variables d'entrées>)
   returns(<déclaration des variables de sorties>);
var <déclaration des variables locales>;
let
<équations>
tel;
```

FIG. 2.1 – Syntaxe d'un programme ALPHA

Systèmes. La syntaxe générale d'un programme ALPHA est donnée dans la figure 2.1. Un programme ou système est une application des variables d'entrée vers les variables de sortie, définie par des équations récurrentes affines reliant les entrées, les sorties et les variables locales. Le langage ALPHA est un langage équationnel, il obéit donc à la règle de l'assignation unique : il y a au plus une équation pour définir une instance d'une variable.

Exemple 2.1 *Voici un exemple de programme* ALPHA *qui décrit dans la figure 2.1 l'addition des colonnes d'une matrice. Nous supposons que la matrice a 4 colonnes. Le circuit associé à cette description est présenté dans le chapitre 1.*

```
system addcol (a:{t,i|t>=i; 1<=i<=4} of integer)
   returns(res:{t|t>=4} of integer);
var b:{t,i|t>=i; 0<=i<=4} of integer;
let
b = case
   {|i=0}:0.(t,i->);
   {|i>0}: b.(t,i->t-1,i-1)+a.(t,i->t,i);
  esac;
res= b.(t->t,4);
tel;
```

Variables. ALPHA est un langage typé et chaque variable (de sortie, d'entrée ou locale) est déclarée avec la syntaxe suivante :

```
<nom de variable>:<domaine> of <type>
```

Le type peut être entier, réel ou booléen. Le domaine est un polyèdre. La syntaxe d'un domaine est la suivante :

```
{<liste d'indices>|<listes d'équations et d'inéquations>}
```

Les identificateurs apparaissant dans la liste d'équations et d'inéquations sont ceux définis dans la liste d'indices. Les noms des indices n'ont de portée qu'à l'intérieur de la définition du domaine. Nous pouvons donc réutiliser les mêmes noms d'indices dans des domaines distincts sans que ces indices soient liés entre-eux.

Exemple 2.2 *Dans l'exemple 1, la variable d'entrée* a *est définie sur le domaine à deux dimensions* `{t,i|t>=i; 1<=i<=4}`*, elle est de type entier et sa déclaration est*

```
a:{t,i|t>=i; 1<=i<=4} of integer
```

Expressions. Une expression décrit un tableau polyédrique de valeurs et possède un type et un domaine hérités de ses sous-expressions. Les expressions sont construites de la façon suivante : les expressions atomiques sont soit des variables soit des constantes (définies sur le domaine $\mathbb{Z}^0$). Les opérateurs sont soit des opérateurs point-à-point (reliant les valeurs) soit des opérateurs spatiaux (transformant les domaines).

Opérateurs point-à-point. Ces opérateurs sont la généralisation en ALPHA des opérateurs scalaires classiques. Décrivons par exemple l'opérateur de conjonction : si e_1 et e_2 sont deux expressions de même dimension n (les domaines sont de même dimension), alors $e_1\ and\ e_2$ est une expression dont :

- la dimension est n,
- le domaine est l'intersection des domaines de e_1 et de e_2,
- la valeur en un point est la conjonction des valeurs de e_1 et de e_2 en ce même point.

Opérateurs spatiaux. Nous n'en utiliserons que trois : l'opérateur `case`, l'opérateur de restriction et l'opérateur de dépendance.

- **Opérateur** `case`. Cet opérateur permet de définir une expression par plusieurs sous-expressions définies sur des domaines disjoints. La syntaxe est la suivante :
  `case <expression>;...;<expression>; esac`
  Le domaine d'une telle expression est l'union des domaines de toutes les sous-expressions, et sa valeur en un point est la seule valeur définie en ce point par une de ces sous-expressions. La variable b dans l'exemple 1 est définie par une expression case.
- **Opérateur de dépendance.** Cet opérateur établit une application des indices d'un domaine vers les indices d'un autre domaine. Sa syntaxe est la suivante :
  `<expression>.<fonction affine>`
  Si nous appelons f la fonction affine, l'expression $e.f$ est une expression dont le domaine est la préimage par f du domaine de la sous-expression e, et dont la valeur au point z est la valeur de la sous-expression e au point $f(z)$. La syntaxe des fonctions affines est la suivante :
  `(<liste d'indices> -> <liste d'expressions affines>)`
  Les noms des indices n'ont de portée qu'à l'intérieur de la dépendance.

  Exemple 2.3 *L'expression* `b.(t,i->t-1,i-1)` *est donc définie sur la préimage du domaine de la variable b par la dépendance* $(t, i \rightarrow t-1, i-1)$*, c'est-à-dire sur le domaine* `{t,i|t>=i; 1<=i<=5}`*. Soit* (t, i) *un point de ce domaine, alors en ce point l'expression* `b.(t,i->t-1,i-1)` *a la valeur de l'instance de b au point* $(t-1, i-1)$*. Remarquons que si à la place de la dépendance* `(t,i->t-1,i-1)`*, nous avions la dépendance* `(l,m->l-1,m-1)`*, l'expression* `b.(l,m->l-1,m-1)` *serait définie sur le même domaine que* `b.(t,i->t-1,i-1)`*. En effet, les noms des indices de la dépendance n'ont aucun lien avec le nom des indices du domaine.*

- **Opérateur de restriction.** Cet opérateur restreint le domaine d'une expression à un sous-domaine donné. Sa syntaxe est la suivante :
  `<domaine>:<expression>`

  Exemple 2.4 *Soit l'expression* `{t,i|i>0}:b.(t,i->t-1,i-1)+a.(t,i->t,i)`*. Pour calculer son domaine de définition nous devons calculer les domaines de l'expression* `b.(t,i->t-1,i-1)+a.(t,i->t,i)` *et l'intersecter avec le domaine* `{t,i|i>0}`*.*
  - *Calcul du domaine de* `b.(t,i->t-1,i-1)+a.(t,i->t,i)` *: c'est l'intersection des domaines de* `b.(t,i->t-1,i-1)` *et de* `a.(t,i->t,i)`*.*
    - *domaine de l'expression* `a.(t,i->t,i)` *:* `{t,i|t>=i;1<=i<=4}`*,*
    - *domaine de* `b.(t,i->t-1,i-1)` *:* `{t,i|t>=i;1<=i<=5}`*,*
    - *intersection des deux domaines :* `{t,i|t>=i;1<=i<=4}`*.*
  - *Calcul de l'intersection des deux domaines* `{t,i|t>=i;1<=i<=4}` *et* `{t,i|i>0}` *:* `{t,i|t>=i;1<=i<=4}`*.*

  L'expression `{t,i|i>0}: b.(t,i->t-1,i-1)+a.(t,i->t,i)` *est donc définie sur le domaine* `{t,i|t>=i;1<=i<=4}`*, et pour tout point* (i, j) *de ce domaine elle vaut la somme de la valeur de l'instance de b au point* $(i-1, j-1)$ *et de la valeur de l'instance de a au point* (i, j)*.*

Paramètres. Les paramètres [89, 29] sont déclarés dans l'entête du système à l'aide d'un domaine. Les indices de ce domaine correspondent aux paramètres. Les paramètres peuvent apparaître à tout endroit où des indices sont manipulés. Ils étendent tous les domaines et toutes les dépendances.

Exemple 2.5 *Nous paramétrons maintenant l'exemple d'addition des colonnes d'une matrice par l qui représente le nombre de colonnes (cf. Fig. 2.5). Le circuit associé à cette description est présenté dans le chapitre 1.*

```
system addcol {L|L>=1}
              (a:{t,i|t>=i; 1<=i<=L} of integer)
   returns(res:{t|t>=L} of integer);
var b:{t,i|t>=i; 0<=i<=L} of integer;
let
b = case
   {t,i|i=0}:0.(t,i-> );
   {t,i|i>0}: b.(t,i->t-1,i-1)+a.(t,i->t,i);
  esac;
res= b.(t->t,L);
tel;
```

Le domaine de définition de a `{t,i|t>=i; 1<=i<=L}` *est en fait un raccourci pour le domaine* `{t,i,L|t>=i; 1<=i<=L}`. *De même la dépendance* `(t,i->t-1,i-1)` *est un raccourci pour* `(t,i,L->t-1,i-1,L)`.

Il existe une autre notation pour les dépendances appelée *notation tableau* dans cette notation, la dépendance `(t,i->t-1,i-1)` s'écrit `[t-1,i-1]`. Ainsi le système présenté dans l'exemple précédent s'écrirait de la manière suivante :

```
system addcol {L|L>=1}
              (a:{t,i|t>=i; 1<=i<=L} of integer)
   returns(res:{t|t>=L} of integer);
var b:{t,i|t>=i; 0<=i<=L} of integer;
let
b[t,i] = case
   {|i=0}:0[];
   {|i>0}: b[t-1,i-1]+a[t,i];
  esac;
res[t]= b[t,L];
tel;
```

Dans cette notation, le nom des indices des dépendances a une portée sur toute l'équation. Ainsi, les indices t, i qui apparaissent dans l'équation définissant `b` ont une portée dans toute l'équation, c'est-à-dire même dans les domaines des restrictions. C'est pour cela que les indices n'apparaissent plus dans ces domaines.

Dans la suite de ce travail, nous n'utiliserons pas la notation tableau pour des dépendances. Nous conserverons cependant cette notation pour exprimer une instance. Ainsi `b[t,i]` représentera seulement l'instance de `b` au point (t, i).

1.2 Sémantique

Nous présentons maintenant la sémantique dénotationnelle du langage ALPHA [69]. Comme nous ne travaillerons par la suite que sur des signaux de contrôle, nous nous restreignons aux valeurs booléennes et nous ne présenterons la sémantique que dans le cas des signaux booléens.

Définition 2.6 (Syntaxe normalisée) *On dit qu'une expression* ALPHA *est sous forme normale, si elle respecte la grammaire définie dans la figure 2.2. En d'autres termes, dans une expression sous forme*

normale, il n'y a pas de case imbriqués, les case ne sont jamais sous portée d'un opérateur point-à-point, les dépendances sont sous la portée d'une variable et il n'y a pas de domaine sous la portée d'un opérateur point-à-point.

```
expcase::=  case <expression>;...;<expression>;esac |
            expression
expression::= domaine:exp | exp
exp::= exp or exp | exp and exp | not exp |
       terminal.(dep)
terminal::= id | const
dep::= indices-><expind>,...,<expind>
expind::= expind + termind | expind - termind |
          termind | - termind
termind::= entier| indice | entier indice
```

FIG. 2.2 – Syntaxe normalisée d'une expression

Toute expression peut être écrite sous forme normale [69]. Nous ne travaillerons que sur des programmes normalisés.

Définition 2.7 (Domaines syntaxiques) *Les domaines syntaxiques de base sont les suivants :*
- *ID : domaine des identificateurs, on notera id un opérateur de ID.*
- *BOOL : domaine des booléens BOOL $= \{True, False\}$, on notera b un opérateur de BOOL.*
- *UOP : domaine des opérateurs unaires, UOP $= \{not\}$.*
- *BOP : domaine des opérateurs binaires, BOP $= \{and, or\}$.*
- *DOM : domaine des domaines polyédriques, on notera dm un opérateur de DOM.*
- *DEP : domaine des fonctions de dépendance, on notera d un opérateur de DEP.*

Le domaine EXP des expressions se définit par la syntaxe abstraite suivante ($e \in EXP$) :

$$e ::= id \mid b \mid not\, e \mid e_1\, and\, e_2 \mid e_1\, or\, e_2 \mid dom\ : e \mid \text{case } e_1 \ldots e_n \text{ esac} \mid id(dep)$$

Le domaine EQ des équations est défini par

$$eq ::= id = e$$

Définition 2.8 (Domaines sémantiques) *On définit le treillis booléen $\mathbb{B}$ par l'ensemble $\{tt, ff, \bot, \top\}$. On associe à cet ensemble l'opérateur Supval, borne supérieure par rapport à l'ordre usuel « est moins bien défini que » ainsi que les opérateurs $\vee$ et $\wedge$. Ces deux opérateurs sont commutatifs. Pour deux arguments, ils sont définis de la manière suivante ($v \in \{tt, ff\}$) :*

$$\begin{array}{rrcl} \textit{Supval}: & v, v & \mapsto & v \\ & \bot, v & \mapsto & v \\ & \top, v & \mapsto & \top \\ & tt, ff & \mapsto & \top \end{array} \qquad \begin{array}{rrcl} \wedge: & tt, tt & \mapsto & tt \\ & ff, ff & \mapsto & ff \\ & tt, ff & \mapsto & ff \\ & \bot, v & \mapsto & \bot \\ & \top, v & \mapsto & \top \end{array} \qquad \begin{array}{rrcl} \vee: & tt, tt & \mapsto & tt \\ & ff, ff & \mapsto & ff \\ & tt, ff & \mapsto & tt \\ & \bot, v & \mapsto & \bot \\ & \top, v & \mapsto & \top \end{array}$$

Ils s'étendent facilement à n arguments.

On associe de plus à $\mathbb{B}$ un opérateur unaire $\neg$ qui renvoie la négation d'une valeur.

$$\begin{array}{rrcl} \neg: & tt & \mapsto & ff \\ & ff & \mapsto & tt \\ & \bot & \mapsto & \bot \\ & \top & \mapsto & \top \end{array}$$

Le treillis DOM des domaines polyédriques est muni des opérations $\cup$ (borne supérieure) et $\cap$ (borne inférieure). On peut maintenant construire les domaines sémantiques des expressions.

- *Une* valeur multidimensionnelle *est une application d'un point de l'espace dans les valeurs scalaires*

$$VAL = \mathbb{Z}^n \to \mathbb{B}$$

- *Une* valeur spatiale $sp = (dom, val)$ *est une application définie par l'application* val *sur le domaine* dom *et par* $\lambda z.\bot$ *sur* $\mathbb{Z}^n \setminus dom$. *On note* dom *et* val *les fonctions pour accéder aux champs.*

$$sp \in SP = DOM \times VAL$$

- *Un* environnement ρ *est une application des identificateurs des variables vers les valeurs spatiales*

$$\rho \in EV = ID \to SP$$

- *La sémantique* $\mathcal{E}(e)$ *d'une expression* e *est une application des environnements vers les valeurs spatiales*

$$\mathcal{E}(e) \in EXP = EV \to SP$$

Notation 2.9 *Nous noterons v une valeur booléenne appartenant à $\{tt, ff\}$, et $\bar{v}$ l'opposée de cette valeur. Les valeurs v et $\neg v$ sont toutes deux différentes de $\bot$ et $\top$.*

Définition 2.10 (Sémantique des expressions) *On définit la fonction sémantique $\mathcal{E}$ des expressions qui associe à chaque construction syntaxique, sa sémantique dans EXP.*

- *Identificateurs :*

$$\mathcal{E}(id) = \lambda\rho.\rho(id)$$

- *Constantes*

$$\mathcal{E}(k) = \lambda\rho.(\mathbb{Z}^0, \lambda z.k)$$

- *Opérateur unaire :*

$$\mathcal{E}(\textit{not}\, e)(\rho) = (\quad dom(\mathcal{E}(e)(\rho)),\ \neg(val(\mathcal{E}(e)(\rho))))$$

- *Opérateurs binaires :*

$$\begin{aligned}\mathcal{E}(e_1 \textit{ and } e_2)(\rho) = (\quad & dom(\mathcal{E}(e_1)(\rho)) \cap dom(\mathcal{E}(e_2)(\rho)),\\ & (val(\mathcal{E}(e_1)(\rho)) \wedge val(\mathcal{E}(e_2)(\rho))))\\ \mathcal{E}(e_1 \textit{ or } e_2)(\rho) = (\quad & dom(\mathcal{E}(e_1)(\rho)) \cap dom(\mathcal{E}(e_2)(\rho)),\\ & (val(\mathcal{E}(e_1)(\rho)) \vee val(\mathcal{E}(e_2)(\rho))))\end{aligned}$$

- *Restriction :*

$$\mathcal{E}(dm : e)(\rho) = (\quad dm \cap dom(\mathcal{E}(e)(\rho)),\ val(\mathcal{E}(e)(\rho)))$$

- *Case :*

$$\mathcal{E}(case\ e_1 \ldots esac)(\rho) = (\quad dom(E(e_1)(\rho)) \cup \ldots,\ Supval(val(E(e_1)(\rho)), \ldots))$$

Le domaine d'une expression case est défini comme étant l'union des domaines des différentes expressions la composant. Sa valeur en un point est la valeur d'une des expressions définissant l'expression case. Si les domaines ne sont pas disjoints deux à deux, alors, pour les points appartenant à ces intersections, la valeur de l'expression case sera $\top$.

– *Dépendance :*

$$\mathcal{E}(e.d)(\rho) = (\quad d^{-1}(dom(\mathcal{E}(e)(\rho))),\\ \qquad\qquad val(\mathcal{E}(e)(\rho) \circ d))$$

où $\circ$ correspond à l'opérateur de composition de fonctions. L'opérateur dépendance compose e et d. La valeur d'une expression dépendance au point z est la valeur de l'expression e au point $d(z)$, donc $d(z)$ appartient au domaine de l'expression e. On en conclut donc que le domaine de $e.d$ est la préimage par d du domaine de e. Remarque : nous notons abusivement d l'objet syntaxique (en partie gauche) et l'objet sémantique (en partie droite).

Définition 2.11 (Sémantique d'une équation) *La sémantique d'une équation est une application de l'ensemble des environnements dans lui-même :*

$$EQ = EV \rightarrow EV$$

Elle est définie de la manière suivante :

$$\mathcal{Q}(X = e)(\rho) = \rho'$$

ou ρ' est défini pour tout identificateur id de la manière suivante :

$$\begin{cases} \textit{si } id = X \quad \textit{alors } \rho'(id) = & (dom(\rho(X)) \cup dom(\mathcal{E}(e)(\rho)),\\ & Supval(val(\rho(X)), val(\mathcal{E}(e)(\rho)))) \\ \textit{si } id \neq X \quad \textit{alors } \rho'(id) = & \rho(id) \end{cases}$$

En d'autres termes, une équation définit une fonction d'enrichissement de l'environnement.

Définition 2.12 (Sémantique d'un système d'équations) *La sémantique d'un système d'équations définit une application de l'ensemble des environnements vers lui-même. Pour un environnement donné, elle se définit comme étant un plus petit point fixe de la fonction suivante :*

$$\lambda r.(\mathcal{Q}(eq_n) \circ \cdots \circ \mathcal{Q}(eq_1))(r)$$

Si μ est l'opérateur de plus petit point fixe, la sémantique du système S pour l'environnement ρ est alors :

$$\mathcal{S}(\text{let } eq_1 \ldots eq_n \text{ tel})(\rho) =\\ (\mu r.(\mathcal{Q}(eq_n) \circ \cdots \circ \mathcal{Q}(eq_1))(r))(\rho)$$

Définition 2.13 (Sémantique d'un programme) *Un programme construit l'environnement initial avec les données en entrée et l'enrichit par le système d'équations. C'est donc une application d'un ensemble de valeurs multidimensionnelles dans un ensemble de valeurs multidimensionnelles*

$$\mathcal{P} \in PROG = VAL^* \rightarrow VAL^*$$

Pour un programme S

$$\begin{array}{ll} system & S \\ & d_1 \ldots d_m \\ returns & d_{m+1} \ldots d_n \\ var & d_{n+1} \ldots d_p \\ & eqs \end{array}$$

où d_i signifie $id_i : dom_i$, l'environnement initial ρ est défini par :

$$\rho_{v_1,\ldots v_m}(id) = \begin{cases} \textit{si } id = id_i,\ 1 \leq i \leq m \quad \textit{alors} \quad \rho_{v_1,\ldots v_m}(id_i) = (dom_i, v_i) \\ \textit{sinon } id = id_i,\ m+1 \leq i \leq p \quad \textit{alors} \quad \rho_{v_1,\ldots v_m}(id_i) = (dom_i, \lambda z.\bot) \end{cases}$$

La sémantique du programme ci-dessus est donc :

$$\mathcal{P}(system\ d_1 \ldots d_m\ returns\ d_{m+1} \ldots d_n\ var\ d_{n+1} \ldots d_p\ eqs) =\\ \lambda v_1 \ldots v_m.(\lambda\rho.val(\rho(id_{m+1})) \ldots val(\rho(id_p))(\mathcal{S}(eqs)(\rho_{v_1,\ldots v_m})))$$

Définition 2.14 (environnement bien défini) *Nous appelons* environnement bien défini *pour un système S, un environnement ρ du système S tel que pour toute variable X du système S définie sur le domaine $\mathcal{D}_X$, pour tout point z de $\mathcal{D}_X$, $\rho(X)(z)$ est différent de la valeur $\perp$ et de la valeur $\top$.*

Définition 2.15 (Environnement sémantiquement correct) *Nous appelons environnement* sémantiquement correct *pour un système S, un environnement bien défini pour S qui est un plus petit point fixe de la sémantique du système S. Nous notons*

$$\rho \propto S$$

pour signifier que ρ est sémantiquement correct pour S.

1.3 Équivalences sémantiques

Nous donnons ici une définition et une notation concernant l'équivalence sémantique des systèmes.

Définition 2.16 (Équivalence de systèmes) *Soient S_1 et S_2 deux systèmes, nous dirons que S_1 et S_2 sont sémantiquement équivalents (noté $S_1 \cong S_2$) si et seulement si :*

- *S_1 et S_2 ont des mêmes variables d'entrée et de de sortie de même noms et définies sur les mêmes domaines,*
- *Pour toute variable X appartenant à S_1 et à S_2, pour tous les environnements ρ_1 et ρ_2 tels que $\rho_1 \propto S_1$ et $\rho_2 \propto S_2$ et tels que pour toute variable d'entrée I*

  $$\rho_1(I) = \rho_2(I)$$

  alors

  $$\rho_1(X) = \rho_2(X)$$

  Nous dirons que ρ_1 et ρ_2 sont équivalents, et nous le noterons $\rho_1 \cong \rho_2$.

Définition 2.17 (Équivalence de deux environnements) *Soient S_1 et S_2 deux systèmes équivalents, soient ρ_1 un environnement sémantiquement correct de S_1 et ρ_2 un environnement sémantiquement correct de S_2. Nous dirons que ρ_1 et ρ_2 sont sémantiquement équivalents (ce qui sera noté $\rho_1 \cong \rho_2$) si et seulement si : Pour toute variable X appartenant à S_1 et à S_2,*

$$\rho_1(X) = \rho_2(X)$$

Définition 2.18 (sursystème) *Soient S_1 et S_2 deux systèmes, on dira que S_2 est un sursystème de S_1 si et seulement si toutes les variables de S_1 sont des variables de S_2, et si toutes les définitions des variables de S_1 sont identiques à celles de S_2.*

Si S_2 est un sursystème de S_1, alors S_1et S_2 sont sémantiquement équivalents.

Nous allons maintenant définir la notion d'équivalence sémantique pour des expressions.

Définition 2.19 (Équivalence sémantique de deux expressions) *Soit ρ un environnement et soient e et f deux expressions, alors e et f sont* sémantiquement équivalentes *pour ρ sur le domaine $\mathcal{D}$ si :*

$$\forall z \in \mathcal{D},\ val(\mathcal{E}(e)(\rho)[z]) = val(\mathcal{E}(f)(\rho)[z])$$

On le note $e \cong_{\mathcal{D}} f$.

Pour tout domaine $\mathcal{D}$, la relation $\cong_{\mathcal{D}}$ est évidemment une relation d'équivalence .

Définition 2.20 (Équivalence sémantique de variables issues de deux systèmes différents) *Soient S_1 et S_2 deux systèmes équivalents, soient X_1 une variable de S_1 et X_2 une variable de S_2, on dira que X_1 est sémantiquement équivalente à X_2 (noté $X_1 \cong X_2$) si et seulement si pour tout couple d'environnement (ρ_1, ρ_2) tel que $\rho_1 \cong \rho_2$ on a*

$$\rho_1(X_1) = \rho_2(X_2)$$

Nous allons maintenant nous intéresser à l'environnement MMALPHA.

2 L'environnement MMALPHA

En parallèle du langage ALPHA a été développé un environnement de programmation MMALPHA. Cet environnement fournit principalement une boîte à outils pour compiler des architectures parallèles sur silicium à partir de spécifications fonctionnelles haut niveau : la version actuelle de MMALPHA permet d'obtenir en quelques heures un code VHDL (Very High Speed Integrated Circuit Hardware Description Language) synthétisable qui peut être appliqué sur des architectures parallèles linéaires. MMALPHA est un ensemble de modules écrits en Mathematica.

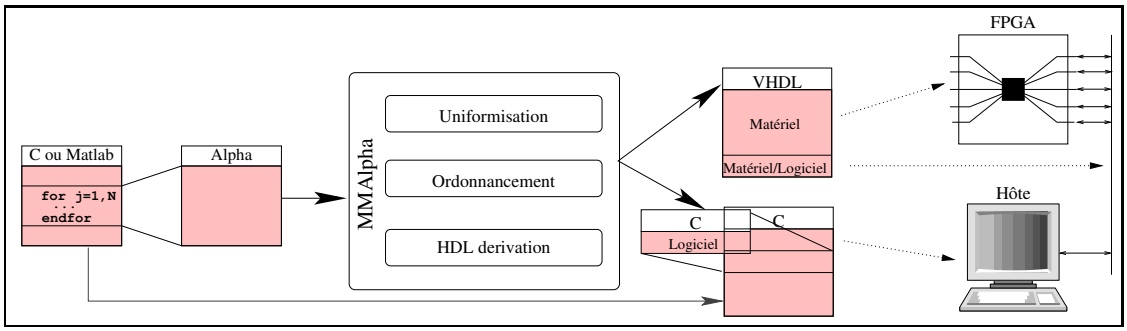


FIG. 2.3 – Flot de conception en MMALPHA. Les boîtes carrées représentent les programmes dans différents langages, et les boîtes arrondies représentent les transformations.

Flot de conception en MMALPHA Le flot de conception avec MMALPHA est représenté dans la figure 2.3. L'application est tout d'abord décrite en C ou en Matlab, et la partie qui doit être implémentée en utilisant une architecture parallèle est réécrite en ALPHA, soit manuellement, soit en utilisant des outils automatiques. Le code est alors chargé dans le logiciel MMALPHA qui l'analyse et, après des transformations, le traduit en une combinaison de VHDL (pour la partie matérielle) et de code C (pour la partie logicielle). Le code VHDL peut ensuite être synthétisé pour obtenir un ASIC (Application-Specific Integrated Circuit) ou configurer une architecture FPGA (Field-Programmable Gate Array).

Outils MMALPHA

- **Analyse statique :** cette analyse permet de vérifier certaines propriétés (par exemple de détecter du code mort, des instances non définies), d'estimer des quantités (le nombre de calculs effectués dans un nid de boucles en fonction des bornes de ces boucles), ou d'inférer des paramètres liés aux implémentations matérielles (les paramètres correspondant par exemple à la dimension de l'architecture). Toutes ces analyses statiques peuvent être utilisées de manière hiérarchique sur des applications ALPHA.
- **Ordonnancement :** L'ordonnanceur est l'outil central de MMALPHA. L'ordonnancement permet de préciser l'instant et le processeur qui exécutera un calcul donné. Un programme ALPHA peut être analysé et ordonnancé pour une implémentation parallèle. L'ordonnanceur permet de calculer des ordonnancements uni- ou multi-dimensionnels, ce qui permet de fournir de nombreuses solutions pour ajuster le degré de parallélisme.
- **Preuves de propriétés :** la plupart des transformations implémentées en ALPHA sont des réécritures qui préservent la sémantique et qui assurent donc l'équivalence entre la spécification d'origine et le code final. Cependant, dans de nombreux cas, il est nécessaire de prouver formellement certaines propriétés que le processus de raffinement ne permettait pas d'assurer. Les outils développés utilisent l'environnement MMALPHA et le prouveur de théorème PVS. Pour plus de

détails se reporter à la section 3.4.2 du chapitre 1. C'est dans ce cadre que s'inscrit le travail effectué au cours de cette thèse.

- **Génération et simulation de code :** la génération de code C à partir d'un système ALPHA est utilisé aussi bien pour la simulation que pour implémenter certaines parties d'un programme. Les systèmes ALPHA peuvent être transformés de manière efficace en programmes C grâce à l'utilisation de techniques de réutilisation de mémoire et d'analyse de nids de boucles [84, 83]. Il est possible de générer du code pour la simulation à tous les niveaux du flot de conception.
- **Génération matérielle :** c'est le but principal de MMALPHA. L'environnement MMALPHA est une boîte à outils qui permet de générer rapidement et sûrement des descriptions matérielles parallèles à partir de spécification haut-niveau. Cette génération se fait à l'aide des techniques suivantes : uniformisation du code, ordonnancement et dérivation matérielle. L'uniformisation est un outil qui permet de transformer des dépendances affines en dépendances uniformes et ainsi de pipeliner les valeurs. Quant à la dérivation matérielle, elle permet de générer automatiquement des signaux de contrôle, de proposer des interprétations matérielles du système et de structurer automatiquement les programmes. La transformation produit du code VHDL synthétisable.

3 La bibliothèque polyédrique

La bibliothèque polyédrique Polylib [100] développée par Wilde au cours de sa thèse est largement utilisée par MMALPHA. Son développement est actuellement effectué conjointement à Rennes, Strasbourg et Brigham Young University. Cette bibliothèque permet d'effectuer des calculs symboliques sur des polyèdres. Dans la Polylib, les polyèdres sont représentés dans leurs deux formes duales :

- listes de contraintes : égalités et inégalités,
- caractéristiques géométriques : sommets, rayons et lignes.

3.1 Représentation des polyèdres

Représentation par contraintes La représentation par contraintes a été présentée dans le chapitre 1, elle consiste à représenter un polyèdre comme étant l'intersection d'un nombre fini de demi-espaces avec $\mathbb{Z}^n$. Les polyèdres se représentent donc de la manière suivante :

$$\{x \in \mathbb{Z}^n | A.x \leq b\} \tag{2.1}$$

où A est une matrice de $\mathbb{Z}^{m \times n}$ et b un vecteur de $\mathbb{Z}^m$.

Représentation géométrique Pour la représentation géométrique l'idée est de représenter les polyèdres par un ensemble de sommets, de rayons et de lignes. Nous donnerons tout d'abord une définition dans $\mathbb{Q}^n$.

Définition 2.21 *Soit x un vecteur de $\mathbb{Q}^n$, et λ un vecteur de coefficients scalaires de $\mathbb{Q}^n$, nous définissons les combinaisons suivantes.*

- *Une combinaison linéaire :* $\sum \lambda_i x_i$
- *Une combinaison positive :* $\sum \lambda_i x_i, \forall i, \lambda_i \geq 0$
- *Une combinaison convexe :* $\sum \lambda_i x_i, \sum \lambda_i = 1, \forall i, \lambda_i \geq 0$

Dans la suite, $\mathcal{K}$ désignera un sous-ensemble de $\mathbb{Q}^n$.

Définition 2.22 (Sommet) *Tout point de $\mathcal{K}$ qui peut s'exprimer comme une combinaison convexe de points distincts de $\mathcal{K}$ est un* sommet *de $\mathcal{K}$ et inversement.*

Définition 2.23 (Rayon) *Un* rayon *de $\mathcal{K}$ est un vecteur $r \in \mathbb{Q}^n$ tel que si x est un point de $\mathcal{K}$, alors $x + \mu r \in \mathcal{K}$ pour tout $\mu \in \mathbb{Q}^n$ et $\mu \geq 0$*

Un rayon n'est pas un ensemble de points, mais une direction dans laquelle $\mathcal{K}$ est infini.

Définition 2.24 (Rayon extrémal) *Un rayon de $\mathcal{K}$ est* extrémal *si et seulement s'il ne peut pas s'exprimer comme une combinaison positive de deux autres rayons distincts de $\mathcal{K}$.*

Définition 2.25 (Ligne) *Une* ligne *(ou rayon bidirectionnel) de $\mathcal{K}$ est un vecteur $l \in \mathbb{Q}^n$ tel que pour tout $x \in \mathcal{K}$, $x + \mu l \in \mathcal{K}$ pour tout $\mu \in \mathbb{Q}^n$.*

La représentation géométrique d'un polyèdre est aussi appelée caractérisation de Minkowski [91]. Elle est définie de la manière suivante :

$$\{x \mid x = L\lambda + R\mu + V\nu,\ \mu, \nu \geq 0,\ \sum \nu = 1, L\} \tag{2.2}$$

où $\lambda \in \mathbb{Q}^l,\ l \in \mathbb{Z},\ \mu \in \mathbb{Q}^r,\ r \in \mathbb{Z},\ \nu \in \mathbb{Q}^v,\ v \in \mathbb{Z}$, où $L \in \mathbb{Q}^{l \times n}$ représente l'ensemble des lignes, $R \in \mathbb{Q}^{r \times n}$ l'ensemble des rayons, et $V \in \mathbb{Q}^{v \times n}$ l'ensemble des sommets. Un polyèdre s'exprime donc en termes de combinaison linéaire de lignes (colonnes de la matrice L), de combinaison convexe de sommets (colonnes de la matrice V) et de combinaison positive de rayons extrémaux (colonnes de la matrice R).

Nous donnons en figure 3.1 un exemple pour illustrer les deux représentations.

Proposition 2.26 *Les caractérisations 2.1 et 2.2 sont équivalentes. De plus, il existe un algorithme pour passer d'une forme à l'autre.*

La démonstration se trouve dans [91, 99]. L'algorithme pour passer d'une représentation à l'autre est en $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$, où n est le nombre de contraintes du polyèdre et d la dimension.

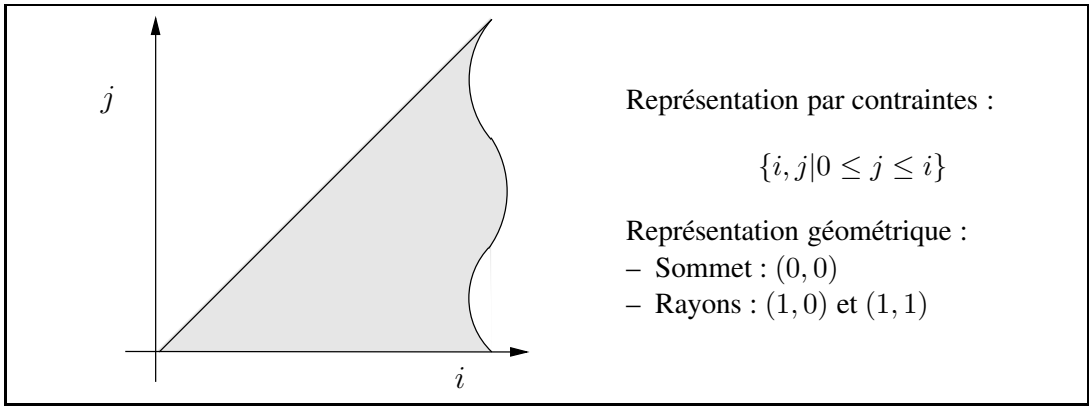


FIG. 2.4 – Représentations d'un polyèdre

3.2 Les différents outils

La bibliothèque polyédrique offre des outils pour effectuer les opérations suivantes :
– intersection de deux polyèdres,
– union de deux polyèdres — le résultat n'est pas un polyèdre,
– différence de deux polyèdres,

– simplification, c'est-à-dire agrandissement[1] d'un polyèdre dans le contexte d'un autre polyèdre
– enveloppe convexe — qui permet de définir un polyèdre à partir d'un ensemble quelconque de points,
– image d'un polyèdre par une transformation affine,
– préimage d'un polyèdre par une transformation affine,
– création d'un polyèdre à partir d'une liste de contraintes,
– création d'un polyèdre à partir d'une liste de sommets, de rayons et de lignes,
– création d'un polyèdre vide,
– création d'un polyèdre universel, c'est-à-dire le polyèdre de dimension n égale à $\mathbb{Z}^n$.

En dehors du calcul de l'enveloppe convexe et de la simplification, nous utiliserons toutes les autres opérations.

[1] Nous définirons un autre opérateur d'agrandissement (*cf.* partie II, chapitre 7)

Deuxième partie

Prouver avec des polyèdres

Chapitre 3

Introduction

Dans cette partie, nous allons définir un cadre formel de preuve. L'idée est de pouvoir spécifier des propriétés et des hypothèses dans le modèle polyédrique, puis de prouver automatiquement ces propriétés. Les premiers chapitres seront donc consacrés à la définition de ce cadre, puis nous nous intéresserons à l'automatisation du processus de preuve et à son application. Enfin, nous verrons comment élargir ce cadre formel. Ce chapitre introductif va nous permettre d'illustrer notre démarche.

Le but de ce chapitre n'est pas de rentrer dans les détails théoriques. Nous supposerons que nous devons prouver une propriété φ sur un système S. Prenons par exemple un système S contenant une variable X définie sur le domaine $\mathcal{D}_X = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{D}_3$ par l'équation suivante :

$$X = \begin{cases} \mathcal{D}_1 : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \\ \mathcal{D}_2 : X.d_2 \\ \mathcal{D}_3 : \mathit{True} \end{cases}$$

et supposons que la propriété φ est la suivante : nous voulons prouver que sur un domaine $\mathcal{D}$ inclus dans le domaine $\mathcal{D}_X$, la variable X vaut vrai. Sur la figure 3.1, nous représentons les domaines $\mathcal{D}$ et $\mathcal{D}_X$ et les sous domaines $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$. Le domaine $\mathcal{D}$ est représenté en gris sur cette figure. Les dépendances d_1 (pour le sous domaine $\mathcal{D}_1$) et d_2 (pour le sous domaine $\mathcal{D}_2$) sont représentées par des vecteurs.

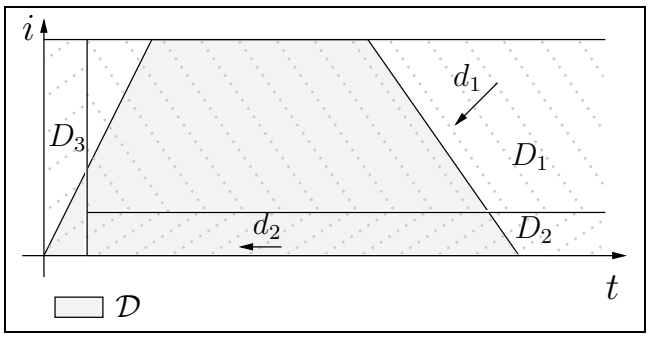


FIG. 3.1 – Domaine de la variable X

La première étape de notre travail sera de définir une syntaxe pour les propriétés. Cette syntaxe sera présentée succinctement dans le chapitre 4 et enrichie dans le chapitre 9. La syntaxe de notre formule φ est la suivante

$$\mathcal{D} : X \downarrow \mathit{tt}$$

Cette formule sera sémantiquement valide si pour tout environnement ρ sémantiquement correct, la valeur de l'expression X vaut vrai en tout point z du domaine $\mathcal{D}$, c'est-à-dire que pour tout point z du domaine $\mathcal{D}$, $val(\mathcal{E}[X](\rho))[z] = v$. Nous le noterons :

$$\models_\rho \mathcal{D} : X \downarrow v$$

Une définition plus formelle sera donnée dans le chapitre 4 et sera enrichie dans le chapitre 9.

Pour prouver que la propriété est valide, il n'est bien sûr pas possible d'énumérer les valeurs de toutes les instances des variables pour calculer la valeur de l'expression e. En effet, le domaine $\mathcal{D}$ est paramétré et peut représenter une infinité de points.

Nous devons donc trouver d'autres techniques pour prouver la validité des formules. Nous allons développer un système de règles de preuve et nous prouverons que ces règles sont correctes. L'existence d'une preuve pour la formule φ sera notée $\vdash \varphi$. L'existence d'une preuve pour la formule φ impliquera alors la validité sémantique de la formule :

$$\vdash \varphi \implies \forall \rho \propto,\ S \models_\rho \varphi \tag{3.1}$$

Notre but est donc de donner une preuve sur l'exemple présenté ci-dessus :

$$\vdash \mathcal{D} : X \downarrow \mathit{tt}$$

Nous allons présenter dans ce chapitre les différentes techniques de preuve servant à définir notre système de règles de preuve. Nous verrons parallèlement comment enchaîner ces règles pour construire une preuve.

Voici les techniques développées :

- **substitution par constante,** (cette technique est une restriction des techniques de substitution utilisées dans [22, 93])
- **recherche de motif,**
- **recherche d'auto-dépendance,**
- **agrandissement de domaine,**

Nous allons illustrer ces différentes techniques sur l'exemple suivant.

La première règle de preuve que nous avons développée est la *substitution par constante*. Ce principe repose sur une réécriture. Nous voulons réécrire l'expression définissant la variable X en une expression sémantiquement équivalente qui ne dépend plus de X. De manière plus formelle, nous voulons trouver une expression e ne contenant plus d'occurrence de X telle que :

$$\forall \rho \propto S,\ \models_\rho \mathcal{D} : X \downarrow \mathit{tt} \iff \models_\rho \mathcal{D} : e \downarrow \mathit{tt}$$

Nous allons ici expliquer comment obtenir cette expression, la preuve de l'équivalence sera donnée dans le chapitre 4, précisons simplement qu'elle repose sur une récurrence sur l'indice temporel. Supposons que pour tout environnement ρ ($\rho \propto S$) $\models_\rho \mathcal{D} : X \downarrow \mathit{tt}$. Ainsi, toutes les instances de X en un point z de $\mathcal{D}$ valent *tt*. Regardons maintenant l'expression $X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2)$ définie sur le domaine $\mathcal{D}_1$. Soit z un point de $\mathcal{D}_1$, deux cas peuvent alors se produire :

- si $d_1(z)$ est un point de $\mathcal{D}$, alors l'instance $X[d_1(z)]$ est égale à *tt*, et l'expression $X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2)$ au point z peut être réécrite en $Y.\delta_1 \vee Y.\delta_2$,
- si $d_1(z)$ n'est pas un point de $\mathcal{D}$, l'expression $X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2)$ reste inchangée.

Pour tous les points $\{z | z \in d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1\}$, nous pouvons alors réécrire l'expression $X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2)$ en $Y.\delta_1 \vee Y.\delta_2$. Nous obtenons ainsi l'expression suivante :

$$\mathcal{D}_1 : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) = \left\{ \begin{array}{l} d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1 : (Y.\delta_1 \vee Y.\delta_2) \\ \mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}) : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \end{array} \right. \tag{3.2}$$

De manière similaire, nous pouvons réécrire l'expression $\mathcal{D}_2 : X.d_2$ dans l'équation suivante :

$$\mathcal{D}_2 : X.d_2 = \left\{ \begin{array}{l} d_2^{-1}(\mathcal{D}) \cap \mathcal{D}_2 : \mathit{True} \\ \mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D}) : X.d_2 \end{array} \right. \tag{3.3}$$

Ainsi, la variable X se réécrit en :

$$X = \begin{cases} \mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}) : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \\ d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1 : Y.\delta_1 \vee Y.\delta_2 \\ \mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D}) : X.d_2 \\ \mathcal{D}_3 \cup (\mathcal{D}_2 \cap d_2^{-1}(\mathcal{D})) : \textit{True} \end{cases}$$

Les différents domaines sont représentés sur la figure (3.2). Il y a équivalence entre l'équation initiale de X et l'équation réécrite. La preuve de cette équivalence sera donnée dans le chapitre 4.

La règle de preuve que nous venons de décrire n'est pas un simple principe de réécriture. Les domaines ont aussi un rôle important. Chaque sous-expression n'est réécrite que sur une partie de son domaine. Il ne faut pas oublier qu'une équation ne définit pas une instance, mais elle définit toutes les instances d'un domaine. Ainsi, l'étape de réécriture permet de réécrire simultanément toutes les instances d'un domaine donné, mais seulement celles-ci. Une instance de X n'appartenant pas au domaine $\mathcal{D}$ ne devra pas être substituée. Pour ne substituer que les instances de $\mathcal{D}$, nous avons dû effectuer des calculs sur les domaines.

En appliquant cette règle de preuve, nous pouvons réécrire la variable X. Cette variable est définie par une expression *case* à quatre branches. Il existera une preuve de la formule $\mathcal{D} : X \downarrow tt$ si et seulement s'il existe une preuve pour les quatre formules définies à partir des expressions des branches de X, c'est-à-dire :

- $(\mathcal{D}_3 \cup (\mathcal{D}_2 \cap d_2^{-1}(\mathcal{D}))) \cap \mathcal{D} : \textit{True} \downarrow tt$
- $(\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D} : X.d_2 \downarrow tt$
- $(d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1) \cap \mathcal{D} : Y.\delta_1 \vee Y.\delta_2 \downarrow tt$
- $(\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \downarrow tt$

Nous avons donc décomposer la preuve en 4 sous-buts, nous appelons cette règle, règle de décomposition. Sa correction repose sur l'implication 3.1 et la définition de la sémantique (nous y reviendrons plus longuement dans le chapitre 4). Les domaines de ces sous-buts sont représentés dans la figure (3.3). Nous allons étudier chacun des sous-buts et continuer l'illustration des différentes techniques.

- **Axiome.** Sur le domaine $(\mathcal{D}_3 \cup (\mathcal{D}_2 \cap d_2^{-1}(\mathcal{D}))) \cap \mathcal{D}$, nous obtenons le nouveau sous-but suivant :

  $$\vdash (\mathcal{D}_3 \cup (\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D}))) \cap \mathcal{D} : \textit{True} \downarrow tt$$

  Ce sous-but de preuve est un axiome, la justification est identique à celle de la technique précédente : utilisation de l'implication 3.1 et de la définition de la sémantique.

  $$\forall \rho \propto S, \forall z \in \mathcal{D},\ val(\mathcal{E}(\textit{True})(\rho))(z) = tt$$

- **Agrandissement de domaine.** Sur le domaine $(\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D}$, nous obtenons le sous-but suivant :

  $$\vdash (\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D} : X.d_2 \downarrow tt \tag{3.4}$$

  Deux cas peuvent alors se produire :
  - soit le domaine $(\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D}$ est vide[1]. Dans ce cas, il n'y a pas de sous-but,
  - ou, comme c'est le cas dans notre exemple, ce domaine n'est pas vide. La solution que nous proposons ici est *d'agrandir le domaine* $\mathcal{D}' = (\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D}$ de validité de ρ pour la variable X en direction de d_2. Pour cela, nous utilisons une heuristique. Le domaine agrandi $\mathcal{D}_{wid}$ est construit à partir du domaine $(\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D}$. Soit z un point de $\mathcal{D}_{wid}$, deux cas se produisent :
    - soit $d_2(z) \in \mathcal{D}_2$, et alors $d_2(z) \in \mathcal{D}_{wid}$,

[1] Le domaine doit être vide au sens entier. Cependant la librairie polyédrique ne permet de tester le vide qu'au sens rationnel. Nous reviendrons dans le chapitre 4 sur ce problème

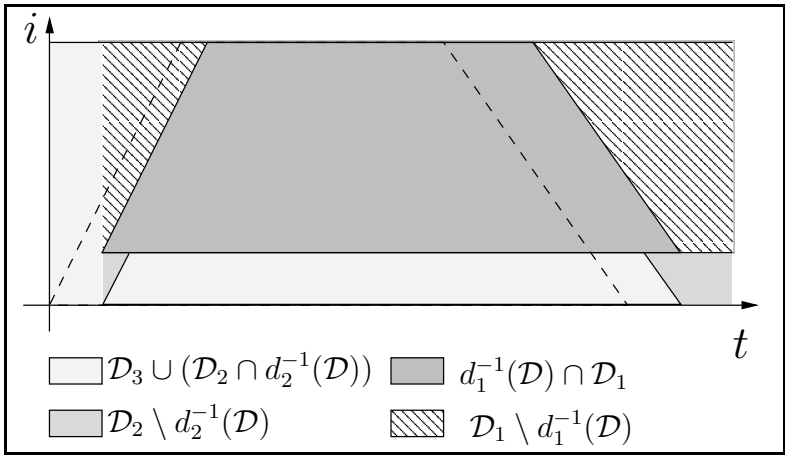


FIG. 3.2 – Domaines de la variable X après substitution

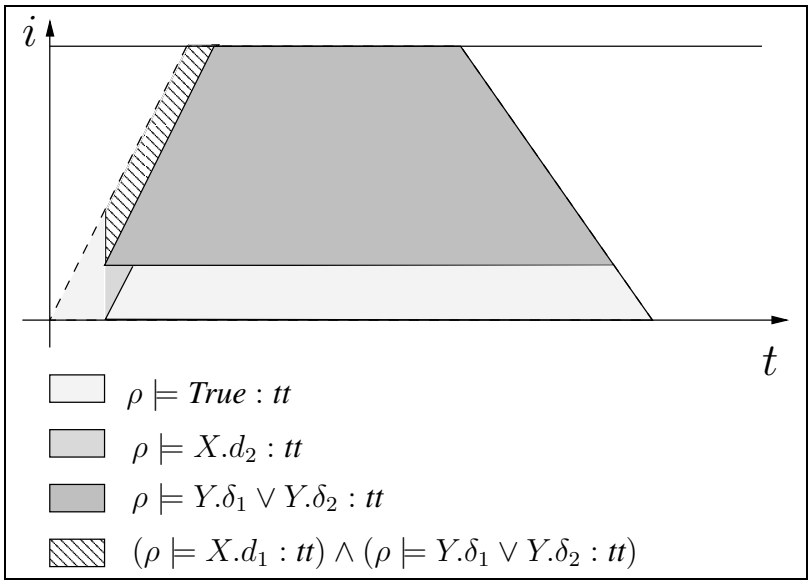


FIG. 3.3 – Sous buts pour la variable X

– soit $d_2(z) \notin \mathcal{D}_2$, et alors $d_2(z) \notin \mathcal{D}_{wid}$.
Dans notre exemple, nous trouverons que

$$\vdash \mathcal{D}_{wid} : X \downarrow tt$$

– **Recherche de motifs et recherche d'auto-dépendance.** Sur le domaine $(d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1) \cap \mathcal{D}$, nous obtenons le sous-but suivant :

$$\vdash (d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1) \cap \mathcal{D} : Y.\delta_1 \vee Y.\delta_2 \downarrow tt \tag{3.5}$$

Dans ce cas, nous ajoutons la variable W définie sur le domaine de l'expression $Y.\delta_1 \vee Y.\delta_2$ au système, et le problème devient :

$$\vdash (d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1) \cap \mathcal{D} : W \downarrow tt$$

L'idée ici sera de réécrire W récursivement. Ce sera effectué par le principe de *détection automatique d'auto-dépendance* décrit dans le chapitre 4. Ce principe est un cas particulier de la *recherche de motif* décrite dans le chapitre 4.

– **Règles de décomposition par rapport à $\vdash$.** Sur le domaine $(\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D}$, nous obtenons le sous-but suivant :

$$\vdash (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \downarrow tt$$

Ce sous-but est équivalent à la conjonction des deux sous-buts suivants :

$$(\vdash (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} : X.d_1 \downarrow tt) \wedge (\vdash (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} : Y.\delta_1 \vee Y.\delta_2 \downarrow tt)$$

La justification s'appuie toujours sur utilisation de l'implication 3.1 et de la définition de la sémantique. Remarquons simplement que dans le cas de la disjonction, la sémantique ne permet pas de générer de nouveaux sous-buts de preuve. Ceci sera illustré dans le chapitre suivant. Nous générons alors deux nouveaux sous-buts :
– l'un concernant $\vdash (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} : X.d_1 \downarrow tt$,
– et l'autre $\vdash (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} : Y.\delta_1 \vee Y.\delta_2 \downarrow tt$.
Ces deux sous-buts sont de même nature que (3.4) et (3.5).

Nous avons décomposé notre but initial en plusieurs sous-buts. Nous expliquerons dans le chapitre 6 comment poursuivre et automatiser ces preuves jusqu'à obtenir des constantes ou des variables d'entrées dans chaque branche. Pour cela nous représenterons les preuves à l'aide d'un arbre, les nœuds fils seront obtenus à l'aide des règles de preuve. Nous prouverons que le processus de construction est fini.

Cet exemple a donc permis d'illustrer certaines des techniques que nous mettons en œuvre pour la vérification. Ces techniques sont variées et s'inspirent de principes classiques (réécritures, recherche d'auto-dépendance, agrandissement de domaine). Leur originalité vient de l'utilisation du modèle polyédrique. Elles ne sont pas appliquées sur une instance mais sur un ensemble d'instances. Elles nécessitent donc de prendre en compte les domaines.

Dans le chapitre 4, nous définirons une « logique polyédrique » : c'est-à-dire que nous définirons une syntaxe pour spécifier nos propriétés, nous en donnerons la sémantique et nous étudierons un système de preuve associé à cette logique. Nous prouverons la correction de ces règles. Dans le cadre de ce système de preuve, nous nous attarderons sur les deux techniques de base : la substitution par constante et la recherche de motif. Le chapitre 5 sera consacré à la recherche d'auto-dépendance, ou comment itérer substitution par constante et recherche de motif. Dans le chapitre 6, nous verrons comment utiliser ces techniques pour construire et automatiser une preuve, ce qui nous conduira à développer dans le chapitre 7 des techniques d'accélération fondées sur l'agrandissement de domaines.

Chapitre 4

Un Système de preuve

Dans ce chapitre, nous présentons toutes les règles de base de notre outil de vérification. Nous commençons par fournir dans la section 1 un modèle de spécification des propriétés. Le modèle que nous fournissons est extrêmement simple mais est suffisant pour définir le système de règle de preuve. Nous verrons dans le chapitre 9 un modèle plus riche. La section 2 nous permet d'introduire la notion de preuve, et nous présentons les règles de preuve les plus simples liées au modèle polyédrique. Les sections 3 et 4 sont consacrées à l'étude de deux règles de preuves spécifiques au modèle polyédrique. Ces deux règles sont la substitution par valeur et la recherche d'auto-dépendance. Enfin, dans la dernière section, nous conclurons.

1 Spécifier des propriétés dans le modèle polyédrique

Dans cette section, nous nous allons nous intéresser à la spécification des propriétés du système. Nous allons tout d'abord définir la notion de formule, ce qui nous amènera à définir formellement la validité.

Définition 4.1 (Formule polyédrique sur un système d'équations récurrentes) *Une* formule polyédrique *ou plus simplement* formule φ *sur le système S est la donnée d'un domaine $\mathcal{D}$, d'une expression booléenne e construite à partir de variables polyédriques de S et d'une valeur booléenne v. On la note $\mathcal{D} : e \downarrow v$.*

La notion de formule polyédrique que nous donnons ici est très simple et ne permet pas de spécifier un grand nombre de propriétés. En effet, pour pouvoir spécifier une propriété à l'aide d'une telle formule, il faut trouver une expression polyédrique e qui sera vraie sur tout un domaine. Dès que l'on veut prouver des propriétés un peu plus complexes comme l'exclusion mutuelle, nous avons besoin de quantificateurs existentiels et nous ne pouvons pas exprimer la propriété à l'aide d'une formule telle que définie ci-dessus. De plus, nos outils ne permettent de valider que des formules de cette forme. Ainsi, nous enrichirons dans le chapitre 9 cette notion de formule, ce qui nous permettra de spécifier des propriétés plus complexes et nous verrons comment adapter nos outils à la preuve de ces propriétés. Nous supposerons ici que nous avons réussi à exprimer la propriété étudiée par une formule.

Définition 4.2 (Validité) *Soit S un système, soit $\mathcal{D} : e \downarrow v$ une formule construite à partir des variables du système. Soit ρ un environnement sémantiquement correct pour ce système.*

- **Validité d'une formule polyédrique pour un environnement.** *La formule $\mathcal{D} : e \downarrow v$ est* valide pour l'environnement *ρ ce qu'on note $\models_\rho \mathcal{D} : e \downarrow v$ si et seulement si*

$$\mathcal{D} \subset dom(\mathcal{E}(e)(\rho)) \wedge \forall z \in \mathcal{D},\ val(\mathcal{E}(e)(\rho))[z] = v$$

– **Validité d'une formule polyédrique.** *La formule $\mathcal{D} : e \downarrow v$ est* valide*, ce qu'on note $\models \mathcal{D} : e \downarrow v$ si et seulement si elle est valide pour tout environnement sémantiquement correct de S.*

Proposition 4.3 *Soit S un système et soit ρ un environnement sémantiquement correct de S. Soit e, e' deux expressions polyédriques, soit X une variable polyédrique définie par*

$$X = \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \\ \mathcal{D}_n : e_n \end{cases}$$

Soit $\mathcal{D}$ un domaine et soit v une valeur booléenne appartenant à $\{tt, ff\}$. Nous avons alors les équivalences suivantes :

$$\begin{array}{rcl}
\models_\rho \mathcal{D} : e \wedge e' \downarrow tt & \iff & \models_\rho \mathcal{D} : e \downarrow tt \ \textit{et} \ \models_\rho \mathcal{D} : e' \downarrow tt \\
\models_\rho \mathcal{D} : e \vee e' \downarrow ff & \iff & \models_\rho \mathcal{D} : e \downarrow ff \ \textit{et} \ \models_\rho \mathcal{D} : e' \downarrow ff \\
\models_\rho \mathcal{D} : e.d' \downarrow v & \iff & \models_\rho d'(\mathcal{D}) : e \downarrow v \\
\models_\rho \mathcal{D} : X \downarrow v & \iff & \models_\rho \mathcal{D} : \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad\quad \downarrow v \\ \mathcal{D}_n : e_n \end{cases} \\
& \iff & \models_\rho \mathcal{D} \cap \mathcal{D}_1 : e_1 \downarrow v \ \textit{et} \ \ldots \ \textit{et} \ \models_\rho \mathcal{D} \cap \mathcal{D}_n : e_n \downarrow v
\end{array}$$

Démonstration : *Pour démontrer ces équivalences, il suffit d'utiliser la sémantique de e et d'utiliser les règles de distribution du quantificateur universel. Nous ne montrerons que l'équivalence concernant la dépendance.*

$$\begin{array}{rl}
& \models_\rho \mathcal{D} : e.d' \downarrow v \\
\iff & \forall z \in \mathcal{D}' \cap \ dom(\mathcal{E}(e.d')(\rho)),\ val(\mathcal{E}(e.d')(\rho))[z] = v \\
\iff & \forall z \in \mathcal{D}' \cap d'^{-1}(dom(\mathcal{E}(e.d')(\rho))),\ val(\mathcal{E}(e)(\rho)) \circ d'[z] = v \\
\iff & \forall d'(z) \in d'(\mathcal{D}' \cap d'^{-1}(dom(\mathcal{E}(e.d')(\rho)))),\ val(\mathcal{E}(e)(\rho))[d'(z)] = v \\
\iff & \forall z \in d'(\mathcal{D}') \cap d' \circ d'^{-1}(dom(\mathcal{E}(e.d')(\rho)))),\ val(\mathcal{E}(e)(\rho))[z] = v
\end{array}$$

Notons que si d est une dépendance uniforme alors $d(\mathcal{D}')$ est un polyèdre. Sinon, $d(\mathcal{D}')$ est un $\mathbb{Z}$-polyèdre (cf. annexe D). Les règles que nous présenterons dans ce chapitre peuvent toutes êtres généralisés au cas des $\mathbb{Z}$-polyèdres. Cependant, comme dans le chapitre suivant, nous devrons nous restreindre aux dépendances uniformes, nous considérons dès à présent que toutes les dépendances sont uniformes et qu'ainsi $d(\mathcal{D}')$ est un polyèdre. ❑

Il est important de remarquer que nous n'avons pas d'équivalence pour la disjonction avec la valeur *tt*, la conjonction avec la valeur *ff* et la négation. En effet, soit $\mathcal{D} : e \downarrow tt$ une formule, si $e = e' \vee e''$ alors

$$\models_\rho \mathcal{D} : e \downarrow tt \iff \models_\rho \mathcal{D} : e' \downarrow tt \text{ ou } \models \mathcal{D} : e'' \downarrow tt$$

– L'expression $\models_\rho \mathcal{D} : e \downarrow tt$ signifie qu'en tout point z du domaine $\mathcal{D}$ $e'[z]$ ou $e''[z]$ valent vrai. C'est-à-dire que pour un point z du domaine $e[z]$ peut valoir vrai, et pour un autre point z' ce sera $e[z']$ qui vaudra vrai.
– L'expression $\models_\rho \mathcal{D} : e' \downarrow tt \vee \models_\rho \mathcal{D} : e'' \downarrow tt$ signifie que soit sur tout le domaine $\mathcal{D}$, l'expression e' vaut vrai, soit sur tout le domaine $\mathcal{D}$, l'expression e'' vaut vrai.

Nous ne pouvons montrer qu'une implication, pour l'autre nous ne pouvons pas distribuer la quantification universelle dans le « ou ».

$$\begin{array}{lll} \models_\rho \mathcal{D} : e' \vee e'' \downarrow tt & \iff & \mathcal{D} \subset dom(\mathcal{E}(e' \vee e'')(\rho)) \\ & & \wedge \quad \forall z \in \mathcal{D}, val(\mathcal{E}(e' \vee e'')(\rho))[z] = tt \\ & \iff & \mathcal{D} \subset dom(\mathcal{E}(e' \vee e'')(\rho)) \\ & & \wedge \quad \forall z \in \mathcal{D}, val(\mathcal{E}(e')(\rho))[z] = tt \vee val(\mathcal{E}(e'')(\rho))[z] = tt \\ & \not\Longrightarrow & \mathcal{D} \subset dom(\mathcal{E}(e' \vee e'')(\rho)) \\ & & \wedge \quad \forall z \in \mathcal{D}, val(\mathcal{E}(e')(\rho))[z] = tt \\ & & \vee \quad \forall z \in \mathcal{D}, val(\mathcal{E}(e'')(\rho))[z] = tt \end{array}$$

Définition 4.4 (Implication) *Soient S un système, Γ un ensemble de formules et φ une formule sur ce système, on dit que Γ implique φ (φ est une conséquence logique de Γ) ce qu'on note $\Gamma \models \varphi$ si pour tout environnement sémantiquement correct ρ tel que toutes les formules de Γ sont valides, φ est valide pour ρ.*

Définition 4.5 (Contexte) *Soient S un système et $\Gamma \models \varphi$ une implication sur ce système, on appelle contexte l'ensemble des formules polyédriques Γ situé à gauche du symbole $\models$.*

2 Preuve dans le modèle polyédrique

Nous avons dans la section précédente présenté la sémantique des formules polyédriques. Puisqu'il n'est pas possible de prouver qu'une formule est valide en s'appuyant simplement sur la sémantique, nous devons donner une autre caractérisation de la validité sémantique d'une formule. Nous allons maintenant introduire la notion de preuve en utilisant les propriétés du modèle polyédrique. Dans cette section nous présenterons les règles de preuve les plus simples, et nous introduirons la notion de règle sûre.

L'existence d'une preuve pour la formule $\mathcal{D} : e \downarrow v$ dans le système S sous le contexte Γ (Γ représente donc un ensemble de formules) sera notée

$$\Gamma; S \vdash \mathcal{D} : e \downarrow v$$

S'il existe une preuve pour une formule dans le système S, alors la formule est valide :

$$\Gamma; S \vdash \mathcal{D} : e \downarrow v \quad \Longrightarrow \quad \Gamma \models \mathcal{D} : e \downarrow v$$

Comme le but de ce chapitre est avant tout de présenter les différentes techniques utilisées dans les règles de preuve, nous ne tirerons pas parti du contexte. Nous supposons donc que ce contexte est inexistant et l'existence d'une preuve pour la formule $\mathcal{D} : e \downarrow v$ dans le système S sera donc notée pour l'instant :

$$S \vdash \mathcal{D} : e \downarrow v$$

Nous définirons par la suite ce que l'existence d'une preuve signifie. Comme nous supposons que le contexte est vide, les règles que nous présentons ne sont pas suffisantes : nous ne présenterons que les règles d'introduction des opérateurs à droite du séquent. Nous verrons dans le chapitre 9 que des règles similaires peuvent être utilisées sur les formules du contexte.

Définition 4.6 (règles de preuve) *Soient e_1, e_2, e des expressions polyédriques, soit v une valeur appartenant à $\{tt, ff\}$, soit $\mathcal{D}$ un domaine et soit X une variable définie par l'équation suivante :*

$$X = \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \\ \mathcal{D}_n : e_n \end{cases}$$

Les règles de preuve sont les suivantes :

– *Axiomes :*

$$\frac{}{S \vdash \mathcal{D} : \mathit{True} \downarrow \mathit{tt}}\ \text{AX1} \qquad \frac{}{S \vdash \emptyset : e \downarrow v}\ \text{AX2} \qquad \frac{}{S \vdash \mathcal{D} : \mathit{False} \downarrow \mathit{ff}}\ \text{AX3}$$

– *Règles de décomposition des opérandes :*

$$\frac{S \vdash \mathcal{D} : e_1 \downarrow \mathit{tt},\ \ S \vdash \mathcal{D} : e_2 \downarrow \mathit{tt}}{S \vdash \mathcal{D} : e_1 \wedge e_2 \downarrow \mathit{tt}}\ \text{ET} \qquad \frac{S \vdash \mathcal{D} : e_1 \downarrow \mathit{ff},\ \ S \vdash \mathcal{D} : e_2 \downarrow \mathit{ff}}{S \vdash \mathcal{D} : e_1 \vee e_2 \downarrow \mathit{ff}}\ \text{OU}$$

– *Règle de la négation :*

$$\frac{S \vdash \mathcal{D} : e \downarrow \mathit{ff}}{S \vdash \mathcal{D} : \neg e \downarrow \mathit{tt}}\ \text{NEG1} \qquad \frac{S \vdash \mathcal{D} : e \downarrow \mathit{tt}}{S \vdash \mathcal{D} : \neg e \downarrow \mathit{ff}}\ \text{NEG2}$$

– *Règle de la dépendance :*

$$\frac{S \vdash d(\mathcal{D}) : e : v}{S \vdash \mathcal{D} : e.d \downarrow v}\ \text{DEP}$$

– *Règle de l'équation :*

$$\frac{S \vdash \mathcal{D} : \begin{cases} \mathcal{D}_1 : e_1 & \\ \vdots & \downarrow v \\ \mathcal{D}_n : e_n & \end{cases}}{S \vdash \mathcal{D} : X \downarrow v}\ \text{EQ.}$$

– *Règle de séparation des branches :*

$$\frac{S \vdash \mathcal{D} \cap \mathcal{D}_1 : e_1 \downarrow v,\ \ \ldots,\ \ S \vdash \mathcal{D} \cap \mathcal{D}_n : e_n \downarrow v}{S \vdash \begin{cases} \mathcal{D}_1 : e_1 & \\ \vdots & \downarrow v \\ \mathcal{D}_n : e_n & \end{cases}}\ \text{SEP.}$$

Certaines de ces règles sont définies pour des formules contenant une expression quelconque, d'autres pour des formules dont l'expression est une variable, nous avons une règle concernant le passage d'une variable à son expression, mais nous n'avons pas de règle pour passer d'une expression à une variable. Dans la section suivante, nous allons définir une règle de recherche d'auto-dépendance. Pour pouvoir appliquer cette règle, nous aurons tout d'abord besoin d'une règle permettant de passer d'une expression polyédrique à une variable. Cette règle est appelée règle de la nouvelle variable :

$$\frac{S' \vdash \mathcal{D} : X' \downarrow v}{S \vdash \mathcal{D} : e \downarrow v}\ \text{NOUV}$$

où X' est une variable fraîche définie par l'expression e sur le domaine de e, et où le système S' est le système S auquel a été ajoutée la variable X' et l'équation définissant cette variable.

Remarque 4.7 (Axiome AX2**)** *L'axiome* AX2 *pose problème dans la librairie polyédrique. En effet, un domaine est vide dans la librairie polyédrique s'il est vide au sens rationnel. Or, nous travaillons sur des polyèdres à points entiers. Ainsi, nous souhaitons pouvoir tester si un polyèdre est vide au sens entier et dans ce cas il n'est pas obligatoirement vide au sens rationnel : il suffit par exemple d'étudier le domaine suivant* $\{t \mid 1 \leq 10t \leq 9\}$ *est vide au sens entier mais non vide au sens rationnel. L'implémentation de cette règle ne tient pas compte de cette difficulté et nous ne testons les domaines*

vides qu'au sens rationnels. En pratique cette approximation ne nous a pas pour le moment posé de problème. Il sera cependant nécessaire de le prendre en compte. Pour cela nous pourrons utiliser des outils comme Omega [53], PIP [35],ou les polynômes d'Ehrhardt [98] qui permettent de compter le nombre de points d'un polyèdre.

La notion de règle ne nous intéresse que si elle est sûre.

Définition 4.8 (Règle sûre) *Soit*

$$\frac{S_1 \vdash \mathcal{D}_1 : e_1 \downarrow v_1 \qquad \dots \qquad S_n \vdash \mathcal{D}_n : e_n \downarrow v_n}{S \vdash \mathcal{D} : e \downarrow v}$$

une règle où les systèmes S_1,..., S_n sont des sursystèmes de S (cf. page 44). Cette règle est sûre si et seulement si nous avons une conséquence logique entre les prémisses et la conclusion de la règle pour le système S'. C'est-à-dire :

$$\{\mathcal{D}_1 : e_1 \downarrow v_1 \wedge \dots \wedge \mathcal{D}_n : e_n \downarrow v_n\} \models \mathcal{D} : e \downarrow v$$

On dira qu'une règle est *inversible* si la règle est sûre et que les prémisses sont une conséquence logique de la conclusion.

Théorème 4.9 *Les règles de preuves présentées dans la définition 4.6 sont sûres.*

Démonstration : *Nous n'allons démontrer la sûreté que pour une des règles :*

$$\frac{S \vdash \mathcal{D} : e_1 \downarrow \mathit{ff}, \;\; S \vdash \mathcal{D} : e_2 \downarrow \mathit{ff}}{S \vdash \mathcal{D} : e_1 \vee e_2 \downarrow \mathit{ff}} \text{ OU}$$

Nous devons montrer que :

$$\forall \rho \propto S, \models_\rho \mathcal{D} : e_1 \downarrow \mathit{ff} \wedge \models_\rho \mathcal{D} : e_2 \downarrow \mathit{ff} \implies \models_\rho \mathcal{D} : (e_1 \vee e_2) \downarrow \mathit{ff}$$

Il suffit d'appliquer la proposition 4.3 pour la disjonction et nous obtenons l'équivalence suivante :

$$\forall \rho \propto S, \models_\rho \mathcal{D} : e_1 \downarrow \mathit{ff} \wedge \models_\rho \mathcal{D} : e_2 \downarrow \mathit{ff} \iff \models_\rho \mathcal{D} : (\neg e_1) \wedge (\neg e_2) \downarrow \mathit{ff}$$

Pour chaque règle nous pouvons montrer l'équivalence, elles sont donc inversibles. ❑

Ces règles ne sont pas suffisantes pour effectuer des preuves. Nous n'avons par exemple pas de règle pour la formule suivante :

$$\mathcal{D} : e_1 \vee e_2 \downarrow \mathit{ff}$$

Nous allons développer dans les sections suivantes d'autres types de règles qui pourront s'appliquer sur la formule précédente.

3 Substitution par constante

Dans cette section, nous allons présenter la règle de substitution par constante. L'idée est, comme nous l'avons vu dans le chapitre précédent, de « réécrire » l'expression d'une variable X en une expression plus simple qui définira une variable X_{subs}. Cette expression ne contiendra plus d'occurrence de X dans certaines de ses branches. Notre nouvelle règle sera la suivante :

$$\frac{S' \vdash \mathcal{D} : X_{subs} \downarrow v}{S \vdash \mathcal{D} : X \downarrow v} \text{ SUBS}$$

où S' est le sursystème de S contenant la variable X_{subs}. Nous avons esquissé dans le chapitre 3 comment obtenir l'expression de X_{subs}. Dans la section 3.1, nous définirons formellement la variable X_{subs}. Dans la section 3.2, nous prouverons que la règle de substitution par constante est sûre. Enfin, nous verrons une règle de simplification qui permettra de simplifier les résultats obtenus grâce à la règle de substitution par constante.

3.1 Définitions de X_{subs} par substitution.

Avant de donner l'équation définissant X_{subs}, nous introduisons différentes notions. Soit X une variable polyédrique définie sur le domaine $\mathcal{D}_X$. Soit $\mathcal{D}$ un sous domaine de $\mathcal{D}_X$.

Dans l'exemple du chapitre 3, nous avons vu que les instances de X n'étaient pas toutes substituées. La première définition que nous donnons permet de définir l'ensemble des points pour lesquels les instances ne seront pas substituées par la constante. Cet ensemble est appelé ensemble des points externes. Les instances définies sur ces points externes :

- soit ne dépendent pas d'instances de X,
- soit dépendent d'instances de X définies sur des points extérieurs au domaine $\mathcal{D}$.

Définition 4.10 (Points externes) *On définit l'ensemble des* points externes *de X pour le domaine $\mathcal{D}' \subset \mathcal{D}_X$ et la dépendance d par rapport au domaine $\mathcal{D}$, que l'on note $\mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}$, comme étant l'ensemble des points z de $\mathcal{D}'$ tels que les instances de variables dont $X[z]$ dépend sont soit des instances de variables différentes de X, soit des instances de X qui ne dépendent pas de X au point $d(z)$, soit des instances de X au point $d(z)$ qui ne sont pas dans $\mathcal{D}$:*

$$\begin{aligned}\mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}} = \{z \in \mathcal{D}' \cap \mathcal{D} \mid \forall Y, \forall z', X[z] \rightarrow Y[z'] \implies & \quad Y \neq X \\ & \vee \quad (Y = X \wedge z' \neq d(z)) \\ & \vee \quad (Y = X \wedge z' = d(z) \wedge z' \notin \mathcal{D})\}\end{aligned}$$

On omettra $\mathcal{D}'$, si $\mathcal{D}' = \mathcal{D}_X$, et nous le noterons alors $\mathcal{XP}_{X,d,\mathcal{D}}$.

Exemple 4.11 *En reprenant l'exemple du chapitre précédent, nous définissons $\mathcal{XP}_{X,d_1,\mathcal{D}_1,\mathcal{D}}$ et $\mathcal{XP}_{X,d_2,\mathcal{D}_2,\mathcal{D}}$ de la manière suivante :*

$$\begin{aligned}\mathcal{XP}_{X,d_1,\mathcal{D}_1,\mathcal{D}} &= (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D} \\ \mathcal{XP}_{X,d_2,\mathcal{D}_2,\mathcal{D}} &= (\mathcal{D}_2 \setminus d_2^{-1}(\mathcal{D})) \cap \mathcal{D}\end{aligned}$$

Nous représentons en gris foncé sur la figure 4.1, les points de $\mathcal{XP}_{X,d_1,\mathcal{D}_1,\mathcal{D}}$.

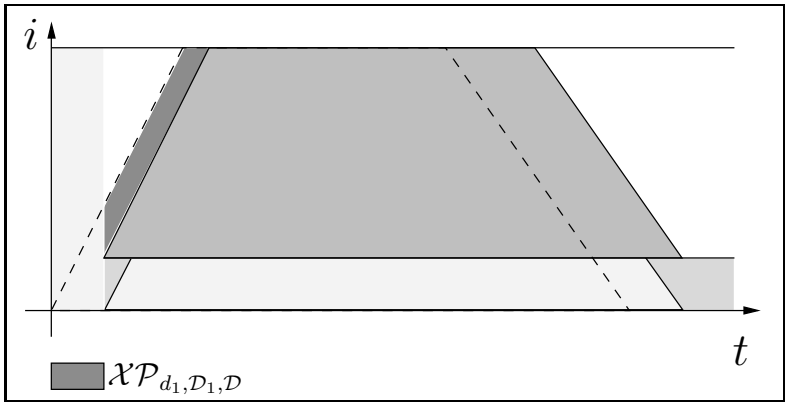


FIG. 4.1 – Domaine $\mathcal{XP}_{X,d_1,\mathcal{D}_1,\mathcal{D}}$ pour l'exemple présenté dans le chapitre 3

Substitution par constante. Dans ce paragraphe, nous allons voir comment obtenir la variable X_{subs} : il suffit en fait de substituer les instances n'appartenant pas aux points externes par la constante de la formule étudiée. Cette variable sera définie dans le système S'. Nous définissons tout d'abord une notation pour représenter la substitution d'une expression par une autre :

Notation 4.12 (Substitution) *Soient e, h, g trois expressions polyédriques telles que h soit une sous expression de e, et g soit définie sur un domaine de même dimension que h. Nous notons $e[g/h]$ la substitution de l'expression h par l'expression g dans l'expression e.*

Nous noterons abusivement $e[v/h]$ la substitution de l'expression h :
- par la constante polyédrique *True* si v représente la valeur booléenne *tt*,
- ou par la constante polyédrique *False* si v représente la valeur booléenne *ff*.

La variable v représente une valeur sémantique, ce n'est pas une expression syntaxique.

Lemme 4.13 (Substitution par constante) *Soit ρ un environnement sémantiquement correct et e une expression booléenne qui dépend de $X.d$. Alors, si ρ valide $X.d$ pour la valeur v sur un domaine $\mathcal{D}$, les expressions e et $e[v/X.d]$ sont sémantiquement équivalentes (cf. page 44) pour l'environnement ρ sur le domaine $\mathcal{D}$.*

Démonstration : *La démonstration est immédiate par récurrence sur la profondeur des expressions ; il suffit de remarquer que $val(\mathcal{E}(X.d)\rho) = v$.* □

Définition de X_{subs}. Soit e l'expression polyédrique définissant la variable X, soit e' l'expression $e[v/X.d]$, soit $\mathcal{D}'$ le domaine de l'expression e. On définit une nouvelle variable X_{subs} par l'équation suivante :

$$X_{subs} = \begin{cases} \mathcal{D}_X \setminus (\mathcal{D}' \cap d^{-1}(\mathcal{D}) \cap \mathcal{D}) : X \\ \mathcal{D}' \cap d^{-1}(\mathcal{D}) \cap \mathcal{D} : e' \end{cases} \tag{4.1}$$

Exemple 4.14 *Reprenons l'exemple présenté dans le chapitre introductif 3, l'expression e est définie par $X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2)$, la dépendance d correspond à d_1 et e' est donc définie par $Y.\delta_1 \vee Y.\delta_2$. On obtient ainsi la définition suivante pour X_{subs} :*

$$X_{subs} = \begin{cases} \mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}) : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \\ d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1 : Y.\delta_1 \vee Y.\delta_2 \\ \mathcal{D}_2 : X.d_2 \\ \mathcal{D}_3 : True \end{cases}$$

3.2 Sûreté

Nous allons montrer que la règle de substitution est sûre, c'est-à-dire que nous devons prouver que :

$$\{\mathcal{D} : X_{subs} \downarrow v\} \models \mathcal{D} : X \downarrow v$$

Nous devons donc prouver l'implication

$$\forall \rho \propto S', \models_\rho \mathcal{D} : X_{subs} \downarrow v \implies \models_\rho \mathcal{D} : X \downarrow v \tag{4.2}$$

En fait, nous prouverons l'équivalence, ainsi la règle de substitution sera inversible.

Proposition 4.15 *Soit ρ un environnement sémantiquement correct, alors*

$$\models_\rho \mathcal{D} : X_{subs} \downarrow v \iff \models_\rho \mathcal{D} : X \downarrow v$$

Démonstration : *La démonstration est faite par récurrence forte*[1] *sur l'indice temporel* t_z *de* z*, où* z *est dans* $\mathcal{D}$*. Nous rappelons que les domaines sont ordonnancés, et qu'ainsi l'un des indices représente le temps. Soit* t_0 *une valeur fixée de l'indice temporel* t_z*. Le domaine* D_{t_0} *(cf. figure 4.2) est une restriction du domaine* $\mathcal{D}$ *aux points dont la valeur de l'indice temporel est inférieure à* t_0 *:*

$$\mathcal{D}_{t_0} = \{z | t_z \leq t_0 \wedge z \in \mathcal{D}\}$$

Remarquons tout d'abord que la suite $(\mathcal{D}_t)_{t\in\mathbb{N}}$ *de domaines est une suite croissante qui converge vers*

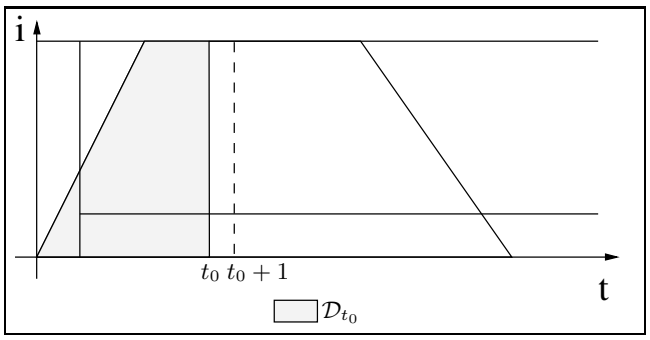


FIG. 4.2 – Domaine $\mathcal{D}_{t_0}$

$\mathcal{D}$*. Notre hypothèse de récurrence est la suivante :*

$$\models_\rho \mathcal{D}_{t_0} : X_{subs} \downarrow v \Longleftrightarrow \models_\rho \mathcal{D}_{t_0} : X \downarrow v$$

- *Pour* $t_0 = \min\{t_z \mid z \in \mathcal{D}\}$*, le domaine* $\mathcal{D}_{t_0}$ *est un sous-ensemble de* $\mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}$*. En effet pour tout point* z *appartenant à* $\mathcal{D}_{t_0}$*,* $d(z)$ *n'est pas dans* $\mathcal{D}$ *car le système est ordonnancé et donc* $t_{d(z)}$ *est inférieur à* t_0*. Par définition de* X_{subs}*, nous savons que sur le domaine* $\mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}$*, la variable* X_{subs} *est définie par la variable* X*. L'équivalence est triviale.*
- *Nous supposons que notre hypothèse de récurrence est vraie pour toute valeur* t *inférieure ou égale à* t_0*, nous devons montrer l'équivalence suivante :*

$$\models_\rho \mathcal{D}_{t_0+1} : X_{subs} \downarrow v \Longleftrightarrow \models_\rho \mathcal{D}_{t_0+1} : X \downarrow v$$

Les domaines $\mathcal{D}_{t_0}$ *et* $\mathcal{D}_{t_0+1}$ *sont représentés sur la figure 4.3. Soit* z_0 *un point de* $\mathcal{D}_{t_0+1} \backslash \mathcal{D}_{t_0}$*, alors,*

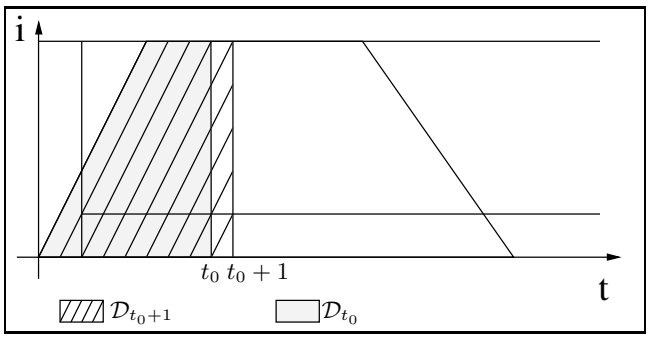


FIG. 4.3 – Domaines $\mathcal{D}_{t_0}$ et $\mathcal{D}_{t_0+1}$

z_0 *est soit un point de* $(\mathcal{D}_{t_0+1} \setminus \mathcal{D}_{t_0}) \cap \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}$*, soit un point de* $\mathcal{D}_{t_0+1} \setminus (\mathcal{D}_{t_0} \cup \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}})$ *:*

[1] Soient $x_0, \ldots, x_n, \ldots$ un ensemble de propriétés. Dans un raisonnement par récurrence forte, si on sait que pour un n quelconque, si toutes les propriétés $x_0, \ldots, x_n$ sont vraies alors x_{n+1} est vraie, alors on aura montré que toutes les propriétés x_n sont vraies.

– *Si z_0 est un point de $(\mathcal{D}_{t_0+1} \setminus \mathcal{D}_{t_0}) \cap \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}$, par définition de X_{subs} sur ce domaine, on a $X_{subs} \cong X$.*
– *Si z_0 est un point de $\mathcal{D}_{t_0+1} \setminus (\mathcal{D}_{t_0} \cup \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}})$, alors sur ce domaine, nous savons que $X[z_0]$ est définie par l'équation suivante :*

$$X[z_0] = g(\dots, X[d(z_0)], \dots)$$

Comme nous travaillons sur un système ordonnancé, le point $d(z_0)$ est dans $\mathcal{D}_{t_0}$.
– *Nous commençons par montrer l'implication*

$$\models_\rho \mathcal{D}_{t_0+1} : X_{subs} \downarrow v \implies \models_\rho \mathcal{D}_{t_0+1} : X \downarrow v$$

Supposons que X_{subs} soit valide sur le domaine $\mathcal{D}_{t_0+1}$ pour l'environnement ρ, alors, nous avons en particulier que X_{subs} est valide sur $\mathcal{D}_{t_0}$:

$$\models_\rho \mathcal{D}_{t_0+1} : X_{subs} \downarrow v \implies \models_\rho \mathcal{D}_{t_0} : X_{subs} \downarrow v$$

En appliquant notre hypothèse de récurrence, nous savons que :

$$\models_\rho \mathcal{D}_{t_0} : X \downarrow v$$

On peut en particulier l'appliquer au point $d(z_0)$. Ainsi, nous montrons que :

$$\models_\rho \mathcal{D}_{t_0+1} \setminus (\mathcal{D}_{t_0} \cup \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}) : X.d \downarrow v$$

Or, d'après le lemme 4.13, les expressions g et $g[v/X.d]$ sont sémantiquement équivalentes sur le domaine $\mathcal{D}_{t_0+1} \setminus (\mathcal{D}_{t_0} \cup \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}})$. Par définition de X_{subs} et comme $\mathcal{D}_{t_0+1} \setminus (\mathcal{D}_{t_0} \cup \mathcal{XP}_{X,d,\mathcal{D}',\mathcal{D}}) \subset \mathcal{D}' \cap d^{-1}(\mathcal{D})$, on obtient :

$$\models_\rho \mathcal{D}_{t_0+1} : X_{subs} \downarrow v \implies \models_\rho \mathcal{D}_{t_0+1} : X \downarrow v$$

– *L'implication*

$$\models_\rho \mathcal{D}_{t_0+1} : X \downarrow v \implies \models_\rho \mathcal{D}_{t_0+1} : X_{subs} \downarrow v$$

se prouve de manière similaire

❑

Dans la proposition précédente seule une occurrence de X a été substituée, nous pouvons étendre l'équivalence à toutes les occurrences de X.

Corollaire 4.16 *Soit ρ un environnement sémantiquement correct, soit X une variable, et v une valeur. Soient $d_1, \dots, d_n$ les auto-dépendances (cf. page 30) de X, on note X_{subs} la variable obtenue à partir de X en transformant récursivement l'équation (4.1) pour toutes les dépendances $d_1, \dots d_n$. Alors, on a*

$$\models_\rho \mathcal{D} : X_{subs} \downarrow v \iff \models_\rho \mathcal{D} : X \downarrow v$$

Ce dernier corollaire nous permet de conclure que la règle de substitution par constante est sûre et comme nous avons montré l'équivalence qu'elle est inversible.

3.3 Simplification propositionnelle

Après la substitution par constante, nous obtenons des expressions booléennes contenant des constantes. Ces formules sont simplifiées en adaptant les règles de simplification classique de logique au modèle polyédrique. Soit e une expression polyédrique, nous appliquons récursivement sur la structure syntaxique de e les transformations suivantes :

$$\begin{array}{rcl@{\qquad}rcl} e \wedge \mathit{True} & \rightsquigarrow & e & \mathit{True} \wedge e & \rightsquigarrow & e \\ e \wedge \mathit{False} & \rightsquigarrow & \mathit{False} & \mathit{False} \wedge e & \rightsquigarrow & \mathit{False} \\ e \vee \mathit{True} & \rightsquigarrow & \mathit{True} & \mathit{True} \vee e & \rightsquigarrow & \mathit{True} \\ e \vee \mathit{False} & \rightsquigarrow & e & \mathit{False} \vee e & \rightsquigarrow & e \end{array}$$

jusqu'à ce qu'il n'y ait plus de constantes dans l'expression ou que l'expression soit une constante. Soit X une variable polyédrique, on note X_{simp} la variable définie par l'expression de X sur laquelle les règles de simplification ont été appliquées. La règle de preuve de simplification se définit par

$$\frac{S' \vdash \mathcal{D} : X_{simp} \downarrow v}{S \vdash \mathcal{D} : X \downarrow v}\ \text{SIMP}$$

Cette règle de preuve est sûre. Il suffit d'appliquer les règles de définition de la définition de la sémantique. Cette règle est même inversible : nous pouvons montrer l'équivalence $\forall \rho \propto,\ \models_\rho \mathcal{D} : X_{simp} \downarrow v \iff \models_\rho \mathcal{D} : X \downarrow v$.

4 Recherche d'auto-dépendances

Dans cette section, nous allons présenter notre deuxième technique de preuve. Nous avons jusqu'à présent vu comment substituer une variable par sa valeur. Cependant dans certains cas cette méthode ne fonctionne pas car la variable n'est pas définie récursivement. L'idée est donc de réécrire récursivement cette variable, c'est-à-dire de rechercher des auto-dépendances. Nous allons effectuer ce travail en deux étapes. Tout d'abord, à partir d'une expression m définissant une variable M, nous recherchons dans toutes les expressions des autres variables s'il existe une dépendance d tel que $m.d$ est une sous-expression. L'expression m est appelée motif. Si $m.d$ est une sous-expression, nous la substituons par l'expression $M.d$. Nous appliquerons ensuite ce résultat à la recherche d'auto-dépendances.

Exemple 4.17 *Prenons par exemple les deux variables M et W suivantes :*

$$M = X.(t, i \to t-1, i-1) \vee Y.(t, i \to t-1, i)$$
$$W = X.(t, i \to t-1, i-2) \vee Y.(t, i \to t-1, i-1) \vee W.(t, i \to t-1, i+3)$$

où X et Y sont deux variables polyédriques.
Le motif m correspond à l'expression $X.(t, i \to t-1, i-1) \vee Y.(t, i \to t-1, i)$. Nous recherchons une dépendance d telle que $m.d$ est une sous-expression de l'expression définissant W. Prenons la dépendance $(t, i \to t, i-1)$, l'expression $m.d$ vaut $X.(t, i \to t-1, i-1) \circ (t, i \to t, i-1) \vee Y.(t, i \to t-1, i) \circ (t, i \to t, i-1)$, c'est-à-dire, $X.(t, i \to t-1, i-2) \vee Y.(t, i \to t-1, i-1)$. Ainsi $m.d$ est une sous-expression de W, et nous la substituons par $M.d$. La variable W est donc maintenant définie par :

$$W = M.(t, i \to t, i-1) \vee W.(t, i \to t-1, i+3)$$

Nous définirons tout d'abord dans la section 4.1 la notion de motif et comment rechercher un motif. Nous verrons que substituer, dans une expression, un motif par la variable le définissant préserve la sémantique. Dans la section 4.2, nous verrons comment résoudre des systèmes d'équations sur les dépendances. Enfin, dans la section 4.3, nous appliquerons la recherche de motif pour créer une règle de preuve de recherche d'auto-dépendance.

4.1 Recherche de motifs

Soient $\barwedge$ un opérateur booléen (appartenant à $\{\vee, \wedge\}$).

Définition 4.18 (valeur absorbante, valeur neutre)
- *Nous appelons valeur absorbante pour l'opérateur $\barwedge$, la valeur tt si $\barwedge$ est une disjonction, ou ff si $\barwedge$ est une conjonction.*
- *Nous appelons valeur neutre pour l'opérateur $\barwedge$, la valeur ff si $\barwedge$ est une disjonction, ou tt si $\barwedge$ est une conjonction.*

Définition 4.19 ($\barwedge$-expression, forme $\barwedge$-normale)
- *Nous appelons $\barwedge$-expression, une expression e telle que $\barwedge$ se trouve à la racine de l'arbre syntaxique abstrait de e.*
- *Nous appelons forme $\barwedge$-normale,*
  - *une forme normale disjonctive si $\barwedge$ est une disjonction,*
  - *une forme normale conjonctive si $\barwedge_1$ (resp. $\barwedge_2$) est une conjonction.*

Définition 4.20 (Motif) *Un motif m est une expression polyédrique définie par un unique type d'opération booléenne.*

Ainsi, l'expression $X.(t, i \rightarrow t-1, i-1) \vee Y.(t, i \rightarrow t-1, i)$ est un motif mais l'expression $X.(t, i \rightarrow t-1, i-1) \vee (Y.(t, i \rightarrow t-1, i) \wedge Y.(t, i \rightarrow t, i-1))$ n'en est pas un.

Pour comprendre pourquoi nous demandons qu'un motif soit définit par un uninique type d'opération, il faut tout d'abord savoir comment sont obtenus les motifs : ils peuvent être obtenus de deux manières :
- soit ils correspondent à un invariant donné par l'utilisateur et dans ce cas ils sont associés à une valeur v
- soit ils sont obtenus au cours d'une preuve après une substitution par constante et après avoir appliqué les règles de décomposition de $\vdash$, ce sont des sous-buts de preuve sur lesquels aucune règle ne peut être appliquée. Ils sont là encore associés à une valeur v.

Dans les deux cas, les motifs sont associés à une valeur v. Cette association induit l'unicité du type d'opération d'un motif. Nous montrons que la valeur v est nécessairement absorbante. Sinon, supposons $m = \bigbarwedge_{i=1}^{n}{}_1 m_i$. Si v est neutre, on peut alors appliquer une des règles de décomposition des opérandes (règles ET et OU), et on obtient les sous-buts suivants :

$$S \vdash m_1 \downarrow v, \ldots ; S \vdash m_n \downarrow v$$

On peut appliquer les règles de décomposition par rapport à $\vdash$, nous obtenons donc n nouveaux sous-buts, et chacun de ses sous-buts peut être un motif.

Soit M une variable définie par l'équation suivante :

$$M = \bigbarwedge_{i=1}^{l_M}{}_1 X_i.d_i$$

Soit v la valeur absorbante pour $\barwedge_1$. Soit W une variable définie par l'équation suivante :

$$W = \bigbarwedge_{j=1}^{l_W}{}_2 \bigbarwedge_{i=1}^{l_j}{}_1 X'_i.d'_i$$

Nous allons utiliser le motif m définissant la variable[2] M pour réécrire l'expression définissant la variable W.

L'idée est de substituer chaque sous terme de W qui correspond au motif de M par une occurrence de M instanciée avec les dépendances adéquates.

Soit,

- T_W l'ensemble de toutes les $\rtimes_1$-expressions de W,
- τ un élément de T_W,
- *Var*(τ) l'ensemble de toutes les paires (V, d) où V est une variable et d une dépendance, telles que $V.d$ apparaît dans la définition de τ,
- $Used(M)$ l'ensemble de toutes les paires (V, d) où V une variable et d une dépendance telle que $V.d$ apparaît dans la définition de la variable M.

Exemple 4.21 *Soient M et W définies de la manière suivante :*

$$\begin{array}{rcl} M & = & X.d_1 \vee Y.d_2 \\ W & = & (X.\delta_1 \vee X.\delta_2 \vee Y.\delta_3) \wedge (X.\delta_4 \vee W.\delta_5) \end{array}$$

Le motif m est défini par $X.d_1 \vee Y.d_2$ et la valeur v associée est tt. Cette valeur est absorbante pour $\vee$. Le but est de faire apparaître une dépendance entre W et M.

On a donc

$$\begin{array}{rcl} Used(M) & = & \{(X, d_1), (Y, d_2)\} \\ T_W & = & \{(X.\delta_1 \vee X.\delta_2 \vee Y.\delta_3), (X.\delta_4 \vee W.\delta_5)\} \end{array}$$

Prenons par exemple $\tau = X.\delta_1 \vee X.\delta_2 \vee Y.\delta_3$, alors on a :

$$\textit{Var}(\tau) = \{(X, \delta_1), (X, \delta_2), (Y, \delta_3)\}$$

Nous cherchons une dépendance d telle que $M.d$ correspond à une sous expression de τ. Si une telle dépendance existe, alors chaque variable apparaissant dans M apparaît aussi dans τ. Ainsi, il doit exister une injection φ entre $Used(M)$ et *Var*(τ) qui associe chaque paire de $Used(M)$ à une autre paire de *Var*(τ) de telle sorte que les premiers éléments des deux paires sont égaux.

$$\begin{array}{rccc} \varphi : & Used(M) & \rightarrow & \textit{Var}(\tau), \\ & (V, d) & \mapsto & (V, d') \end{array}$$

Si φ n'existe pas alors M n'est pas un motif pour τ.

Exemple 4.22 Suite de l'exemple 4.21. *Nous avons ici deux injections possibles :*

$$\begin{array}{rcl} \varphi_1 : & (X, d_1) \mapsto (X, \delta_1) \\ & (Y, d_2) \mapsto (Y, \delta_3) \end{array} \quad \begin{array}{rcl} \varphi_2 : & (X, d_1) \mapsto (X, \delta_2) \\ & (Y, d_2) \mapsto (Y, \delta_3) \end{array}$$

Nous supposons donc que φ est bien définie et que c'est une injection. Par extension, nous notons φ, l'application associant d à d' quand $\varphi(V, d) = (V, d')$. D'après la définition de M, nous savons que pour toute dépendance d_M, l'expression M est définie par :

$$M.d_M = \bigrtimes_{i=1}^{l_M}{}_1 X'_i.d'_i \circ d_M$$

Ainsi, si (V, d) est une paire de $Used(M)$, alors $(V, d \circ d_M)$ est une paire de $Used(M.d_M)$. Comme nous souhaitons trouver une dépendance d'' telle que $M.d''$ est une sous-expression de τ, il faut trouver

[2]Nous rappelons que M n'est pas un motif, mais une variable, en revanche l'expression définissant M est un motif. On note cette expression m.

une dépendance d'' telle que pour toute paire (V, d) de $Used(M)$, $d \circ d'' = \varphi(d) = d'$. Notons $\mathcal{S}(\varphi)$, l'ensemble de toutes les dépendances telles que M s'associe à τ. Nous devons résoudre le système suivant :

$$\mathcal{S}(\varphi) = \cap_{(V,d)\in Used(X)}\{d'' \mid d \circ d'' = \varphi(d)\} \tag{4.3}$$

Exemple 4.23 Suite de l'exemple 4.21.

$$\begin{array}{rcl} \mathcal{S}(\varphi_1) & = & \{d'' \mid d_1 \circ d'' = \delta_1\} \cap \{d'' \mid d_2 \circ d'' = \delta_3\} \\ \mathcal{S}(\varphi_2) & = & \{d'' \mid d_1 \circ d'' = \delta_2\} \cap \{d'' \mid d_2 \circ d'' = \delta_3\} \end{array}$$

Ainsi, si d' est une dépendance de $\mathcal{S}(\varphi_1) \cup \mathcal{S}(\varphi_2)$, la variable W peut se réécrit de la manière suivante :

$$W = M.d' \wedge (X.\delta_4 \vee W.\delta_5) \tag{4.4}$$

Cette réécriture pose cependant problème. Supposons que nous ayons en plus du motif m, un motif $m' = X.\delta'_1 \wedge X.\delta'_2$ définissant une variable M'. Si nous avions commencé par rechercher le motif m' dans la variable W, nous aurions pu (sous certaines conditions sur les dépendances) substituer l'expression $X.\delta_1 \wedge X.\delta_2$ par la variable M'. Cette substitution n'est plus possible si nous avons introduit la variable M. Ainsi pour permettre la substitution par M et par M' et ainsi permettre de substituer tous les motifs, nous préférons réécrire W de la manière suivante :

$$W = (X.\delta_1 \wedge X.\delta_2 \wedge Y.\delta_3 \wedge M.d') \wedge (X.\delta_4 \vee W.\delta_5)) \tag{4.5}$$

Les expressions $(X.\delta_1 \wedge X.\delta_2 \wedge Y.\delta_3 \wedge M.d') \wedge (X.\delta_4 \vee W.\delta_5))$ et $M.d' \wedge (X.\delta_4 \vee W.\delta_5)$ sont bien sûr équivalentes.

En supposant maintenant que $\mathcal{D} : M \downarrow v$ est une formule valide, et en appliquant la substitution sur W, on obtient alors :

$$\begin{array}{rcl} W & = & \mathcal{D}_W \setminus {d'}^{-1}(D) : (X.\delta_1 \wedge X.\delta_2 \wedge Y.\delta_3 \wedge M.d') \wedge (X.\delta_4 \vee W.\delta_5) \\ & & {d'}^{-1}(D) : X.\delta_4 \vee W.\delta_5 \end{array}$$

Pour trouver toutes les dépendances qui s'associent à τ, nous faisons l'union de $\mathcal{S}(\varphi)$ pour toutes les injections possibles. Remarquons que toutes les dépendances de $\mathcal{S}(\varphi)$ ont la même dimension. Puisque (V, d) est dans $Used(M)$, l'expression $V.d$ a la même dimension que M. Par conséquent, toutes les matrices représentant les dépendances de $Used(M)$ ont le même nombre de colonnes. Nous pouvons conclure que tout les éléments de l'ensemble (4.3) ont le même nombre de lignes. De manière similaire comme toutes les matrices représentant $\varphi(d)$ partagent le même nombre de lignes, chaque élément de l'ensemble (4.3) partage le même nombre de colonnes.

Notons W_M la variable W réécrite avec la variable M définie par le motif m. Nous allons montrer que W et W_M sont sémantiquement équivalentes (*cf.* page 44). Pour cela nous devons d'abord prouver que substituer une variable par sa définition préserve la sémantique.

Proposition 4.24 *Soit S un système, soient X et Y deux variables de ce système telles que $X \rightarrow Y$:*

$$\begin{array}{l} X = f \\ Y = e \end{array}$$

où f et e sont deux expressions polyédriques, on a alors l'équivalence sémantique suivante :

$$f \cong_{\mathcal{D}_X} f[e/Y]$$

Démonstration : *La démonstration est donnée en annexe A.* □

Nous pouvons maintenant en déduire l'équivalence sémantique entre W et W_M pour tout environnement sémantiquement correct. En effet, si nous notons f_M l'expression définissant W_M et f l'expression définissant W, $f_M[m/M]$ est la même expression que f, elles sont donc sémantiquement équivalentes, et comme f_M est sémantiquement équivalent à f d'après la proposition précédente, on en conclut que $W \cong W_M$.

4.2 Résoudre un système de dépendances

Résoudre une équation $d \circ d'' = \varphi(d) \circ d$ dans l'ensemble (4.3) revient à calculer une matrice X telle que $AX = B$, où A et B sont des matrices connues. Comme nous manipulons des matrices à coefficients entiers, X est solution d'une équation diophantienne [91]. La librairie polyédrique offre un algorithme permettant de résoudre de tels systèmes. Cet algorithme s'appuie sur les formes normales hermitiennes. Le calcul de l'intersection de l'ensemble des solutions de tels systèmes est présenté dans l'annexe C.

4.3 Application à la recherche d'auto-dépendance

Rappelons que la méthode de substitution par constante peut être appliquée sur une variable W seulement si W dépend d'elle même. Nous allons utiliser la recherche de motif pour définir une nouvelle règle : l'idée est de réécrire W de sorte que W dépende d'elle même. Soit W une variable définie par l'équation suivante :

$$W = \underset{i}{\Join_1} X_i.d_i$$

L'expression $\Join_{1i} X_i.d_i$ sera le motif m. Nous remplaçons dans W toutes les variables apparaissant dans la partie droite par leur définition, et nous réécrivons l'équation sous forme $\Join_2$-normale. Puis, nous cherchons le motif m dans cette équation. Ainsi, nous réécrivons W sous forme récursive.

Pour illustrer cette recherche d'auto-dépendance, nous reprenons l'exemple de l'introduction.

Exemple 4.25 *Rappelons qu'il fallait prouver que :*

$$S \vdash (d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1) \cap \mathcal{D} : Y.\delta_1 \vee Y.\delta_2 \downarrow tt$$

Nous avions introduit une variable W définie par $Y.\delta_1 \vee Y.\delta_2$. Supposons maintenant que Y soit définie par l'équation suivante (cf. figure 4.4) :

$$Y \quad = \quad Y.\delta_3 \vee Y.\delta_4$$

Nous substituons Y par sa définition dans W, et nous obtenons l'équation (cf. figure 4.5) :

$$W = Y.\delta_3 \circ \delta_1 \vee Y.\delta_4 \circ \delta_1 \vee Y.\delta_3 \circ \delta_2 \vee Y.\delta_4 \circ \delta_2$$

Nous appliquons alors sur W la recherche d'auto-dépendance avec le motif $Y.\delta_1 \vee Y.\delta_2$. Il y a six injections possibles, nous n'en considérerons ici que deux (les ensembles de solutions engendrés par les quatre autres étant vides).

$$\begin{array}{rll} \varphi_1 : & (Y,\delta_1) \mapsto (Y,\delta_3 \circ \delta_1) & \varphi_2 : \quad (Y,\delta_1) \mapsto (Y,\delta_4 \circ \delta_1) \\ & (Y,\delta_2) \mapsto (Y,\delta_3 \circ \delta_2) & \qquad\quad (Y,\delta_2) \mapsto (Y,\delta_4 \circ \delta_2) \end{array}$$

Nous avons donc deux systèmes à résoudre :

$$\left\{ \begin{array}{rcl} \delta_3 \circ \delta_1 & = & \delta_1 \circ d' \\ \delta_3 \circ \delta_2 & = & \delta_2 \circ d' \end{array} \right. \qquad \left\{ \begin{array}{rcl} \delta_4 \circ \delta_1 & = & \delta_1 \circ d'' \\ \delta_4 \circ \delta_2 & = & \delta_2 \circ d'' \end{array} \right.$$

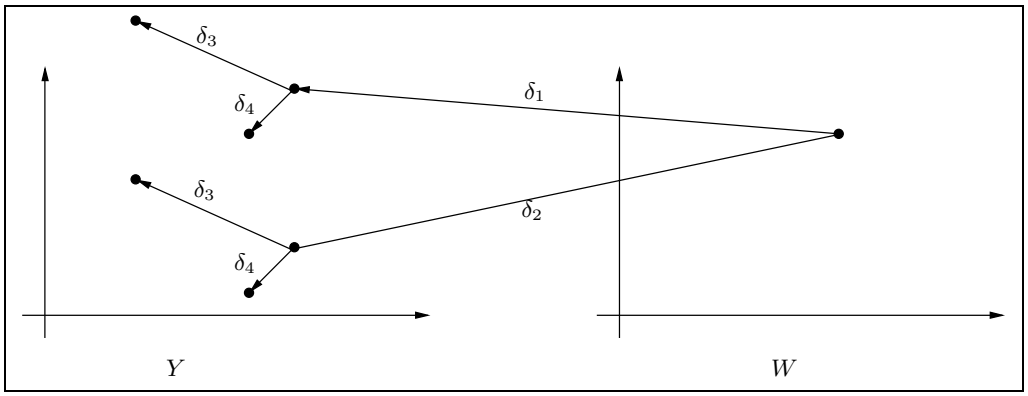


FIG. 4.4 – Variables W et Y et leurs dépendances.

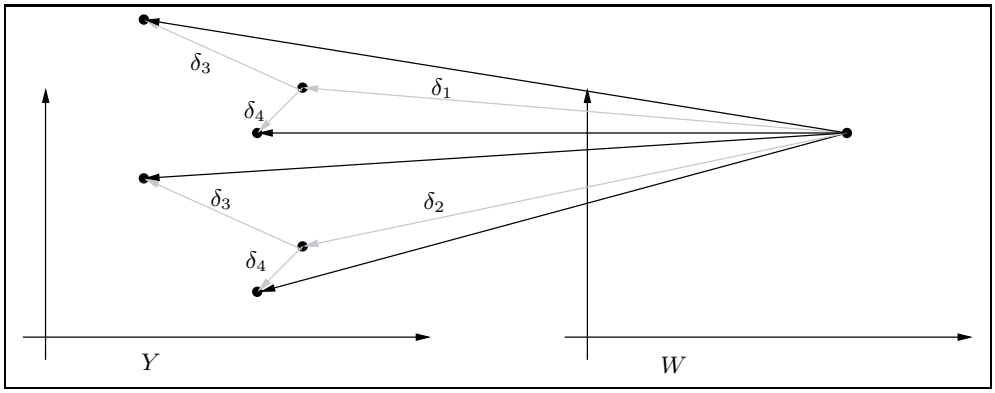


FIG. 4.5 – Variables W et Y et leurs dépendances.

Pour résoudre ces systèmes, comme nous considérons des dépendances uniformes, il suffit d'inverser la matrice, dans un cadre plus général, il faudrait appliquer la méthode de l'annexe C. Dans ce dernier cas, nous trouvons pour le premier système $d' = \delta_3$, et pour le second $d'' = \delta_4$. Ainsi, on a (cf. figure 4.6) :

$$W = \mathcal{D}_W : W.\delta_3 \vee W.\delta_4$$

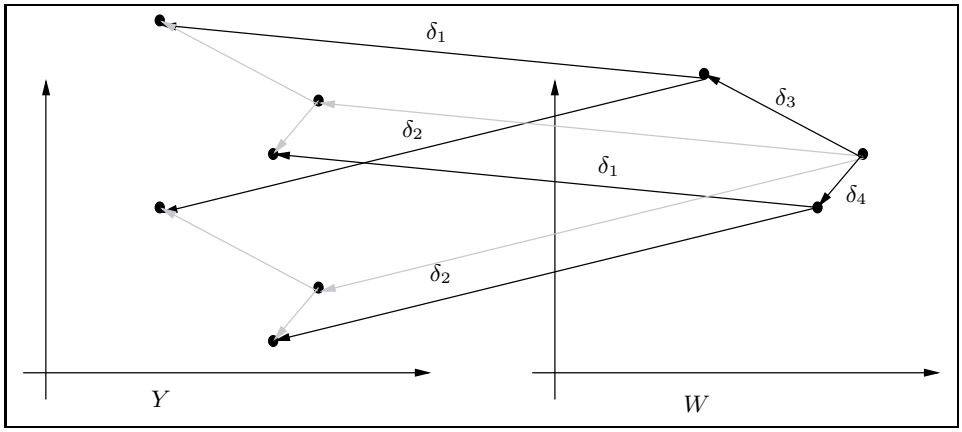


FIG. 4.6 – Nouvelle définition de W

Le problème est donc maintenant transformé en la recherche d'une preuve de :

$$S \vdash_{(d_1^{-1}(\mathcal{D})\cap\mathcal{D}_1)\cap\mathcal{D}} W.\delta_3 \vee W.\delta_4 \downarrow tt$$

Sur ce problème, nous pouvons maintenant appliquer la substitution par constante. Sur un certain sous domaine de $(d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1) \cap \mathcal{D}$, W sera définie par True, les autres sous domaines correspondront à des variables d'entrée.

Nous pouvons maintenant définir la nouvelle règle de preuve correspondant à la recherche d'auto-dépendance.

Sûreté Notons W_{rec} la variable W réécrite avec des auto-dépendances, notre nouvelle règle est donc :

$$\frac{S' \vdash \mathcal{D} : W_{rec} \downarrow v}{S \vdash \mathcal{D} : W \downarrow v} \text{ AUTO-DEP}$$

où S' est un sursystème de S contenant la variable W_{rec}.

Nous allons maintenant nous intéresser à la sûreté et à l'inversibilité de cette règle.

Proposition 4.26 *Cette règle de recherche d'auto-dépendance est sûre et inversible.*

Démonstration : *Il faut donc montrer que*

$$\models_\rho \mathcal{D} : W_{rec} \downarrow v \implies \models_\rho \mathcal{D} : W \downarrow v$$

par rapport au système S'. Nous allons en fait montrer l'équivalence. La première étape est de substituer les occurrences de X par leur définition dans la variable W. Nous savons que cette étape préserve la sémantique. Si l'on note W' la variable W ainsi réécrite, nous avons l'équivalence :

$$\models_\rho \mathcal{D} : W \downarrow v \iff \models_\rho \mathcal{D} : W' \downarrow v$$

Nous appliquons la recherche du motif défini par l'expression de W dans W' et nous obtenons la variable W_{rec}. En appliquant la proposition 4.26, on obtient l'équivalence suivante

$$\models \mathcal{D} : W' \downarrow v \iff \models \mathcal{D} : W_{rec} \downarrow v$$

et ainsi nous avons montré :

$$\models_\rho \mathcal{D} : W_{rec} \downarrow v \iff \models_\rho \mathcal{D} : W \downarrow v$$

La règle de recherche d'auto-dépendance est donc sûre et inversible. ❑

Maintenant que nous avons défini toutes les règles de notre système de preuve et que nous avons prouvé que ces règles étaient sûres, nous pouvons définir formellement la notion d'existence d'une preuve.

Définition 4.27 (Existence d'une preuve) *Une preuve de la formule $\mathcal{D} : e \downarrow v$ existe si et seulement s'il existe une séquence finie de formules étendues par une valeur $\alpha_1, \ldots, \alpha_n$ telle que $\alpha_n = \mathcal{D} : e \downarrow v$ et chaque α_i est soit un axiome soit peut être déduit à partir d'une formule précédente en utilisant une des règles de preuve.*

5 Conclusion

Dans ce chapitre, nous avons défini plusieurs règles de preuve basiques dans la section 2 et nous avons montré que ces règles étaient sûres. Dans la section 3, nous avons défini la règle de substitution par constante et dans la section 4, nous avons défini la règle de recherche d'auto-dépendance. Toutes nos règles sont sûres et même inversibles. Nous avons donc actuellement les règles de preuve suivantes :

- règles de décomposition des opérandes,
- règle de la négation,
- règle de la dépendance,
- règle de l'équation,
- règle de la séparation des branches,
- règle de la substitution par constante,
- règle de la simplification,
- règle de recherche d'auto-dépendance.

Nous allons voir dans les chapitres suivants comment construire des preuves à partir de ces règles.

Chapitre 5

Itérations

Dans le chapitre 4, nous avons introduit deux règles spécifiques au modèle polyédrique : la substitution par constante et la recherche d'auto-dépendance. La substitution par constante permet de substituer une variable par une constante, elle s'appuie sur un principe de récurrence. Le but de la recherche d'auto-dépendance est de transformer l'expression d'une équation sur laquelle aucune règle ne peut être appliquée en une équation récurrente. Nous allons étudier ici comment allier ces deux règles de preuve.

Soit W une variable polyédrique pour laquelle nous voulons rechercher des auto-dépendances, et soit v une valeur booléenne. Nous supposons que nous devons prouver :

$$\vdash \mathcal{D} : W \downarrow v$$

Pour la variable W, deux cas peuvent se produire : soit elle est définie par une équation dont le membre droit contient la variable W (une auto-dépendance), soit elle dépend seulement d'autres variables.

- Dans le premier cas, nous appliquons la substitution par constante. La nouvelle expression définissant W contient :
  - soit des constantes
  - soit des expressions polyédriques dépendant d'autres variables
  - soit des expressions polyédriques dépendant toujours de W ; ces instances de W n'ont pas été substituées car elles n'étaient pas dans le domaine de validité $\mathcal{D}$.

  Pour chaque branche qui n'est pas définie par une constante, nous ajoutons une nouvelle variable définie, sur le sous domaine de la branche considérée, par l'équation dont le membre droit est défini par l'expression correspondante. Ceci nous conduit donc au second cas.
- Dans le second cas, nous appliquons la recherche d'auto-dépendance. Si nous trouvons une auto-dépendance, la variable W est alors définie récursivement et nous nous retrouvons dans le premier cas. Sinon, nous devons recommencer une nouvelle recherche d'auto-dépendance.

Nous répétons alternativement les deux opérations. Nous nous arrêtons quand, après une substitution par constante, soit nous trouvons $\bar{v}$ dans une des branches (et dans ce cas, nous avons un contre-exemple) soit dans toutes les branches, nous avons une variable d'entrée ou une constante.

Le problème auquel nous allons nous intéresser dans cette section est le suivant :

> *Nous devons déterminer dans quels cas l'itération qui alterne substitution par constante et recherche d'auto-dépendance s'arrête.*

Nous effectuerons la recherche d'auto-dépendance seulement si l'itération est finie.

Nous supposons que la variable W est une variable qui a été ajoutée après une substitution par constante. Cette variable n'est donc pas définie par une auto-dépendance. Nous supposons que la seule règle de preuve qui puisse être appliquée à W est la règle de recherche d'auto-dépendance (dans les autres cas nous utilisons les règles qui peuvent être appliquées). Ainsi, la variable est définie par un motif (*cf.* page 67).

Dans les deux premières sections de ce chapitre nous supposerons que la variable W n'est définie que par des occurrences d'une seule variable :

$$W = \mathcal{D}_W : \bigbbwedge_i X.d_i$$

Nous supposerons aussi que la variable X n'est définie que par des auto-dépendances. Nous nous restreindrons de plus au cas où les dépendances sont uniformes, c'est-à-dire des translations sur les indices. Remarquons simplement que cette dernière restriction n'est pas une restriction trop forte car toute équation récurrente affine peut être réécrite en une équation récurrente uniforme en introduisant de nouvelles variables [67]. Nous évoquerons le cas des dépendances non uniformes dans la conclusion de ce chapitre.

Dans, la section 1, nous supposerons que la variable X est définie par une unique expression, et nous étudierons le problème de l'arrêt selon les opérateurs utilisés dans cette définition. Dans la section 2, nous élargirons nos hypothèses sur X en supposant que X est définie par une expression à plusieurs branches. Dans la section 3, nous verrons comment itérer si W et X sont définies par plusieurs variables, puis supposerons que W n'est définie que par la variable X, elle-même définie par plusieurs variables. Enfin, nous conclurons.

1 Variable X définie par une unique branche

Nous nous restreignons donc au cas le plus simple : la variable X est définie par une expression contenant simplement des occurrences de X et composée d'une seule branche et d'un seul opérateur.

$$X = \bigbbwedge_{i\ 2} X.d'_i \text{ où } \bigbbwedge_2 \text{ appartient à } \{\vee, \wedge\}$$

Nous supposons que tous les d_i sont distincts, puisque les cas d'égalité ont été supprimés par la simplification propositionnelle. Notre problème est donc de savoir s'il est possible d'itérer alternativement recherche d'auto-dépendance et substitution par constante.

Le théorème suivant permet de définir les cas d'itérations finies et infinies.

Théorème 5.1 *Soient W et X deux variables polyédriques. Supposons que nous voulions établir une preuve $\vdash \mathcal{D}_W : W \downarrow v$ où v est la valeur absorbante de $\bbwedge_1$ et $\mathcal{D}_W$ le domaine de W. Alors, si W et X sont définies par les équations suivantes :*

$$W = \mathcal{D}_W : \bigbbwedge_{i\ 1} X.d_i$$

$$X = \mathcal{D}_X : \bigbbwedge_{i\ 2} X.d'_i$$

Nous obtenons les résultats suivants :

- *si $\bbwedge_1 = \bbwedge_2$, l'itération alternant la génération d'auto-dépendances et la substitution par constante est finie,*
- *si $\bbwedge_1 \neq \bbwedge_2$, l'itération alternant la génération d'auto-dépendances et la substitution par constante est infinie.*

Dans la suite, nous supposerons que W est définie par une $\bbwedge_1$-expression de deux termes :

$$W = \mathcal{D}_X : X.d_1 \bbwedge_2 X.d_2$$

La généralisation à n termes sera immédiate.

1.1 Opérateurs $\mathbb{X}_1$ et $\mathbb{X}_2$ égaux

Supposons que X soit définie par une $\mathbb{X}_2$-expression de deux termes.

$$X = X.\delta_1 \mathbb{X}_2 X.\delta_2$$

Nous substituons X par sa définition dans l'équation de W et après distribution des opérateurs $\mathbb{X}_2$, nous obtenons :

$$W = \mathcal{D}_W : X.\delta_1 \circ d_1 \mathbb{X}_2 X.\delta_2 \circ d_1 \mathbb{X}_2 X.\delta_1 \circ d_2 \mathbb{X}_2 X.\delta_2 \circ d_2$$

La recherche d'auto-dépendances passe par la résolution de douze systèmes d'équations entre dépendances. Comme δ_1 et δ_2 ont un rôle symétrique, ces systèmes se regroupent en six types différents. Nous allons voir dans le lemme suivant quels sont les systèmes ayant des solutions. Remarquons d'abord que nous travaillons sur des dépendances uniformes, la loi de composition de deux dépendances, correspond simplement à une addition : les dépendances sont simplement des translations.

Lemme 5.2 *Soient $\delta, \delta', d_1, d_2$ des dépendances uniformes toutes distinctes. On considère les systèmes d'équations suivants d'inconnue f :*

$$\left\{ \begin{array}{rcl} \delta \circ d_1 & = & d_1 \circ f \\ \delta \circ d_2 & = & d_2 \circ f \end{array} \right. \quad (5.1) \qquad \left\{ \begin{array}{rcl} \delta \circ d_2 & = & d_1 \circ f \\ \delta' \circ d_1 & = & d_2 \circ f \end{array} \right. \quad (5.4)$$

$$\left\{ \begin{array}{rcl} \delta \circ d_1 & = & d_1 \circ f \\ \delta' \circ d_2 & = & d_2 \circ f \end{array} \right. \quad (5.2) \qquad \left\{ \begin{array}{rcl} \delta \circ d_1 & = & d_1 \circ f \\ \delta' \circ d_1 & = & d_2 \circ f \end{array} \right. \quad (5.5)$$

$$\left\{ \begin{array}{rcl} \delta \circ d_2 & = & d_1 \circ f \\ \delta \circ d_1 & = & d_2 \circ f \end{array} \right. \quad (5.3) \qquad \left\{ \begin{array}{rcl} \delta \circ d_2 & = & d_1 \circ f \\ \delta' \circ d_2 & = & d_2 \circ f \end{array} \right. \quad (5.6)$$

Les systèmes (5.2, 5.3, 5.5, 5.6) n'ont pas de solutions. Le système (5.1) a exactement une solution. Le système (5.4) a une solution si et seulement si $\delta/2 + d_2 = \delta'/2 + d_1$.

Démonstration : *Les dépendances sont uniformes, elles sont donc commutatives. Ainsi, les systèmes (5.2, 5.5, 5.6) n'ont trivialement pas de solutions. Pour le système (5.3), il existe une solution si et seulement si $d_1 = d_2$ ce qui n'est pas possible, le système (5.3) n'a donc pas de solution. Pour le système (5.1), la solution est triviale, c'est $f = \delta$. Quant au dernier système, on montre facilement que la seule solution est $f = (\delta + \delta')/2$ si et seulement si $\delta/2 + d_1 = \delta'/2 + d_1$ et $(\delta + \delta')/2 \in \mathbb{Z}^n$.* ❑

Ainsi, à l'aide du système (5.1) nous trouvons au moins deux auto-dépendances et W peut s'écrire sous la forme suivante :

$$W = \mathcal{D}_W : W.\delta_1 \mathbb{X}_2 W.\delta_2$$

En appliquant le principe de substitution, nous obtenons alors sur une partie du domaine étudié une constante. Le reste du domaine sera généralement vide, si ce n'est pas le cas, nous verrons dans les chapitres suivants ce qu'il est possible de faire. Dans le cas étudié ici, nous sommes donc sûrs d'arrêter l'itération. La généralisation à n termes est immédiate.

1.2 Opérateurs $\mathbb{X}_1$ et $\mathbb{X}_2$ distincts

En substituant X par sa définition dans l'équation de W, et en regroupant les termes obtenus sous forme $\mathbb{X}_2$-normale, nous obtenons l'équation suivante :

$$\begin{array}{rll} W = & & (X.\delta_1 \circ d_1 \mathbb{X}_1 X.\delta_1 \circ d_2) \\ & \mathbb{X}_2 & (X.\delta_2 \circ d_1 \mathbb{X}_1 X.\delta_2 \circ d_2) \\ & \mathbb{X}_2 & (X.\delta_1 \circ d_1 \mathbb{X}_1 X.\delta_2 \circ d_2) \\ & \mathbb{X}_2 & (X.\delta_2 \circ d_1 \mathbb{X}_1 X.\delta_1 \circ d_2) \end{array}$$

Nous notons $W_1,\ W_2,\ W_3,\ W_4$, les variables polyédriques définies par :

$$\begin{aligned} W_1 &= X.\delta_1 \circ d_1 \between_1 X.\delta_1 \circ d_2 \\ W_2 &= X.\delta_2 \circ d_1 \between_1 X.\delta_2 \circ d_2 \\ W_3 &= X.\delta_1 \circ d_1 \between_1 X.\delta_2 \circ d_2 \\ W_4 &= X.\delta_2 \circ d_1 \between_1 X.\delta_1 \circ d_2 \end{aligned}$$

Soit ρ un environnement sémantiquement correct. Prouver $\models_\rho \mathcal{D}_W : W \downarrow v$ équivaut à prouver $\models_\rho \mathcal{D}_W : W_1 \downarrow v$ et $\models_\rho \mathcal{D}_W : W_2 \downarrow v$ et $\models_\rho \mathcal{D}_W : W_3 \downarrow v$ et $\models_\rho \mathcal{D}_W : W_4 \downarrow v$. Nous devons donc trouver une auto-dépendance dans chaque terme de W. En appliquant le lemme 5.2, nous trouvons une auto-dépendance dans les deux premiers termes (W_1 et W_2). Dans les deux derniers termes, le problème est plus complexe : nous pouvons trouver une auto-dépendance dans certains cas (lemme 5.2, système (5.4)), mais si nous trouvons une auto-dépendance pour l'un des deux termes, l'autre n'aura pas d'auto-dépendance. Ce résultat sera démontré dans le lemme 5.3. Il restera donc dans tous les cas au moins un terme à simplifier. Nous commençons par une explication informelle. Supposons que ce terme correspond à la variable W_3, nous recommençons la recherche d'auto-dépendance dans cette variable, nous obtenons alors :

$$\begin{array}{rll} W_3 = & W_3.\delta_1 & \\ \between_2 & W_3.\delta_2 & \\ \between_2 & (X.\delta_1 \circ \delta_1 \circ d_1 \between_1 X.\delta_2 \circ \delta_2 \circ d_2) & \\ \between_2 & W.\delta_1 \circ \delta_2 & \end{array} \tag{5.7}$$

- S'il existe une auto-dépendances pour W_4, nous montrerons (lemme 5.3) qu'il n'y a pas d'auto-dépendance dans la troisième branche de W_3 (pour cette branche nous introduisons une variable nommée W_3^2 définie par $(X.\delta_1 \circ \delta_1 \circ d_1 \between_1 X.\delta_2 \circ \delta_2 \circ d_2)$), la recherche de motif échoue aussi pour les motifs définis par les variables W, W_3 et W_4.
- S'il n'y a pas d'auto-dépendance dans toutes les branches de l'équation de W_4, il faut ajouter une nouvelle variable W_4^2 et effectuer une nouvelle recherche dans W_4. Nous obtenons une équation similaire à l'équation (5.7) :
  - soit dans toutes les branches nous obtenons une auto-dépendance. Dans ce cas et comme précédemment nous montrerons (lemme 5.3) qu'il n'y a pas d'auto-dépendance dans la troisième branche et que la recherche de motif échoue pour les motifs définis par les autres variables,
  - soit il n'y a pas d'auto-dépendance et nous ajoutons une nouvelle variable W_4^3.

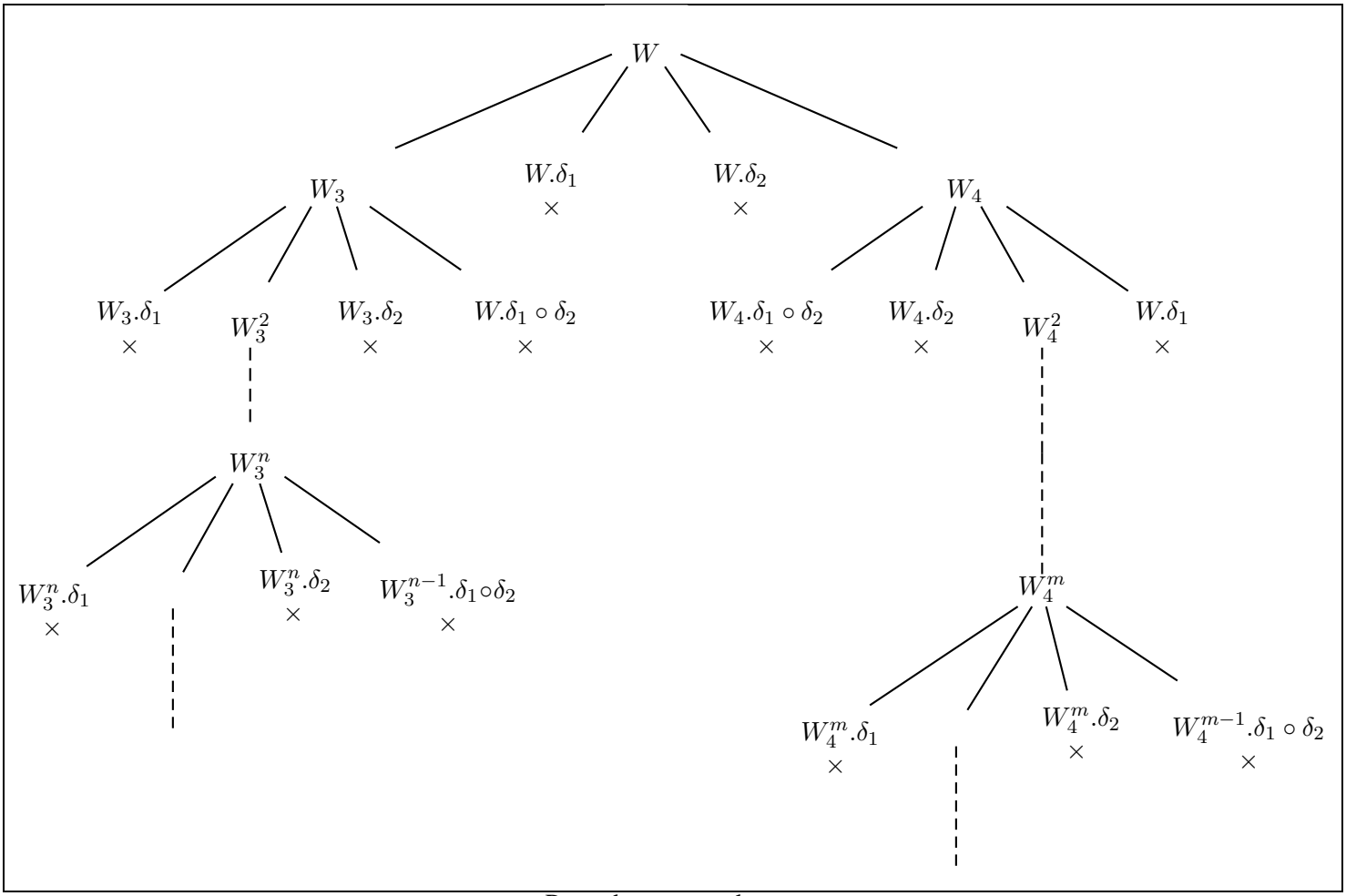
Nous répétons le processus en ajoutant des variables (*cf.* Fig. 5.1). Les variables W_3^n et W_4^m sont définies par les deux équations suivantes :

$$\begin{aligned} W_3^n &= X.\delta_1^n \circ d_1 \between_1 X.\delta_2^n \circ d_2 \\ W_4^m &= X.\delta_2^m \circ d_1 \between_1 X.\delta_1^m \circ d_2 \end{aligned}$$

Après substitution et application de la recherche de motif dans W_3^n et W_4^m, nous obtenons les deux équations suivantes :

$$\begin{array}{rll} W_3^n = & W_3^n.\delta_1 & \\ \between_2 & W_3^n.\delta_2 & \\ \between_2 & X.\delta_1^{n+1} \circ d_1 \between_1 X.\delta_2^{n+1} \circ d_2 & \\ \between_2 & W_3^{n-1}.\delta_1 \circ \delta_2 & \end{array} \qquad \begin{array}{rll} W_4^m = & W_4^m.\delta_1 & \\ \between_2 & W_4^m.\delta_2 & \\ \between_2 & X.\delta_2^{m+1} \circ d_1 \between_1 X.\delta_1^{m+1} \circ d_2 & \\ \between_2 & W_4^{m-1}.\delta_1 \circ \delta_2 & \end{array}$$

Ainsi, dans les deux cas, seule une branche pose problème et les variables rajoutées sont toutes de la forme W_3^i ou W_4^j avec $i \leq n$ et $j \leq m$. Nous remarquons que les deux branches sans auto-dépendance



Pour chaque nœud :

- soit nous avons une auto-dépendance et l'itération s'arrête ($\times$),
- soit nous ajoutons une nouvelle variable.

FIG. 5.1 – Arbre des générations des auto-dépendances

(celles où apparaît encore X) représentent les variables W_3^{n+1} et W_4^{m+1}. Nous allons maintenant chercher si ces deux variables peuvent s'écrire en fonction de W_3^i ou W_4^j. Pour chacune des deux variables, nous pouvons écrire quatre types de systèmes. Pour trouver une auto-dépendance à W, il faut donc trouver une solution à l'un des systèmes pour chacune des variables et vérifier que les deux solutions sont compatibles.

Lemme 5.3

Systèmes issus de W_3	*Systèmes issus de W_4*
$S_3^1 = \begin{cases} \delta_1^n \circ d_1 = \delta_1^i \circ d_1 \circ f \\ \delta_2^n \circ d_2 = \delta_2^i \circ d_2 \circ f \end{cases}$	$S_4^1 = \begin{cases} \delta_1^m \circ d_2 = \delta_1^j \circ d_2 \circ f \\ \delta_2^m \circ d_1 = \delta_2^j \circ d_1 \circ f \end{cases}$
$S_3^2 = \begin{cases} \delta_1^n \circ d_1 = \delta_2^i \circ d_2 \circ f \\ \delta_2^n \circ d_2 = \delta_1^i \circ d_1 \circ f \end{cases}$	$S_4^2 = \begin{cases} \delta_1^m \circ d_2 = \delta_2^j \circ d_1 \circ f \\ \delta_2^m \circ d_1 = \delta_1^j \circ d_2 \circ f \end{cases}$
$S_3^3 = \begin{cases} \delta_1^n \circ d_1 = \delta_1^j \circ d_2 \circ f \\ \delta_2^n \circ d_2 = \delta_2^j \circ d_1 \circ f \end{cases}$	$S_4^3 = \begin{cases} \delta_1^m \circ d_2 = \delta_1^i \circ d_1 \circ f \\ \delta_2^m \circ d_1 = \delta_2^i \circ d_2 \circ f \end{cases}$
$S_3^4 = \begin{cases} \delta_1^n \circ d_1 = \delta_2^j \circ d_1 \circ f \\ \delta_2^n \circ d_2 = \delta_1^j \circ d_2 \circ f \end{cases}$	$S_4^4 = \begin{cases} \delta_1^m \circ d_2 = \delta_2^i \circ d_2 \circ f \\ \delta_2^m \circ d_1 = \delta_1^i \circ d_1 \circ f \end{cases}$

- *Les systèmes ($S_3^1,\ S_4^1,\ S_3^4,\ S_4^4$) n'ont pas de solution.*
- *Si S_3^2 ou S_3^3 a une solution, alors ni S_4^2 ni S_4^3 n'ont de solution, et réciproquement.*

Démonstration :

- *Les systèmes ($S_3^1,\ S_4^1,\ S_3^4,\ S_4^4$) n'ont pas de solution (lemme 5.2).*
- *Supposons que S_3^2 a une solution, alors nous avons l'égalité suivante :*

$$\delta_1^{n+i} \circ d_1^2 = \delta_2^{n+i} \circ d_2^2$$

Nous allons montrer que cette égalité n'est pas compatible avec les systèmes S_4^2 et S_4^3.

- *Si S_4^2 a une solution, de la même manière nous obtenons l'égalité suivante :*

$$\delta_1^{m+j} \circ d_2^2 = \delta_2^{m+j} \circ d_1^2$$

En composant cette égalité avec δ_1^{n+i} et en substituant $\delta_1^{n+i} \circ d_1^2$ par $\delta_2^{n+i} \circ d_2^2$, nous obtenons alors l'égalité :

$$\delta_1^{m+j+n+i} = \delta_2^{m+j+n+i}$$

Ce qui implique $\delta_1 = \delta_2$. Or δ_2 et δ_1 sont différents (sinon l'expression définissant X serait uniquement définie par $X.\delta_1$).

- *Si S_4^3 a une solution nous obtenons de même que $\delta_1 = \delta_2$ ce qui est impossible.*
- *Nous supposons maintenant que S_3^3 a une solution, il suffit de montrer que ce n'est pas compatible avec S_4^3. La démonstration est similaire à ce qui précède.*

□

Nous avons donc montré qu'il était impossible de trouver une auto-dépendance dans chaque branche. Dans le cas où $\mathbb{X}_1$ est différent de $\mathbb{X}_2$, il n'est donc pas possible d'itérer le principe de substitution et la recherche d'auto-dépendance. La généralisation à n termes pour X est immédiate.

1.3 Le cas de la négation

Nous allons maintenant nous intéresser au cas ou dans l'équation de W la variable X est sous la portée d'une négation. Le théorème suivant permet de caractériser les cas où l'itération est finie et les cas où elle est infinie.

Théorème 5.4 *Soient W et X deux variables polyédriques. Si nous souhaitons établir la preuve $\vdash \mathcal{D}_W$: $W \downarrow v$ où v est la valeur absorbante de $\mathbb{X}_1$ et si W et X sont définies par les équations suivantes, où $l_1, l_2, l \in \mathbb{N}^*$ et $\forall i \in \{1, \ldots, l\}, m_i \in \mathbb{N}^*$:*

$$\begin{aligned} W &= (\bigtimes_{j=1}^{l_1}{}_{\mathbb{X}_1} X.d_j) \mathbb{X}_1 (\bigtimes_{j=1}^{l_2}{}_{\mathbb{X}_1} \neg X.d'_j) \\ X &= \bigtimes_{i=1}^{l}{}_{\mathbb{X}_2} \bigtimes_{j=1}^{m_i}{}_{\mathbb{X}_1} X.d^i_j, \ \textit{avec} \ \mathbb{X}_2 \neq \mathbb{X}_1 \end{aligned}$$

alors :

- *l'itération alternant la génération d'auto-dépendance et la substitution par constante est finie dans l'un des trois cas suivants :*
  - $l_1 = l_2 = 1$,
  $$W = X.d_1 \mathbb{X}_1 \neg X.d'_1 \ \textit{et} \ X = \bigtimes_{i=1}^{l}{}_{\mathbb{X}_2} \bigtimes_{j=1}^{m_i}{}_{\mathbb{X}_1} X.d^i_j$$
  - $l_1 = 1 \wedge l_2 > 1 \wedge \forall i, \ m_i = 1$,
  $$W = X.d_1 \mathbb{X}_1 (\bigtimes_{j=1}^{l_2}{}_{\mathbb{X}_1} \neg X.d'_j) \textit{ et } X = \bigtimes_{i=1}^{l}{}_{\mathbb{X}_2} X.d_i$$
  - $l_1 > 1 \wedge l_2 = 1 \wedge l = 1$,
  $$W = (\bigtimes_{j=1}^{l_1}{}_{\mathbb{X}_1} X.d_j) \mathbb{X}_1 \neg X.d'_1 \textit{ et } X = \bigtimes_{j=1}^{m}{}_{\mathbb{X}_1} X.d_j$$
- *dans tous les autres cas, l'itération est infinie.*

La démonstration formelle de ce théorème est donnée en annexe 2.1. Nous en donnerons ici une justification informelle.

Premier cas : $l_1 = l_2 = 1$. Supposons les variables W et X définies par les équations suivantes :

$$\begin{aligned} W &= X.d_1 \mathbb{X}_1 \neg X.d_2 \\ X &= (X.\delta_1 \mathbb{X}_1 X.\delta_2) \mathbb{X}_2 X.\delta_3 \end{aligned}$$

Après substitution de X par sa définition et mise sous forme $\mathbb{X}_2$-normale, nous obtenons :

$$\begin{aligned} W = \quad & (X.\delta_1 \circ d_1 \mathbb{X}_1 X.\delta_2 \circ d_1 \mathbb{X}_1 \neg X.\delta_1 \circ d_2 \mathbb{X}_1 \neg X.\delta_3 \circ d_2) \\ \mathbb{X}_2 \quad & (X.\delta_1 \circ d_1 \mathbb{X}_1 X.\delta_2 \circ d_1 \mathbb{X}_1 \neg X.\delta_2 \circ d_2 \mathbb{X}_1 \neg X.\delta_3 \circ d_2) \\ \mathbb{X}_2 \quad & (X.\delta_3 \circ d_1 \mathbb{X}_1 \neg X.\delta_1 \circ d_2 \mathbb{X}_1 \neg X.\delta_3 \circ d_2) \\ \mathbb{X}_2 \quad & (X.\delta_3 \circ d_1 \mathbb{X}_1 \neg X.\delta_2 \circ d_2 \mathbb{X}_1 \neg X.\delta_3 \circ d_2) \end{aligned}$$

La recherche d'auto-dépendance donne alors :

$$\begin{aligned} W = \quad & (W.\delta_1 \mathbb{X}_1 X.\delta_2 \circ d_1 \mathbb{X}_1 \neg X.\delta_3 \circ d_2) \\ \mathbb{X}_2 \quad & (X.\delta_1 \circ d_1 \mathbb{X}_1 W.\delta_2 \mathbb{X}_1 \neg X.\delta_3 \circ d_2) \\ \mathbb{X}_2 \quad & (W.\delta_3 \mathbb{X}_1 \neg X.\delta_1 \circ d_2) \\ \mathbb{X}_2 \quad & (W.\delta_3 \mathbb{X}_1 \neg X.\delta_2 \circ d_2) \end{aligned}$$

Nous avons trouvé une auto-dépendance dans chaque sous-terme, l'itération s'arrête donc.

Deuxième cas : $l_1 = 1 \wedge l_2 > 1 \wedge \forall i,\ m_i = 1$ **ou** $l_1 > 1 \wedge l_2 = 1 \wedge l = 1$**.** La justification est similaire à la précédente et nous nous contenterons de donner un exemple pour le deuxième cas.

$$\begin{aligned} W &= X.d_1 \between_1 X.d_2 \between_1 \neg X.d_3 \\ X &= X.\delta_1 \between_1 X.\delta_2 \end{aligned}$$

Après substitution de X par sa définition et mise sous forme $\between_2$-normale, nous obtenons :

$$\begin{aligned} W = \quad & X.\delta_1 \circ d_1 \between_1 X.\delta_1 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \\ \between_2 \quad & X.\delta_2 \circ d_1 \between_1 X.\delta_2 \circ d_2 \between_1 \neg X.\delta_2 \circ d_3 \end{aligned}$$

La recherche d'auto-dépendance nous donne alors :

$$W = W.\delta_1 \between_2 W.\delta_2$$

Troisième cas : une itération infinie. La justification est là encore similaire à la justification précédente. L'échec de l'itération est une conséquence du lemme 5.3. Nous donnerons juste un exemple pour illustrer l'utilisation de ce lemme. Nous reprenons l'exemple du cas précédent en remplaçant $\between_1$ par $\between_2$ dans l'équation définissant X :

$$\begin{aligned} W &= X.d_1 \between_1 X.d_2 \between_1 \neg X.d_3 \\ X &= X.\delta_1 \between_2 X.\delta_2 \end{aligned}$$

Après substitution de X par sa définition et mise sous forme $\between_2$-normale, nous obtenons :

$$\begin{aligned} W = \quad & X.\delta_1 \circ d_1 \between_1 X.\delta_1 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3 \\ \between_2 \quad & X.\delta_1 \circ d_1 \between_1 X.\delta_2 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3 \\ \between_2 \quad & X.\delta_2 \circ d_1 \between_1 X.\delta_1 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3 \\ \between_2 \quad & X.\delta_2 \circ d_1 \between_1 X.\delta_2 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3 \end{aligned}$$

Dans le premier et le dernier terme nous trouvons une auto-dépendance et nous obtenons :

$$\begin{aligned} W = \quad & W.\delta_1 \between_1 \neg X.\delta_2 \circ d_3 \\ \between_2 \quad & X.\delta_1 \circ d_1 \between_1 X.\delta_2 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3 \\ \between_2 \quad & X.\delta_2 \circ d_1 \between_1 X.\delta_1 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3 \\ \between_2 \quad & W.\delta_2 \between_1 \neg X.\delta_1 \circ d_3 \end{aligned}$$

En revanche, dans le deuxième et le troisième termes, nous ne trouvons pas d'auto-dépendance. Ainsi, pour la sous-expression $X.\delta_1 \circ d_1 \between_1 X.\delta_2 \circ d_2 \between_1 \neg X.\delta_1 \circ d_3 \between_1 \neg X.\delta_2 \circ d_3$, nous devons résoudre les mêmes types de système que pour la variable W_3 de la sous section 1.2 de ce chapitre. Ainsi, l'itération est infinie.

2 Cas des branches multiples

Nous allons nous intéresser au cas où X est définie par plusieurs branches. Les domaines de chaque branche vont donc cette fois-ci avoir de l'importance. Nous supposons que toutes les dépendances sont uniformes. On note $\mathcal{D}_j$ les domaines des branches de X. Soient W et X deux variables définies par les

équations suivantes :

$$W = D_W : \underset{i}{\bowtie_1} X.d_i$$

$$X = \begin{cases} \vdots \\ \mathcal{D}_j : \underset{i=1 i'=1}{\overset{l_j}{\bowtie_1}\overset{l_i^j}{\bowtie_2}} X.\delta_{i'}^j \\ \vdots \end{cases}$$

Le résultat que nous voulons prouver est le suivant :

> *Si pour tout j, pour tout $i \leq l_j$, $l_i^j = 1$ alors sous certaines conditions (que nous préciserons) l'itération de la recherche d'auto-dépendance, et de la substitution par constante est finie. S'il existe un j tel que $l_i^j \neq 1$, cette itération est infinie.*

Nous allons tout d'abord nous restreindre au cas où X est définie récursivement par une expression conditionnelle à deux branches sans occurrence d'autres variables, où $l_1 = l_2 = 1$ et où W est définie par deux sous-termes. Nous verrons ensuite comment généraliser le résultat. Nous supposons donc que W et X sont définies de la manière suivante :

$$W = D_W : X.d_1 \bowtie_1 X.d_2$$
$$X = \begin{cases} \mathcal{D}_1 : & X.\delta_1 \\ \mathcal{D}_2 : & X.\delta_2 \end{cases}$$

Nous avons représenté les deux variables et leurs dépendances dans la figure 5.2.

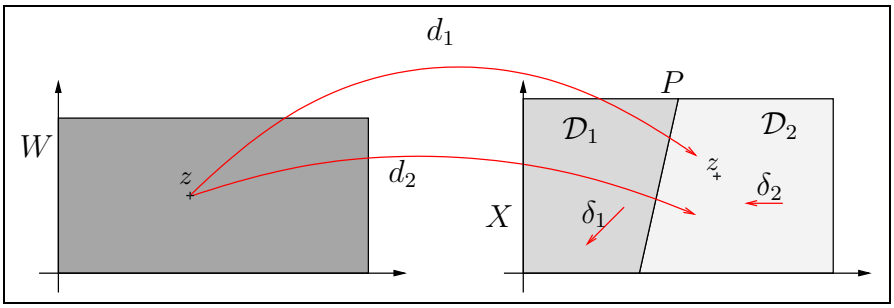


FIG. 5.2 – Variables W et X

Nous commencerons par donner une explication intuitive du résultat. Puis, nous le démontrerons formellement.

2.1 Explication intuitive

Nous faisons une recherche d'auto-dépendance dans W, nous effectuons donc tout d'abord la substitution de X et nous obtenons :

$$W = \begin{cases} \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_1) & : X.\delta_1 \circ d_1 \bowtie_1 X.\delta_1 \circ d_2 \\ \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_2) \cap d_2^{-1}(\mathcal{D}_2) & : X.\delta_2 \circ d_1 \bowtie_1 X.\delta_2 \circ d_2 \\ \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) & : X.\delta_1 \circ d_1 \bowtie_1 X.\delta_2 \circ d_2 \\ \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_2) \cap d_2^{-1}(\mathcal{D}_1) & : X.\delta_2 \circ d_1 \bowtie_1 X.\delta_1 \circ d_2 \end{cases}$$

Puisque les dépendances sont uniformes, nous appliquons le lemme 5.3. Ainsi, nous trouvons une auto-dépendance dans les deux premières branches.

$$W = \begin{cases} \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_1) & : W.\delta_1 \\ \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_2) \cap d_2^{-1}(\mathcal{D}_2) & : W.\delta_2 \\ \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) & : X.\delta_1 \circ d_1 \mathbb{X}_1 X.\delta_2 \circ d_2 \\ \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_2) \cap d_2^{-1}(\mathcal{D}_1) & : X.\delta_2 \circ d_1 \mathbb{X}_1 X.\delta_1 \circ d_2 \end{cases}$$

Dans les deux dernières branches, il n'y a pas d'auto-dépendance. Nous pouvons alors ajouter des variables : nous obtiendrons une solution si et seulement si nous réussissons à prouver qu'après un nombre fini d'itérations, nous avons une auto-dépendance dans chaque branche. Cependant, comme dans le cas précédent où $\mathbb{X}_1 \neq \mathbb{X}_2$ (*cf.* théorème 5.1), nous verrons que nous ne pouvons pas trouver pas d'auto-dépendance dans chaque branche. La solution va ici passer par l'étude des domaines des variables ajoutées. Nous allons prouver que les domaines de certaines branches deviennent vides après un certain nombre d'itérations. Ainsi, il ne sera pas nécessaire de trouver une auto-dépendance dans ces branches.

L'idée est de projeter les dépendances et les domaines sur un vecteur $\vec{\omega}$ orthogonal à l'hyperplan P séparant les domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ (*cf.* figure 5.2). Si nous nous plaçons dans $\mathbb{R}^n$, un domaine est non vide si et seulement si toutes les projections de ce domaine sur ses différents indices sont non vides. La préimage de $\mathcal{D}_1$ par une des dépendances correspond simplement à une translation du domaine $\mathcal{D}_1$:

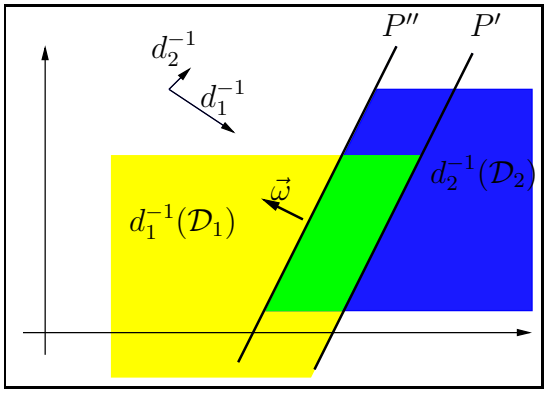


FIG. 5.3 – Intersection des préimages $d_1^{-1}(\mathcal{D}_1)$ et $d_2^{-1}(\mathcal{D}_2)$.

l'hyperplan P délimitant $\mathcal{D}_1$ est translaté et sa préimage P' délimite aussi la préimage de $\mathcal{D}_1$, de même pour $\mathcal{D}_2$ qui est délimité par P''. Ainsi, l'intersection des préimages de $\mathcal{D}_1$ et $\mathcal{D}_2$ est aussi délimitée par les translations P' et P'' de l'hyperplan P (*cf.* figure 5.3). La projection de l'intersection sur les vecteurs engendrant P sera donc non nulle. Pour prouver que l'intersection est nulle, il faut donc prouver que, sur la dimension de l'espace correspondant au vecteur $\vec{\omega}$ orthogonal à l'hyperplan P, la projection de l'intersection est nulle.

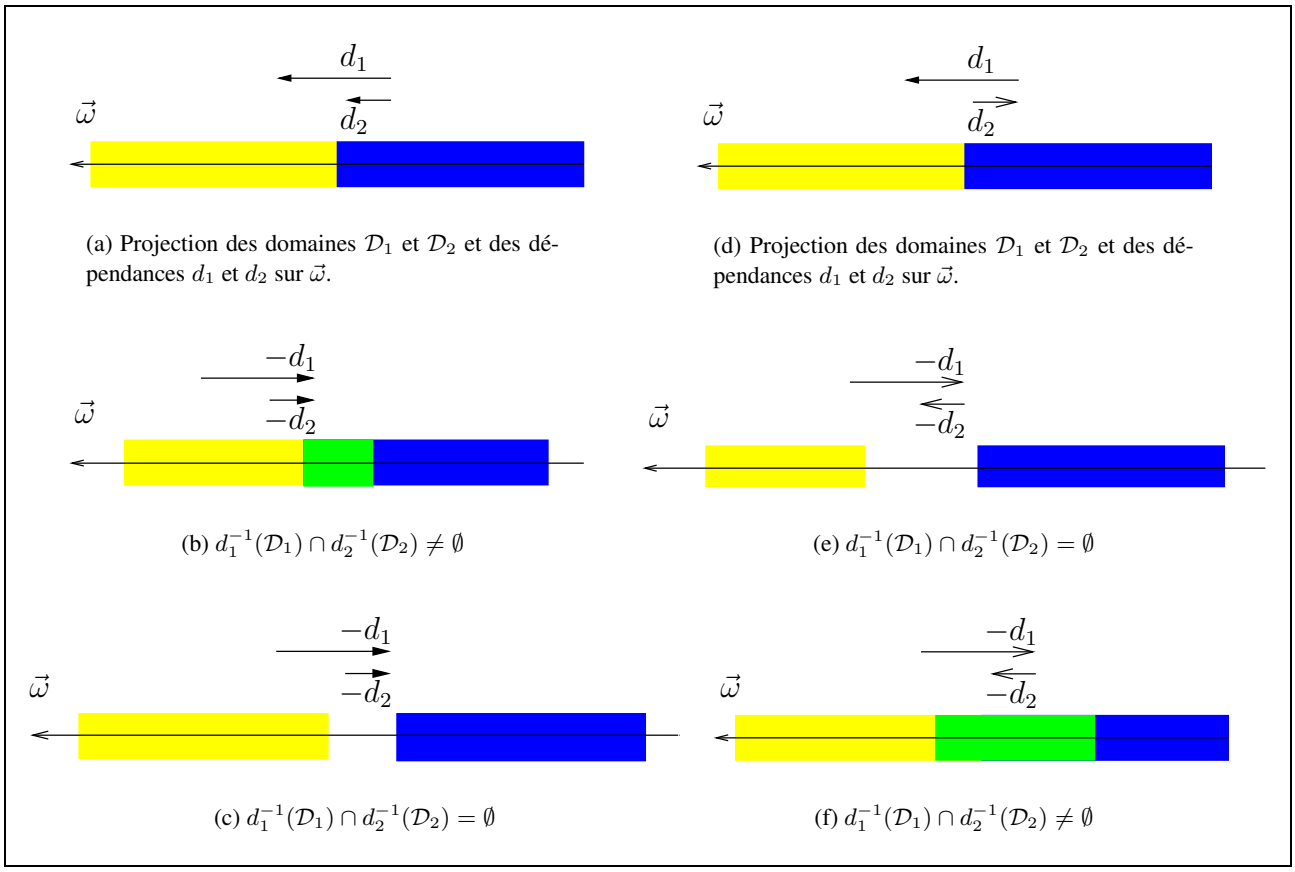
Nous allons donc projeter les domaines des deux dernières branches sur $\vec{\omega}$. La projection dépend simplement des projections de d_1 et d_2 sur $\vec{\omega}$, que nous noterons respectivement α_1 et α_2. On remarque ainsi (*cf.* figure 5.4) que selon leur valeur une des deux intersections sera vide.

Comme nous effectuons simplement des translations, l'une des intersections est vide et l'autre non. Les cas $\alpha_2 < \alpha_1$ et $\alpha_1 < \alpha_2$ sont symétriques

Nous ajoutons donc une variable pour la branche dont le domaine est non vide.

$$Z_1 = \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) : X.\delta_1 \circ d_1 \mathbb{X}_1 X.\delta_2 \circ d_2 \tag{5.8}$$

Nous répétons le processus pour la variable Z_1. Nous substituons X par sa définition et effectuons une



(a) Projection des domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ et des dépendances d_1 et d_2 sur $\vec{\omega}$.

(b) $d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) \neq \emptyset$

(c) $d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) = \emptyset$

(d) Projection des domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ et des dépendances d_1 et d_2 sur $\vec{\omega}$.

(e) $d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) = \emptyset$

(f) $d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) \neq \emptyset$

FIG. 5.4 – Projection des domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ sur $\vec{\omega}$. Calcul de leur préimages dans le cas $\alpha_2 < \alpha_1$.

recherche d'auto-dépendance, nous obtenons alors :

$$Z_1 = \begin{cases} \mathcal{D}_{Z_1} \cap d_1^{-1} \circ \delta_1^{-1}(\mathcal{D}_1) \cap d_2^{-1} \circ \delta_2^{-1}(\mathcal{D}_1) & : Z_1.\delta_1 \\ \mathcal{D}_{Z_1} \cap d_1^{-1} \circ \delta_1^{-1}(\mathcal{D}_2) \cap d_2^{-1} \circ \delta_2^{-1}(\mathcal{D}_2) & : Z_1.\delta_2 \\ \mathcal{D}_{Z_1} \cap d_1^{-1} \circ \delta_1^{-1}(\mathcal{D}_1) \cap d_2^{-1} \circ \delta_2^{-1}(\mathcal{D}_2) & : X.\delta_1 \circ \delta_1 \circ d_1 \Join_1 X.\delta_2 \circ \delta_2 \circ d_2 \\ \mathcal{D}_{Z_1} \cap d_1^{-1} \circ \delta_1^{-1}(\mathcal{D}_2) \cap d_2^{-1} \circ \delta_2^{-1}(\mathcal{D}_1) & : W.\delta_2 \circ \delta_1 \end{cases}$$

Nous remarquons que dans l'une des branches nous n'avons pas trouvé d'auto-dépendance. De même que précédemment on peut montrer que l'un des deux domaines est vide. Si c'est le domaine $\mathcal{D}_{Z_1} \cap d_1^{-1} \circ \delta_1^{-1}(\mathcal{D}_1) \cap d_2^{-1} \circ \delta_2^{-1}(\mathcal{D}_2)$ où nous n'avons pas trouvé d'auto-dépendance, l'itération s'arrête. Mais, si le domaine vide est le domaine $\mathcal{D}_{Z_1} \cap d_1^{-1} \circ \delta_1^{-1}(\mathcal{D}_2) \cap d_2^{-1} \circ \delta_2^{-1}(\mathcal{D}_1)$, nous devons alors ajouter une nouvelle variable et recommencer la recherche d'auto-dépendance. Il faut s'assurer que le processus est fini. Soient q un entier et Z_q, la variable ajoutée à l'étape q. Cette variable est définie sur le domaine $\mathcal{D}_{Z_q} = \bigcap_{i=0}^{q-1} d_1^{-1} \circ {\delta_1^{-1}}^i(\mathcal{D}_1) \cap d_2^{-1} \circ {\delta_2^{-1}}^i(\mathcal{D}_2)$ par $Z_q = \mathcal{D}_{Z_q} : X.{\delta_1}^q \circ d_1 \Join_1 X.{\delta_2}^q \circ d_2$. En substituant X par sa définition, nous obtenons :

$$Z_q = \begin{cases} \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_1) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_1) & : X.\delta_1 \circ {\delta_1}^q \circ d_1 \Join_1 X.\delta_1 \circ {\delta_2}^q \circ d_2 \\ \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_2) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_2) & : X.\delta_2 \circ {\delta_1}^q \circ d_1 \Join_1 X.\delta_2 \circ {\delta_2}^q \circ d_2 \\ \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_1) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_2) & : X.\delta_1 \circ \delta_1^q \circ d_1 \Join_1 X.\delta_2 \circ {\delta_2}^q \circ d_2 \\ \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_2) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_1) & : X.\delta_2 \circ {\delta_1}^q \circ d_1 \Join_1 X.\delta_1 \circ {\delta_2}^q \circ d_2 \end{cases}$$

La recherche d'auto-dépendance nous donne alors :

$$Z_q = \begin{cases} \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_1) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_1) & : Z_q.\delta_1 \\ \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_2) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_2) & : Z_q.\delta_2 \\ \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_1) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_2) & : X.{\delta_1}^{q+1} \circ d_1 \Join_1 X.{\delta_2}^{q+1} \circ d_2 \\ \mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_2) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_1) & : Z_{q-1}.\delta_1 \circ \delta_2 \end{cases}$$

Nous avons représenté l'ensemble des variables ajoutées dans la figure 5.5. Ainsi à chaque étape, nous trouvons dans trois des branches une auto-dépendance alors que dans la dernière il est impossible d'en trouver un. Pour que le processus termine, il faut donc montrer qu'il existe un q tel que $\mathcal{D}_{Z_q} \cap d_1^{-1} \circ (\delta_1^{-1})^q(\mathcal{D}_1) \cap d_2^{-1} \circ (\delta_2^{-1})^q(\mathcal{D}_2)$ est vide. L'idée est, là encore, de projeter l'intersection sur le vecteur orthogonal $\vec{\omega}$.

À chaque étape, les préimages de $\mathcal{D}_1$ et $\mathcal{D}_2$ par respectivement $d_1^{-1} \circ (\delta_1^{-1})^q$ et $d_2^{-1} \circ (\delta_2^{-1})^q$ sont projetées sur $\vec{\omega}$ et leur intersection est calculée. On calcule de plus l'intersection avec le domaine de la variable Z_q. Nous illustrons les différentes étapes de calcul dans la figure 5.6 : l'intersection est vide pour $q = 2$. Intuitivement, nous constatons sur cette figure, et pour cet exemple, qu'à chaque étape :

- l'intersection des deux préimages de $\mathcal{D}_1$ et $\mathcal{D}_2$ est décalée respectivement par les deux dépendances δ_1 et δ_2,
- l'intersection des deux préimages est plus petite que l'intersection précédente.

Ainsi, et comme nous travaillons avec des dépendances entières (leur projection est dans l'ensemble des rationnels) si l'un de ces critères est vérifié, au bout d'un certain nombre d'étapes l'intersection sera vide. Cependant, les dépendances δ_1 et δ_2 ne vérifient pas toujours ces critères. Si la projection de δ_1 sur $\vec{\omega}$ est négative et celle de δ_2 positive, alors l'intersection n'est jamais vide. La figure 5.7 illustre le cas où la projection de δ_1 sur $\vec{\omega}$ est négative et celle de δ_2 positive. Nous reviendrons par la suite sur ce cas où l'itération est infinie.

2.2 Preuve formelle

Nous allons maintenant passer à la preuve formelle de ces résultats. Le théorème que nous voulons prouver est le suivant.

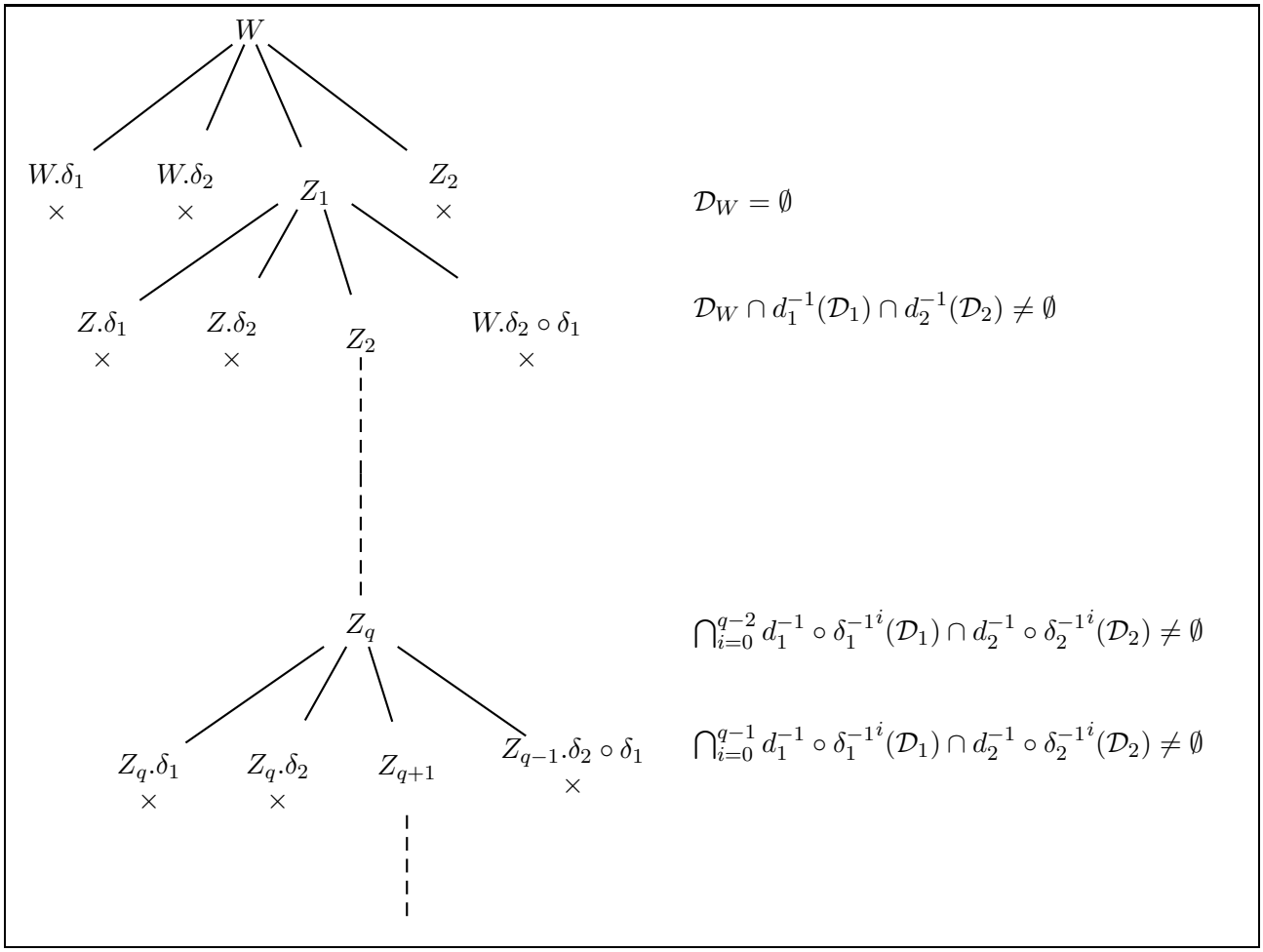


FIG. 5.5 – Arbre des variables ajoutées dans le cas de branches multiples

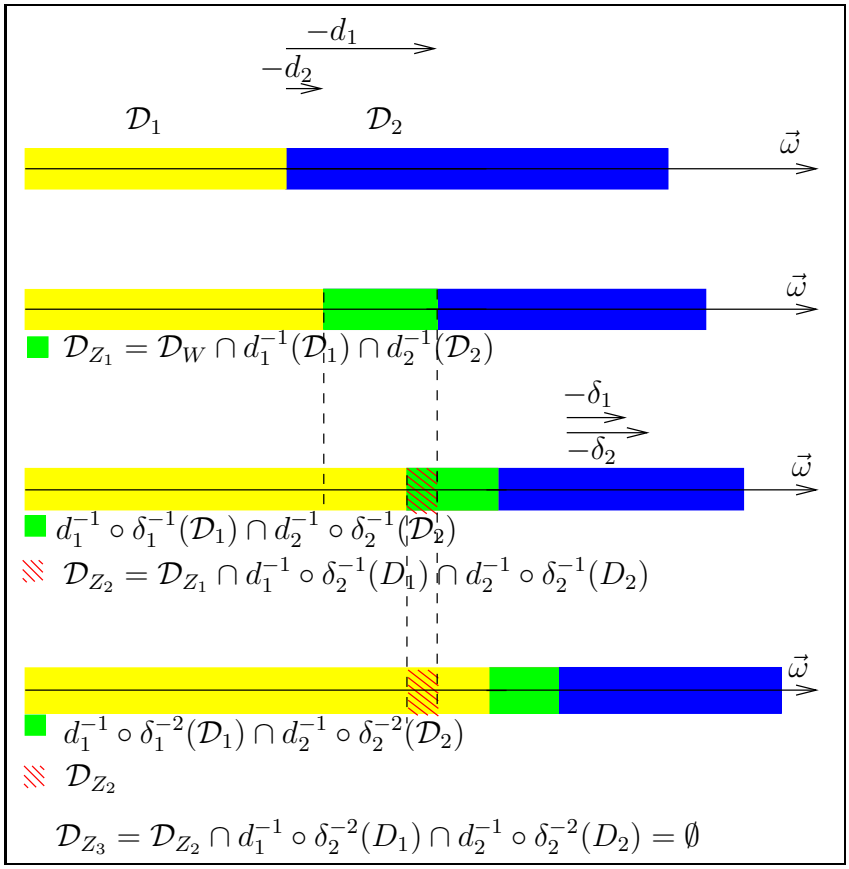


FIG. 5.6 – Intersection des préimages des domaines, projections de δ_1 et δ_2 négatives.

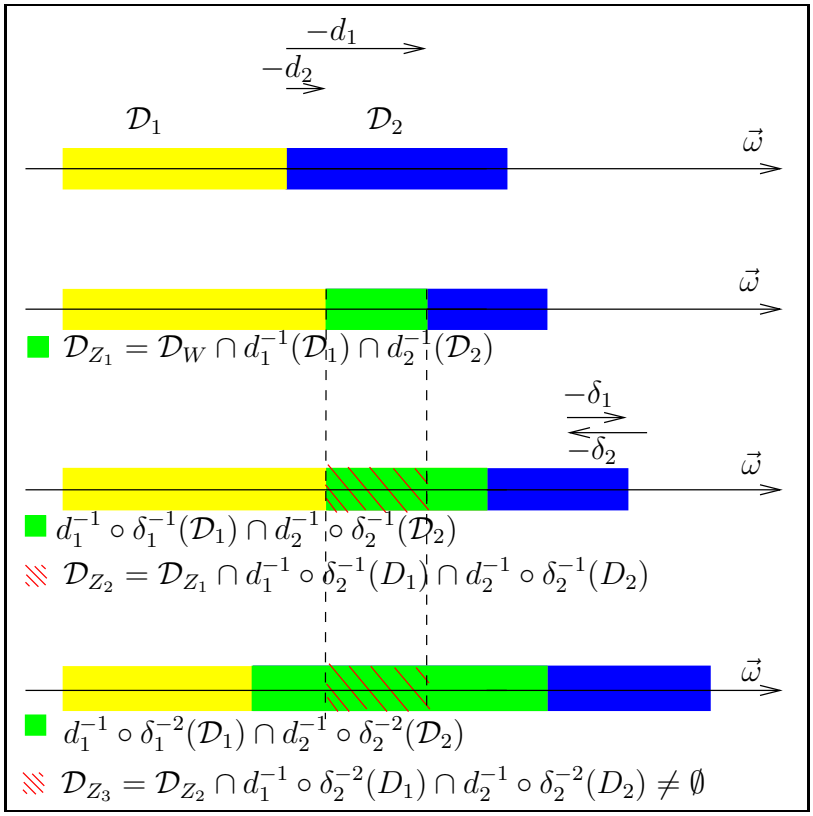


FIG. 5.7 – Intersection des préimages des domaines, projection de δ_2 positive.

Théorème 5.5 *Soient W et X deux variables polyédriques définies par les équations suivantes :*

$$\begin{array}{rcl} W & = & D_W : X.d_i \bowtie_1 X.d_2 \\ X & = & \left\{ \begin{array}{ll} \mathcal{D}_1 : & X.\delta_1 \\ \mathcal{D}_2 : & X.\delta_2 \end{array} \right. \end{array}$$

L'itération de la recherche d'auto-dépendance et de la substitution par constante est finie si $\beta_2 < 0$ ou $\beta_1 > 0$, où β_1 (resp. β_2) est la projection de δ_1 (resp. δ_2) sur un vecteur $\vec{\omega}$ orthogonal à un hyperplan P séparant les domaines $\mathcal{D}_1$ et $\mathcal{D}_2$.

La preuve de ce théorème est séparée en plusieurs propositions. Nous commençons par travailler dans $\mathbb{R}^p$. Soit P un hyperplan de dimension $p-1$ et $\vec{\omega}$ un vecteur orthogonal à cet hyperplan. L'hyperplan est engendré par la famille libre de vecteurs $\vec{\omega}_1, \ldots, \vec{\omega}_{p-1}$. La famille de vecteurs $\vec{\omega}_1, \ldots, \vec{\omega}_{p-1}, \vec{\omega}$ forme une base de $\mathbb{R}^p$. Nous notons P_1 et P_2 les demi-espaces respectivement fermé et ouvert délimités par l'hyperplan P tels que $P_1 = \{x \mid x = y + \alpha\vec{\omega}, \alpha \leq 0, y \in P\}$ et $P_2 = \{x \mid x = y + \alpha\vec{\omega}, \alpha > 0, y \in P\}$.

Les dépendances $d_1, d_2, \delta_1, \delta_2$ sont uniformes : elles correspondent à des translations. Elles sont définies comme combinaisons linéaires des vecteurs $\vec{\omega}_1, \ldots, \vec{\omega}$. On pose donc :

$$\begin{array}{rcl} d_1 & = & \sum_{i=1}^{p-1} \alpha_{1,i}\vec{\omega}_i + \alpha_1\vec{\omega} \\ d_2 & = & \sum_{i=1}^{p-1} \alpha_{2,i}\vec{\omega}_i + \alpha_2\vec{\omega} \\ \delta_1 & = & \sum_{i=1}^{p-1} \beta_{1,i}\vec{\omega}_i + \beta_1\vec{\omega} \\ \delta_2 & = & \sum_{i=1}^{p-1} \beta_{2,i}\vec{\omega}_i + \beta_2\vec{\omega} \end{array}$$

La première proposition permet de caractériser quand l'intersection devient vide dans $\mathbb{R}^p$.

Proposition 5.6 *Avec les notations précédentes, on a les deux équivalences suivantes :*

$$\forall q \in \mathbb{N},\ \bigcap_{i=1}^{q} d_1^{-1}(\delta_1^{-1})^i(P_1) \cap d_2^{-1}(\delta_2^{-1})^i(P_2) \neq \emptyset \text{ si et seulement si } \alpha_2 - \alpha_1 > 0 \wedge \beta_1 \leq 0 \wedge \beta_2 \geq 0$$

$$\textit{et} \quad \forall q \in \mathbb{N},\ \bigcap_{i=1}^{q} d_2^{-1}(\delta_1^{-1})^i(P_1) \cap d_1^{-1}(\delta_2^{-1})^i(P_2) \neq \emptyset \text{ si et seulement si } \alpha_1 - \alpha_2 > 0 \wedge \beta_1 \leq 0 \wedge \beta_2 \geq 0$$

La démonstration est située en annexe A section 2.2.

La proposition ci-dessous nous permet de justifier que l'on peut se limiter à l'étude d'une seule branche, le domaine de l'autre étant forcément vide.

Corollaire 5.7 *Avec les notations précédentes, on a*

$$d_1^{-1}(P_1) \cap d_2^{-1}(P_2) \neq \emptyset \ \textit{ssi}\ \ \alpha_2 - \alpha_1 > 0$$
$$\textit{et} \quad d_2^{-1}(P_1) \cap d_1^{-1}(P_2) \neq \emptyset \ \textit{ssi}\ \ \alpha_1 - \alpha_2 > 0$$

Démonstration : *Il suffit d'appliquer la proposition précédente pour* $q = 0$. ❑

Nous revenons maintenant aux domaines (dans $\mathbb{Z}^p$). Nous savons que les domaines $\mathcal{D}_1$ et $\mathcal{D}_2$ ont une intersection nulle, il existe donc un hyperplan P séparant $\mathcal{D}_1$ et $\mathcal{D}_2$, on note P_1 le demi-plan contenant $\mathcal{D}_1$ et P_2 le demi-plan contenant $\mathcal{D}_2$. Nous obtenons le corollaire suivant.

Corollaire 5.8

$$d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) \neq \emptyset \Longrightarrow d_1^{-1}(\mathcal{D}_2) \cap d_2^{-1}(\mathcal{D}_1) = \emptyset$$

$$d_1^{-1}(\mathcal{D}_2) \cap d_2^{-1}(\mathcal{D}_1) \neq \emptyset \Longrightarrow d_1^{-1}(\mathcal{D}_1) \cap d_2^{-1}(\mathcal{D}_2) = \emptyset$$

Ainsi, seules les valeurs β_2 et β_1 influent sur l'itération et nous avons montré le théorème.

Nous avons fourni une caractérisation pour le problème de l'arrêt de l'itération. Remarquons que le cas où l'itération ne s'arrête pas n'est pas un cas réaliste. En effet, cela signifie que les instances de X du domaine $\mathcal{D}_1$ ne dépendent que d'instances de X du domaine $\mathcal{D}_1$, et que les instances de X du domaine $\mathcal{D}_2$ ne dépendent que d'instances de X du domaine $\mathcal{D}_2$ (*cf.* figure 5.8). Il n'y a donc pas d'interactions entres les instances des deux domaines et la variable X pourrait être remplacée par deux variables totalement disjointes. Dans des systèmes concrets, nous ne rencontrerons jamais de telles situations.

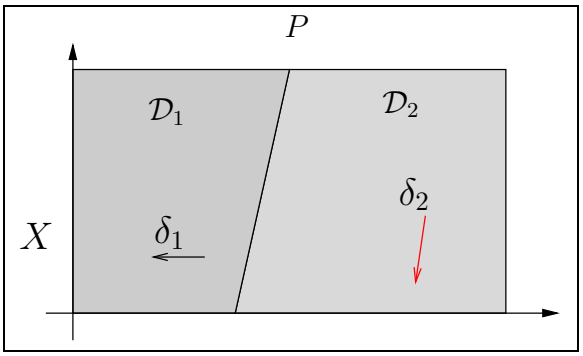


FIG. 5.8 – Variable X pour laquelle l'itération est infinie

Nous allons maintenant passer à la généralisation des résultats ; les résultats sont assez similaires.

2.3 Généralisation de l'expression de la variable X

Nous allons ici généraliser l'expression définissant la variable X. Tout d'abord, nous allons toujours supposer que la variable X est définie par deux branches, mais nous supposerons que les expressions de ces branches utilisent plusieurs occurrences de X. Puis, nous étudierons le cas où X est définie par plusieurs branches. Enfin, nous commenterons ces résultats.

L'expression d'une des branches de X contient plusieurs occurrences de X. Tout d'abord, nous considérons le cas où l'une des branches de l'équation définissant la variable X contient plusieurs occurrences de X. La variable X est donc définie par l'équation suivante :

$$X = \begin{cases} \mathcal{D}_1 : & X.\delta_1 \\ \mathcal{D}_2 : & X.\delta_2 \between_1 X.\delta_3, \text{ où } \between_1 \in \{\vee, \wedge\} \end{cases}$$

Le résultat est assez semblable au précédent. Nous pouvons montrer que pour que l'itération soit finie, il faut que les projections de δ_2 et δ_3 sur $\vec{\omega}$ soient négatives. En effet, supposons δ_3 positif, alors on peut à chaque étape k d'itération ajouter une variable W_k sur le domaine $\mathcal{D}_{W_k}$:

$$\begin{aligned} W_k = \quad & X.\delta_3^k \circ d_1 \\ \between_1 \quad & \bigbetween_{1,\, p+q=k, p\leq m_k} X.\delta_2^p \circ \delta_3^q \circ d_1 \\ \between_1 \quad & \bigbetween_{1,\, p+q+r=k, r>1, p\leq m_{k-1}} X.\delta_1^r \circ \delta_2^p \circ \delta_3^q \circ d) \\ \between_1 \quad & X.\delta_1^k \circ d_2 \end{aligned}$$

$$\begin{aligned} D_{W_k} = \quad & D_{W_{k-1}} \\ \cap \quad & d_1^{-1} \circ \delta_3^{-k}(\mathcal{D}_2) \\ \cap \quad & \cap_{p+q=k, p\leq m_k} d_1^{-1} \circ \delta_3^{-q} \circ \delta_2^{-p}(\mathcal{D}_2) \\ \cap \quad & \cap_{p+q+r=k, r>1, p\leq m_{k-1}} d_1^{-1} \circ \delta_3^{-q} \circ \delta_2^{-p} \circ \delta_1^{-r}(\mathcal{D}_1) \\ \cap \quad & d_2^{-1} \circ \delta_1^{-r}(\mathcal{D}_1) \end{aligned}$$

où m_k est un entier dépendant de k. On définit la variable W_0 par W et m_0 vaut 0. En appliquant la recherche d'auto-dépendance sur cette variable nous générons un certain nombre de variables (dont les domaines sont pour la plupart vide). Nous allons ici nous intéresser à deux de ces variables qui auront la

même forme que W_k et nous représentons les domaines de ces deux variables dans la figure 5.10.

$$
\begin{aligned}
W^1_{k+1} = \quad & X.\delta_3^{k+1} \circ d_1 \\
\curlywedge_1 \quad & \bigcurlywedge_{1 \atop p+q=k, p\leq m_k} X.\delta_2^{p} \circ \delta_3^{q+1} \circ d_1 \\
\curlywedge_1 \quad & \bigcurlywedge_{1 \atop p+q=k, p\leq m_k} X.\delta_2^{p+1} \circ \delta_3^{q} \circ d_1 \\
\curlywedge_1 \quad & \bigcurlywedge_{1 \atop p+q+r=k, r>1, p\leq m_{k-1}} X.\delta_1^{r+1} \circ \delta_2^{p} \circ \delta_3^{q} \circ d_1 \\
\curlywedge_1 \quad & X.\delta_1^{k+1} \circ d_2 \\
W^2_{k+1} = \quad & X.\delta_3^{k+1} \circ d_1 \\
\curlywedge_1 \quad & \bigcurlywedge_{1 \atop p+q=k, p< m_k} X.\delta_2^{p} \circ \delta_3^{q+1} \circ d_1 \\
\curlywedge_1 \quad & \bigcurlywedge_{1 \atop p+q=k, p< m_k} X.\delta_2^{p+1} \circ \delta_3^{q} \circ d_1 \\
\curlywedge_1 \quad & X.\delta_1^{1} \circ \delta_2^{m_k} \circ \delta_3^{k-m_k} \circ d_1 \\
\curlywedge_1 \quad & \bigcurlywedge_{1 \atop p+q+r=k, r>1, p\leq m_{k-1}} X.\delta_1^{r+1} \circ \delta_2^{p} \circ \delta_3^{q} \circ d_1 \\
\curlywedge_1 \quad & X.\delta_1^{k+1} \circ d_2
\end{aligned}
$$

On remarque alors que $m^1_{k+1} = m_k + 1$ et que $m^2_{k+1} = m_k$.

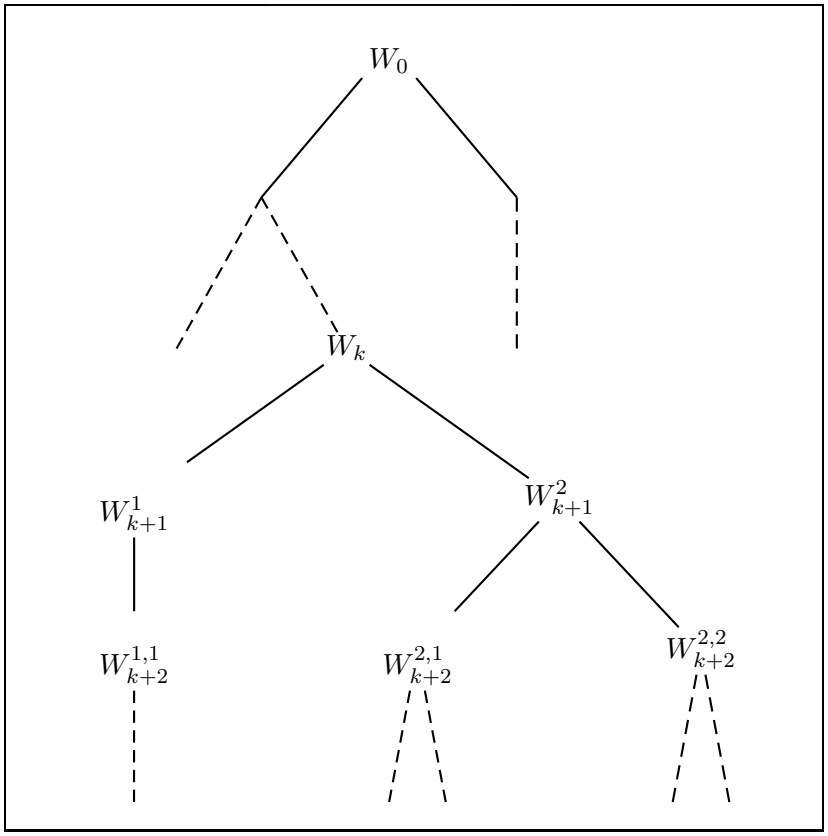
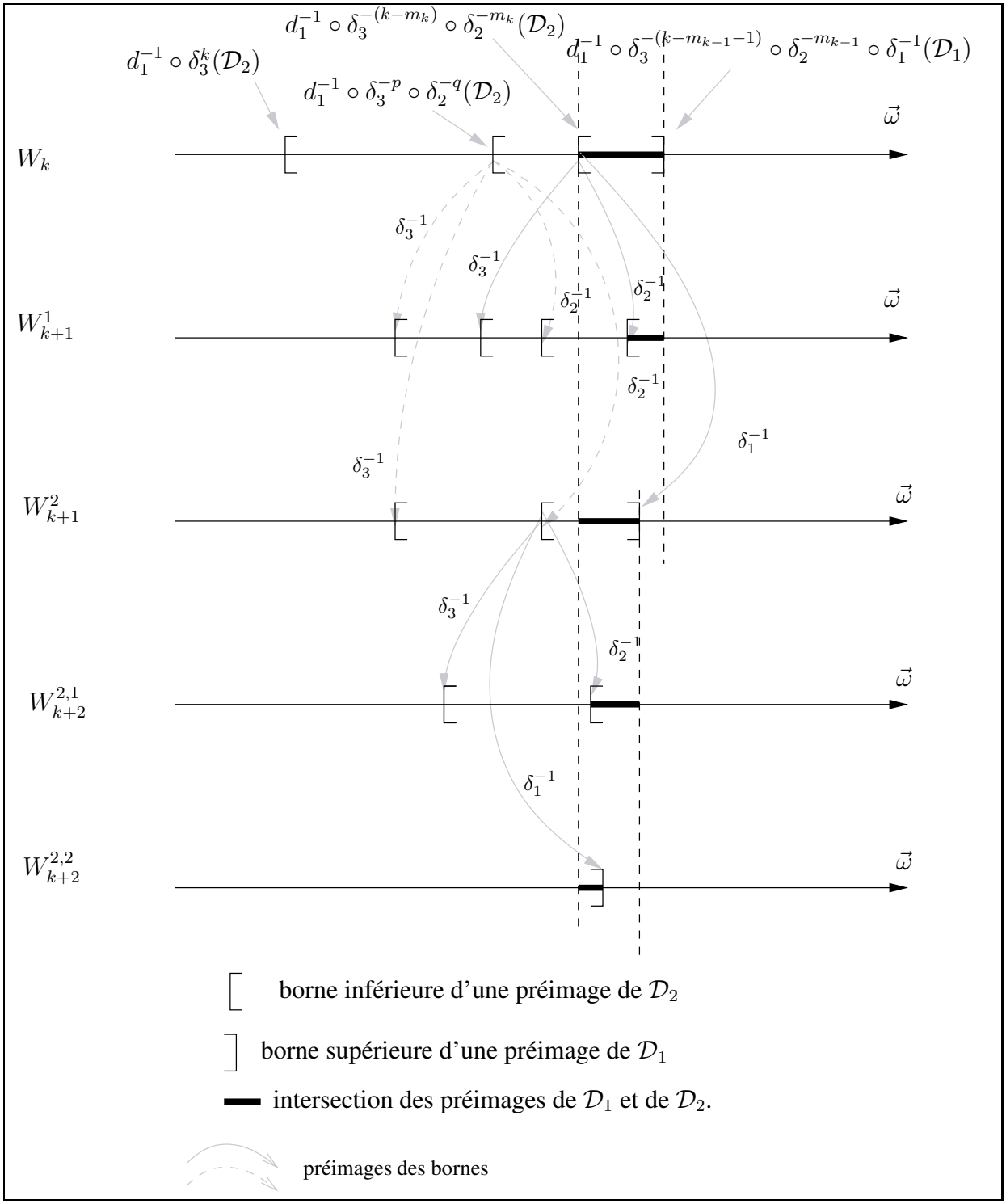


FIG. 5.9 – Arborescence des variables W_k.

Dans les deux variables W^1_{k+1} et W^2_{k+1}, la recherche d'auto-dépendance échouera. La figure 5.10 illustre les domaines de ces deux variables.



La préimage par la dépendance δ_1 fait changer le sens de la borne car nous prenons la préimage du domaine $\mathcal{D}_1$. Nous ne représentons pas les préimages de toutes les bornes.

FIG. 5.10 – Domaines des variables W_k, W_{k+1}^1, W_{k+1}^2, $W_{k+2}^{2,1}$, $W_{k+2}^{2,2}$ projetés sur $\vec{\omega}$.

– Pour la variable W^1_{k+1}, nous avons fait une substitution de $X.\delta_2^{m_k} \circ \delta_3^{k-m_k} \circ d_1$ par la deuxième branche de X. Nous devons donc calculer (entre autres) les deux préimages de $\mathcal{D}_2$ par $(\delta_2^{m_k+1} \circ \delta_3^{k-m_k} \circ d_1)^{-1}$ et $(\delta_2^{m_k} \circ \delta_3^{k-m_k+1} \circ d_1)^{-1}$. Le domaine de W^1_{k+1} n'est pas vide, et la recherche de motif échoue. Nous pouvons donc répéter le processus pour cette variable (non représentée sur la figure).
– Pour la variable W^2_{k+1}, nous avons fait une substitution de $X.\delta_2^{m_k} \circ \delta_3^{k-m_k} \circ d_1$ par la première branche de X. Nous calculons cette fois-ci la préimage de $\mathcal{D}_1$ par $(\delta_2^{m_k} \circ \delta_3^{k-m_k} \circ \delta_1 \circ d_1)^{-1}$. Le domaine de W^2_{k+1} n'est pas vide, nous pouvons donc recommencer une recherche d'auto-dépendance et nous engendrons les deux variables $W^{2,1}_{k+2}$ et $W^{2,2}_{k+2}$.

A chaque étape, nous engendrons soit deux nouvelles variables soit une variable. Nous avons représenté dans la figure 5.9 l'arborescence des variables générées par la variable W_0. Nous pouvons donc générer une infinité de variables, dont les domaines ne tendent pas à l'infini vers le vide mais contiennent toujours un point. L'itération sera donc infinie.

Variable X définie par plusieurs branches. Si maintenant nous supposons que X est définie par plus de deux branches :

$$X \quad = \quad \begin{cases} \mathcal{D}_1 : & X.\delta_1 \\ \mathcal{D}_2 : & X.\delta_2 \\ \mathcal{D}_3 : & X.\delta_3 \end{cases}$$

Nous étudions les branches deux à deux. Pour chaque couple de branches nous pouvons associer un hyperplan et un vecteur orthogonal. Nous appliquons le théorème 5.5 pour chaque couple de branches et chaque hyperplan. L'itération sera finie si pour chaque couple de branche l'itération est finie. Cependant, il n'est pas nécessaire d'appliquer le théorème pour tous les couples de branches, car si les domaines des branches ne sont pas mitoyens l'intersection de leurs préimages par d_2 et d_1 sera toujours vide.

Commentaires Ces résultats peuvent paraître décevants car plus contraignants que le cas de base. Il ne faut cependant pas perdre de vue que nos preuves porteront sur des systèmes réels. Pour de tels systèmes, le cas théorique général n'est absolument pas réaliste, comme nous allons le voir ci-dessous.

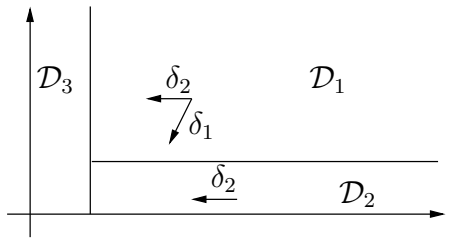
Si nous étudions comment sont définis des systèmes réels, nous constatons que les variables sont définies par plusieurs branches. Ces branches correspondent soit à une phase d'initialisation, soit aux cellules du bord du réseau (pour un réseau linéaire, celles-ci correspondent à la première et à la dernière cellule), soit au cas général (une cellule quelconque ne correspondant pas au bord du réseau, à un instant situé après la phase d'initialisation). Les équations des différentes branches sont définies de la manière suivante :

– cas général : une équation récurrente uniforme quelconque ;
– phase d'initialisation : les expressions font intervenir soit d'autres variables, soit des constantes : elles ne correspondent donc pas à notre étude actuelle ;
– cellules au bord du réseau : c'est un cas dégénéré du cas principal. L'équation a la même structure et les mêmes dépendances que l'équation principale. Certaines occurrences de variables peuvent avoir disparu (les instances de ces occurrences sortiraient du domaine de définition de X) : la figure 5.11 représente un cas réaliste.

Ainsi, dans des cas concrets, nous avons des dépendances en commun dans les différentes branches. Dans ce cas, l'itération s'arrête, puisqu'il suffit qu'il y ait une dépendance en commun entre deux branches pour que l'itération s'arrête. Ces résultats sont donc en pratique très utiles, et nous pouvons itérer la recherche d'auto-dépendance et la substitution par constante.

$$X = \begin{cases} \mathcal{D}_1 : & X.\delta_1 \uplus_1 X.\delta_2 \\ \mathcal{D}_2 : & X.\delta_2 \\ \mathcal{D}_3 : & Y.\delta \end{cases}$$

(a) équation définissant X



(b) Représentation de X et de ses dépendances

FIG. 5.11 – Variable X représentant un cas réaliste.

3 Les autres cas

Dans les deux sections précédentes nous avons étudié le problème de l'itération sous les hypothèses suivantes :

- La variable W n'est définie que par des occurrences de X et par un unique type d'opérateur,
- La variable X n'est définie que par des occurrences d'elle-même. Nous avons tout d'abord étudié le cas où X était défini par une unique branche puis nous avons étudié le cas où X est défini par plusieurs branches. Dans les deux cas, nous avons caractérisé les cas où l'itération était finie.

Nous allons maintenant généraliser cette étude. Nous commencerons par voir le cas où W est défini par plusieurs variables et nous caractériserons les cas où la recherche d'auto-dépendance réussit, puis nous supposerons que X est définie par plusieurs variables.

3.1 Motif W défini par plusieurs variables

Nous allons caractériser quelques cas de succès de la recherche d'auto-dépendance. Nous supposons la variable W définie par deux variables polyédriques X et Y :

$$W = X.d_1 \uplus_1 Y.d_2, \text{ où } \uplus_1 \in \{\vee, \wedge\}$$

Théorème 5.9 *Nous allons ici donner quelques cas où la recherche d'auto-dépendance réussit :*

- *Si les variables X et Y sont définies par les équations suivantes :*

$$\begin{aligned} X &= X.\delta_1 \uplus_1 e \\ Y &= Y.\delta_2 \uplus_1 e' \end{aligned}$$

  où e et e' sont des expressions polyédriques quelconques, alors la recherche d'auto-dépendance réussira si et seulement si les dépendances δ_1 et δ_2 sont parallèles.
- *Si les variables X et Y sont définies par :*

$$\begin{aligned} X &= Y.\delta_1 \uplus_1 e \\ Y &= X.\delta_2 \uplus_1 e' \end{aligned}$$

  où e et e' sont des expressions polyédriques quelconques, alors la recherche d'auto-dépendance réussira.
- *Si maintenant la variable X est définie par :*

$$X = Y.\delta_2 \uplus_1 X.\delta_1 \ldots$$

  alors la recherche d'auto-dépendance réussira si et seulement si

$$\delta_2 = d_2 \circ \delta_1 \circ d_1^{-1}$$

La démonstration est donnée en annexe 2.3.

Nous avons choisi de ne pas effectuer de recherche d'auto-dépendance dès que W était définie par plusieurs variables, les cas de succès étant en pratique trop rares. Nous verrons dans le chapitre suivant d'autres techniques permettant de construire des preuves pour de telles variables.

3.2 Variable X définie par plusieurs variables

Nous supposons que W est définie par des occurrences d'une seule variable :

$$W = \underset{i}{\barwedge_1} X.d_i, \text{ où } \barwedge_1 \in \{\vee, \wedge\}$$

et nous allons nous intéresser aux cas où la variable X est définie par des occurrences de plusieurs variables.

- Si X est définie par

  $$X = \bigbarwedge_{i=1}^{l}{}_1 X_i.d'_i$$

  alors il n'y a qu'une étape d'itération : nous trouvons une auto-dépendance et après simplification nous obtenons une constante. L'itération est donc finie (si nous supposons bien sûr que l'un des X_i représente X).
- Si maintenant X est définie par

  $$X = \bigbarwedge_{i=1}^{l}{}_2 X_i.d'_i$$

  avec $\barwedge_2 \neq \barwedge_1$, deux cas peuvent se produire :
  - soit il y a strictement plus d'une occurrence de X dans $\{X_1, \ldots, X_l\}$: l'itération est infinie,
  - soit il y a une seule occurrence de X dans $\{X_1, \ldots, X_l\}$: nous avons là encore deux possibilités
    - si toutes les occurrences des variables sont uniques : l'itération est finie,
    - s'il existe j et j' tels que $X_j = X_{j'}$, alors dans certains cas, l'itération ne sera pas finie. Il faut pour cela regarder le graphe de dépendance du système étudié : si la variable X et la variable X_j appartiennent à la même composante fortement connexe l'itération sera infinie, sinon l'itération sera finie.

  Actuellement, nous préférons ne pas effectuer de recherche d'auto-dépendance, car dans les cas où l'itération est finie, nous devons effectuer au minimum $l - 1$ nouvelles recherches d'auto-dépendance à chaque étape. En effet, si on suppose tous les X_i distincts, la première étape de la recherche d'auto-dépendance est de substituer la variable X par sa valeur et de transforme l'expression sous forme $\barwedge_2$-normale (nous n'écrivons pas les termes faisant apparaître plusieurs variables) :

  $$W = \bigbarwedge_{i=1}^{l}{}_2 \underset{j}{\barwedge_1} X_i.d'_i \circ d_j \barwedge_1 \ldots$$

  La recherche d'auto-dépendance est effectuée dans le terme ne faisant intervenir que X, et dans les $l-1$ termes ne faisant intervenir qu'une seule variable nous pouvons recommencer une recherche d'auto-dépendance. Ainsi, à chaque étape nous effectuons au moins $l - 1$ recherches d'auto-dépendance et pour chacune de ces recherches d'auto-dépendance, nous devons recommences de nouvelles recherches d'auto-dépendance. La complexité globale est donc exponentielle.
- Si X est définie par :

  $$X = \bigbarwedge_{j=1}^{l}{}_1 \bigbarwedge_{i=1}^{l_j}{}_2 X_i.d'_i$$

Nous pouvons effectuer là encore une recherche d'auto-dépendance si pour tout j, $\bigwedge\!\!\!\!\bigwedge_{i=1}^{l_j}{}_2\, X_i.d'_i$ répond aux critères énoncés dans le cas précédent. Cependant, nous n'effectuons pas la recherche pour la même raison que précédemment (complexité exponentielle).

4 Conclusion

Dans ce chapitre, nous avons étudié le problème de l'arrêt de l'itération de la recherche d'auto-dépendance et de la substitution par constante en faisant l'hypothèse que nos dépendances étaient uniformes. Dans le problème général, l'équation générale de W est la suivante

$$W = \mathbb{X}_2\, \mathbb{X}_1\, X_i.d_i$$

Nous avons vu dans l'introduction qu'il était toujours possible de se ramener à une équation définie par un unique opérateur :

$$W = \mathbb{X}_2\, \mathbb{X}_1\, X_i.d_i$$

Pour cette définition générale de W, nous avons caractérisé quelques cas (section 3) où la recherche d'auto-dépendance réussissait. Cependant, nous ne préférons pas itérer le processus d'itération qui sera rarement fini.

En restreignant l'équation de W au cas où W est définie par l'occurrence d'une unique variable X, nous avons pu obtenir des caractérisations beaucoup plus précises en supposant que X n'était définie que par des occurrences d'elle-même.

- Nous avons tout d'abord supposé que X était définie par une unique expression 1, et nous avons vu que si X était définie par l'opérateur $\mathbb{X}_1$ l'itération était finie, et que s'il y avait une occurrence de l'opérateur $\mathbb{X}_2$ l'itération était infinie. Nous avons aussi étudié l'influence de l'occurrence d'une négation dans la définition de X, et nous avons pu caractériser les cas finis des cas infinis.
- Nous avons ensuite étudié le cas où X était définie par une expression conditionnelle à plusieurs branches(*cf.* section 2). Nous avons vu que dans le cas théorique l'itération était rarement finie. Cependant, en nous restreignant à des cas réalistes, l'itération termine. Nous pouvons donc itérer la substitution par constante et la recherche d'auto-dépendance dans le cas d'expressions conditionnelles à plusieurs branches.

Enfin, en supposant X défini par des occurrences de plusieurs variables, nous avons pu caractériser quelques cas où l'itération terminait. Cependant la complexité de cette itération étant exponentielle, nous ne l'utilisons pas en pratique.

Pour résumer les cas où nous effectuons l'itération, il faut que W soit définie à l'aide d'une unique variable. Soit $\mathbb{X}_1$ l'opération utilisée dans l'équation de W. La variable X doit alors être définie par une $\mathbb{X}_1$-expression. Dans tous les autres cas, nous n'effectuons pas d'itération. Cependant cela ne pose pas de problème, car nous présenterons dans le chapitre suivant d'autres types d'itérations qui permettent de résoudre les cas dans lesquels l'itération de la substitution par constante et de la recherche d'auto-dépendance n'a pas été effectuée.

Pour conclure, nous donnons quelques résultats supplémentaires. Tout d'abord nous supprimons l'hypothèse de dépendance uniforme, puis dans la section suivante nous verrons une extension possible de ces résultats pour la preuve d'équivalence de SERU.

4.1 Dépendances non uniformes

Supprimons l'hypothèse de dépendance uniforme, le problème devient alors très complexe et reste à notre connaissance ouvert. Soient W et X deux variables polyédriques définies par les équations

suivantes.

$$\begin{aligned} W &= \mathcal{D}_W : X.d_1 \ \between_1 \ X.d_2 \\ X &= X.\delta \end{aligned}$$

Nous nous limitons ici au cas le plus simple pour X. On substitue les instances de X par leur définition dans l'équation de W. On obtient ainsi :

$$W = \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_X) \cap d_2^{-1}(\mathcal{D}_X) : X.\delta \circ d_1 \vee X.\delta \circ d_2$$

Nous avons deux systèmes à résoudre :

$$\left\{ \begin{array}{l} \delta \circ d_1 = d_1 \circ f \\ \delta \circ d_2 = d_2 \circ f \end{array} \right. \quad \left\{ \begin{array}{l} \delta \circ d_1 = d_2 \circ f \\ \delta \circ d_2 = d_1 \circ f \end{array} \right.$$

Pour résoudre ces deux systèmes, nous utilisons l'algorithme présenté dans l'annexe C.

Supposons qu'aucun des deux systèmes n'ait de solution, nous introduisons alors une nouvelle variable Z définie sur le domaine $\mathcal{D}_Z = \mathcal{D}_W \cap d_1^{-1}(\mathcal{D}_X) \cap d_2^{-1}(\mathcal{D}_X)$ par $Z = \mathcal{D}_Z : X.\delta \circ d_1 \vee X.\delta \circ d_2$. Nous cherchons donc une dépendance f telle que $Z = Z.f$ ou $Z = W.f$. Nous avons alors quatre systèmes à résoudre :

$$\left\{ \begin{array}{l} \delta^2 \circ d_1 = \delta \circ d_1 \circ f \\ \delta^2 \circ d_2 = \delta \circ d_2 \circ f \end{array} \right. \quad \left\{ \begin{array}{l} \delta^2 \circ d_1 = \delta \circ d_2 \circ f \\ \delta^2 \circ d_2 = \delta \circ d_1 \circ f \end{array} \right.$$

$$\left\{ \begin{array}{l} \delta^2 \circ d_1 = d_1 \circ f \\ \delta^2 \circ d_2 = d_2 \circ f \end{array} \right. \quad \left\{ \begin{array}{l} \delta^2 \circ d_1 = d_2 \circ f \\ \delta^2 \circ d_2 = d_1 \circ f \end{array} \right.$$

Si nous répétons le processus, la question que l'on se pose est donc la suivante : existe-t-il un n et un m tels que l'un des deux systèmes suivants ait une solution.

$$\left\{ \begin{array}{l} \delta^n \circ d_1 = \delta^m \circ d_1 \circ f \\ \delta^n \circ d_2 = \delta^m \circ d_2 \circ f \end{array} \right. \quad \left\{ \begin{array}{l} \delta^n \circ d_1 = \delta^m \circ d_2 \circ f \\ \delta^n \circ d_2 = \delta^m \circ d_1 \circ f \end{array} \right.$$

Si les dépendances d_1 et d_2 correspondent à une projection suivie d'une translation et si δ est une dépendance uniforme, nous sommes capable de résoudre le problème. Mais, ici nous nous intéressons au cas beaucoup plus général et nous allons voir comment formuler ce problème.

Nous nous intéressons au premier système, le deuxième se comportant de manière identique. Nous utilisons la représentation matricielle des dépendances. Le problème se formule alors de la manière suivante :

> *Soit A une matrice carrée à coefficients dans $\mathbb{Z}$, soient D_1 et D_2 deux matrices à coefficients dans $\mathbb{Z}$. Existe-t-il deux entiers m et n tels que le système S_1 ci-dessous ait une solution X dans l'ensemble des matrices à coefficients dans $\mathbb{Z}$?*

$$S_1 = \left\{ \begin{array}{l} A^n.D_1 = A^m.D_1.X \\ A^n.D_2 = A^m.D_2.X \end{array} \right.$$

Si les matrices sont toutes inversibles, le système se réécrit en

$$S_2 = \left\{ \begin{array}{l} D_1^{-1}.A^{n-m}.D_1 = X \\ D_2^{-1}.A^{n-m}.D_2 = X \end{array} \right.$$

Il faut donc être capable de trouver n et m tels que

$$A^{n-m}.D_1.D_2^{-1} = D_1.D_2^{-1}.A^{n-m}$$

Nous pouvons montrer que dans les cas où les matrices ne sont pas inversibles nous devons résoudre une équation similaire mais cette fois ci, modulo un sous-anneau de matrices.

D'une manière plus synthétique, le problème est donc le suivant.

Soient A, P_1, P_2 trois matrices à coefficients dans $\mathbb{Z}$ inversibles, existe-t-il un entier k tel que

$$P_1^{-1}.A^k.P_1 = P_2^{-1}.A^k.P_2$$

ou, autrement dit, existe-t-il un entier k tel que la matrice A^k soit la même dans les bases P_1 et P_2 ?

À notre connaissance, le seul problème de décidabilité ressemblant (mais nous n'avons, pour le moment, pas réussi à nous y ramener) est ouvert. C'est le problème de Skolem : soit A une matrice carrée de dimension $n \times n$ à coefficients dans $\mathbb{Z}$, existe-t-il un entier k tel que le coefficient $(A^k)_{1n}$ soit égal à 0 ? Ce problème est actuellement ouvert dès que n est supérieur à 3 [42].

4.2 Utiliser la recherche d'auto-dépendances pour des preuves d'équivalence

Pour terminer, nous revenons sur le cas où W est définie par :

$$W = X.d_1 \between_1 \neg X.d_2$$

Quelle que soit la définition de X nous effectuons une étape d'itération et l'itération s'arrête. Ce cas est intéressant car il permet de prouver l'équivalence entre deux parties du domaine de X. Nous pensons qu'il serait aussi intéressant d'utiliser cette méthode pour effectuer des preuves d'équivalence entre deux variables. Pour cela, il faudrait autoriser la recherche d'auto-dépendance dans le cas suivant :

$$W = X_1.d_1 \between_1 \neg X_2.d_2$$

où les variables X_1 et X_2 sont distinctes. Ce cas demanderait une étude plus approfondie lors de l'itération recherche d'auto-dépendance, substitution par constante. Cela pourrait être une suite de ce travail.

Le chapitre suivant sera consacré à l'automatisation du processus de preuve ce qui nous permettra de présenter l'implémentation.

Chapitre 6

Automatisation et implémentation du processus de preuve

Dans le chapitre précédent, nous avons vu comment itérer la recherche d'auto-dépendance et la substitution par constante. Ces deux règles sont extraites du système de preuve présenté dans le chapitre 4 (définition 4.6, p. 59). Nous rappelons que le but de ce travail est de prouver automatiquement des propriétés sur un système. Le but de ce chapitre sera donc de fournir un algorithme de construction de la preuve. Dans le chapitre 3 d'introduction à cette partie, nous avons esquissé sur un exemple comment effectuer une preuve. Nous allons reprendre ici cet exemple pour montrer la structure de données que nous allons utiliser, c'est-à-dire un arbre.

Rappelons que le problème était de prouver que pour tout environnement ρ, tel que $\rho \propto S$,

$$\models_\rho \mathcal{D}_X : X$$

nous devons donc donner une preuve :

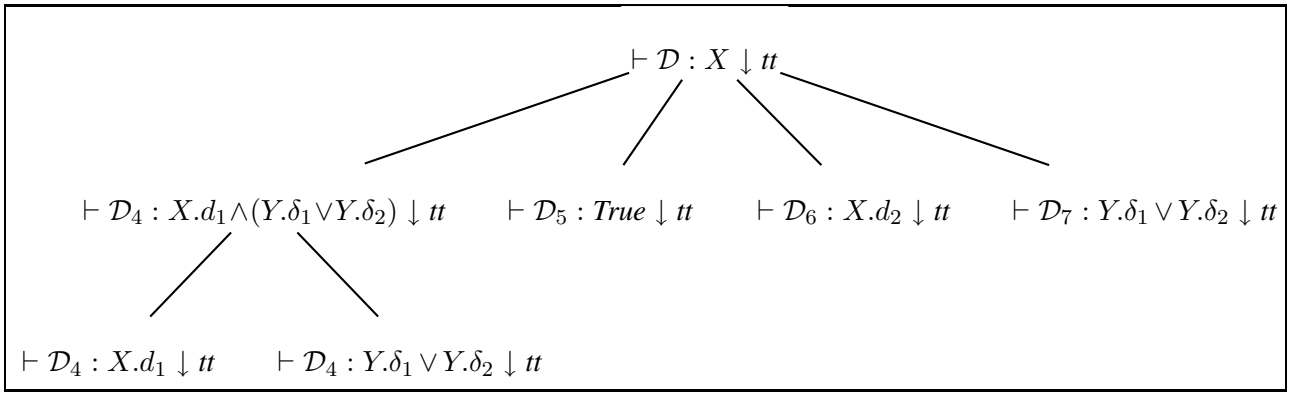
$$S \vdash \mathcal{D}_X : X \downarrow tt$$

Dans le chapitre 3, nous avons esquissé les différentes règles de preuve à notre disposition, ce qui nous a permis de montrer le mécanisme de preuve. Dans la figure 6.1, nous reprenons sous forme d'arbre le mécanisme de preuve en le formalisant. Les fils de chaque nœud de l'arbre sont obtenus en appliquant une règle de preuve. Ils correspondent aux nouveaux sous-buts à prouver.

Ce chapitre sera organisé de la manière suivante. Dans la première section, nous présenterons un premier algorithme s'appuyant sur les règles de preuve que nous avons présentées dans les chapitres 4 et 5. Nous présenterons tout d'abord les différentes fonctions implémentant les règles de preuve, puis l'algorithme qui automatise la construction de l'arbre de preuve et nous étudierons le problème de l'arrêt de cet algorithme. La section 2 présentera un nouveau système de règles de preuve qui permettra une meilleure automatisation de la construction de l'arbre. Dans la section 3, nous présenterons tout d'abord un nouvel algorithme de construction de l'arbre s'appuyant sur le nouveau système de règles, et enfin, dans l'optique d'obtenir une construction la plus automatique possible, nous raffinerons cet algorithme en itérant la construction de l'arbre.

1 Construction de l'arbre et preuve

Dans cette section, nous allons nous intéresser à la construction d'un arbre de preuve. Pour cela, nous allons nous inspirer des règles de preuves présentées dans le chapitre 4. Nous commencerons par rappeler les règles de preuve. Puis, nous verrons sur un exemple les différentes règles de construction de l'arbre de preuve. Nous verrons alors le lien entre la construction d'un arbre et l'existence d'une



$$\begin{aligned}
\mathcal{D}_4 &= \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D} \\
\mathcal{D}_5 &= d_2^{-1}(\mathcal{D}) \cap \mathcal{D}_2 \cap \mathcal{D} \\
\mathcal{D}_6 &= \mathcal{D}_2 \setminus (d_2^{-1}(\mathcal{D})) \cap \mathcal{D} \\
\mathcal{D}_7 &= d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1 \cap \mathcal{D}
\end{aligned}$$

FIG. 6.1 – structure de la preuve de l'exemple présenté dans l'introduction (*cf.* chapitre 3).

preuve. Nous donnerons alors un algorithme de construction de l'arbre de preuve. Enfin, nous verrons que l'algorithme termine, soit en fournissant une preuve ou un contre-exemple, soit sur un échec de la construction.

1.1 Règles de preuve

Soit S un système, soit $\mathcal{D} : e \downarrow v$ une formule. Le but est de fournir une preuve de la validité de cette formule, c'est-à-dire prouver

$$S \vdash \mathcal{D} : e \downarrow v$$

Nous rappelons que notre système de preuve est formé des règles de preuve suivantes :

- axiomes : AX1, AX2, AX3 (p. 59),
- règles de décomposition des opérandes : OU, ET (p 59),
- règle de la négation : NEG1, NEG2 (p. 59),
- règle de la dépendance : DEP (p. 59),
- règle de l'équation : EQ (p. 59),
- règle de la séparation des branches : SEP (p. 59),
- règle de la substitution par constante : SUBS (p. 61),
- règle de la simplification : SIMP (p. 66),
- règle de recherche d'auto-dépendance : AUTO (p. 72),
- règle de l'introduction de variable : NOUV (p. 60).

De plus nous rappelons qu'une preuve de la formule $\mathcal{D} : e \downarrow v$ existe si et seulement s'il existe une séquence finie de formules étendues par une valeur $\alpha_1, \ldots, \alpha_n$ telle que $\alpha_n = \mathcal{D} : e \downarrow v$ et chaque α_i est soit un axiome soit peut être déduit à partir d'une formule précédente en utilisant une des règles de preuve.

1.2 Règles de construction de l'arbre

Les règles de construction de l'arbre sont calquées sur les règles de preuve. Elles représentent l'implémentation de ces règles de preuve. Nous illustrerons ce chapitre par l'exemple suivant.

Exemple 6.1 *Soient S un système, X, Y, Z trois variables polyédriques de ce système et $\mathcal{D}_X$ le domaine de la variable X. On suppose X définie par l'équation suivante :*

$$X = \begin{cases} \mathcal{D}_1 : X.d_1 \wedge (Y.d_2 \vee Y.d_3) \\ \mathcal{D}_2 : (Y.d_4 \vee Z.d_5) \wedge \neg Y.d_6 \end{cases}$$

De plus, on suppose que le problème est de prouver que pour tout environnement ρ, tel que $\rho \propto S$,

$$\models_\rho \mathcal{D}_X : X$$

nous devons donc donner une preuve :

$$S \vdash \mathcal{D}_X : X \downarrow tt$$

Pour cela, nous allons construire un arbre de preuve qui, sous certaines conditions sur les feuilles de cet arbre, constituera une preuve pour la formule $\mathcal{D}_X : X \downarrow tt$.

L'arbre de preuve se construit de manière récursive. Les nœuds de cet arbre représentent des formules étendues, c'est-à-dire une expression e, son domaine $\mathcal{D}$ et sa valeur v de validité.

$$N = \langle e, \mathcal{D}, v \rangle$$

Racine La racine de l'arbre correspond à la formule étendue à prouver.

Exemple 6.2 *Dans notre exemple, la racine de l'arbre sera :*

$$N_0 = \langle X, \mathcal{D}_X, tt \rangle$$

Ici, l'expression à prouver est représentée par X.

Pour construire les fils d'un nœud, nous avons développé des fonctions s'appuyant sur les règles de preuve. Ces fonctions ont été implémentées en Mathematica, qui sert d'interface à MMALPHA. Les expressions et les systèmes polyédriques sont représentés par des arbres syntaxiques ALPHA. Ces arbres syntaxiques sont des listes Mathematica. L'utilisateur entre son expression sous forme de chaîne de caractères, et grâce à l'analyseur syntaxique de MMALPHA, la chaîne de caractères est transformée en une liste manipulable par Mathematica. L'objet de base de Mathematica est la liste, et de nombreuses fonctions sont prédéfinies pour manipuler les listes. L'implémentation des diverses fonctions de construction de l'arbre de preuve passe principalement par des manipulations d'arbres syntaxiques ALPHA. Nous utilisons donc les fonctions prédéfinies en Mathematica pour manipuler ces arbres et les fonctions prédéfinies en MMALPHA. En ce qui concerne les domaines des expressions, nous utilisons la librairie polyédrique pour effectuer les différents calculs (images, préimages, intersections, unions).

Les fonctions implémentant les règles de preuve ont toutes en entrée un nœud et un système et, en sortie, une liste de nœuds et un système. La liste de ces fonctions et leur spécification se trouvent en appendice E. Nous allons maintenant détailler la construction de l'arbre à partir de l'exemple.

Exemple 6.3 – *Nous partons donc de la racine N_0 :*

$$N_0 = \langle X, \mathcal{D}_X, tt \rangle$$

Nous posons $S_0 = S$. Si l'on étudie l'équation définissant la variable X, nous remarquons la présence d'auto-dépendances. La fonction **substitution_constante** *qui implémente la règle de même nom est appliquée sur ce nœud et fournit le résultat suivant :*

$$\textbf{substitution\_constante}\ (N_0, S_0) = (N_1, S_1)$$

où N_1 est défini par

$$N_1 = \langle X_{subs}, \mathcal{D}_X, tt \rangle$$

La variable X_{subs} est définie par l'équation suivante dans le nouveau système S_1.

$$X_{subs} = \begin{cases} \mathcal{D}_1 : True \wedge (Y.d_2 \vee Y.d_3) \\ \mathcal{D}_2 : (Y.d_4 \vee Z.d_5) \wedge \neg Y.d_6 \end{cases}$$

Le système S_1 est constitué du système S_0, de la variable X_{subs} et de l'équation définissant cette variable.

– *L'expression du nœud N_1 est une variable dont l'équation ne contient plus d'auto-dépendance. Appliquer la règle de substitution par constante n'apporterait donc pas de résultats nouveaux. En revanche utiliser la règle de l'équation permet d'obtenir l'expression de la variable. Ainsi, la fonction* **équation** *est appliquée sur le nœud N_1 et sur le système S_1, elle donne le résultat suivant :*

$$\textbf{équation}\ (N_1, S_1) = (N_2, S_2)$$

où le nœud N_2 est défini par :

$$N_2 = \left\langle \begin{cases} \mathcal{D}_1 : True \wedge (Y.d_2 \vee Y.d_3) \\ \mathcal{D}_2 : (Y.d_4 \vee Z.d_5) \wedge \neg Y.d_6 \end{cases}, \mathcal{D}_X, tt \right\rangle$$

Le système S_2 est identique au système S_1.

– *L'expression du nœud N_2 contient la constante True, la règle de simplification peut donc être utilisée sur le nœud N_2 et le système S_2 pour obtenir :*

$$\textbf{simplification}\ (N_2, S_2) = (N_3, S_3)$$

où le nœud N_3 est défini par

$$N_3 = \left\langle \begin{cases} \mathcal{D}_1 : (Y.d_2 \vee Y.d_3) \\ \mathcal{D}_2 : (Y.d_4 \vee Z.d_5) \wedge \neg Y.d_6 \end{cases}, \mathcal{D}_X, tt \right\rangle$$

Le système S_3 correspond au système S_2 et donc au système S_1.

– *L'expression du nœud N_3 est une expression à plusieurs branches. La règle de séparation des branches s'applique donc sur un tel nœud. La fonction correspondante est* **séparation_branches***, elle fournit le résultat :*

$$\textbf{séparation\_branches}\ (N_3, S_3) = (N_4, N_5, S_4)$$

Nous obtenons ici deux nouveaux nœuds définis par :

$$N_4 = \langle Y.d_2 \vee Y.d_3, \mathcal{D}_1 \cap \mathcal{D}_X, tt \rangle$$
$$N_5 = \langle (Y.d_4 \vee Z.d_5) \wedge \neg Y.d_6, \mathcal{D}_2 \cap \mathcal{D}_X, tt \rangle$$

Le système S_4 correspond au système S_3.

– *En supposant la variable Y définie par $Y = Y.d$, nous reconnaissons, pour le nœud N_4 la situation où la recherche d'auto-dépendance va réussir. La première étape est d'ajouter une nouvelle variable. C'est le rôle de la fonction* **ajout_variable**. *L'application de cette fonction au nœud N_4 et au système S_4 permet d'obtenir :*

$$\textbf{ajout\_variable}(N_4, S_4) = (N_6, S_6)$$

Le nœud N_6 est défini par :

$$N_6 = \langle W, \mathcal{D}_1 \cap \mathcal{D}_X, tt \rangle$$

où la variable W est définie dans le système S_6 par

$$W = Y.d_2 \vee Y.d_3$$

Le système S_6 correspond au système S_4 et contient en plus la variable W et l'équation définissant cette variable.

– *La seconde étape consiste à appliquer la fonction* **recherche_auto_dépendance** *sur le nœud N_6 et le système S_6 :*

$$\textbf{recherche\_auto\_dépendance}(N_6, S_6) = (N_7, S_7)$$

où le nœud N_7 est défini par

$$N_7 = \langle W_{rec}, \mathcal{D}_1 \cap \mathcal{D}_X, tt \rangle$$

La variable W_{rec} est définie par :

$$W_{rec} = W_{rec}.d$$

Le système S_7 est défini par le système S_6 et il contient de plus la variable W_{rec} et l'équation la définissant. La règle de substitution par constante peut maintenant être utilisée.

– *Nous reprenons maintenant le nœud N_5. L'expression de ce nœud est une conjonction, la règle de séparation des opérandes s'applique. La fonction* **séparation_opérandes** *implémente cette règle, elle fournit le résultat suivant :*

$$\textbf{séparation\_opérandes}(N_5, S_4) = (N_8, N_9, S_9)$$

où les nœuds N_8 et N_9 sont définis par :

$$N_8 = \langle Y.d_4 \vee Z.d_5, \mathcal{D}_2 \cap \mathcal{D}_X, tt \rangle$$
$$N_9 = \langle \neg Y.d_6, \mathcal{D}_2 \cap \mathcal{D}_X, tt \rangle$$

Le système S_9 correspond au système S_4.

– *Sur le nœud N_8 la règle d'ajout de variable est utilisée, puis plus une seule règle ne peut être utilisée : la recherche d'auto-dépendance échoue.*

– *L'expression du nœud N_9 est définie par une négation. La règle à appliquer est donc celle de la négation. Elle est implémentée par la fonction* **négation**.

$$\textbf{négation}(N_9, S_9) = (N_{10}, S_{10})$$

où le nœud N_{10} est défini par

$$N_{10} = \langle Y.d_6, \mathcal{D}_2 \cap \mathcal{D}_X, ff \rangle$$

Le système S_{10} est égal au système S_9 (et donc au système S_1).

– *Pour le nœud N_{10}, nous avons une dépendance. La fonction* **dépendance** *qui implémente la règle de même nom est utilisée et donne le résultat suivant :*

$$\textbf{dépendance}(N_{10}, S_{10}) = (N_{11}, S_{11})$$

où

$$N_{11} = \langle Y, d_6^{-1}(\mathcal{D}_2 \cap \mathcal{D}_X), \mathit{ff} \rangle$$

Le système S_{11} correspond au système S_1.

Nous avons donc présenté ici les différentes règles de construction de l'arbre de preuve. L'application des règles est automatique (nous verrons dans la section suivante l'automatisation de l'enchaînement des règles). Nous allons maintenant nous intéresser à l'arrêt de la construction de l'arbre et au lien existant entre cette construction et l'existence ou non d'une preuve.

Arrêt de la construction de l'arbre. La construction d'une branche de l'arbre se termine quand aucune fonction ne peut être appliquée sur le dernier nœud (feuille) de la branche. Dans ce cas, l'expression e de cette feuille peut-être de trois formes :

– soit une constante,
– soit un motif composé de variables d'entrée,
– soit une expression sur laquelle aucune règle ne peut s'appliquer.

Les deux premiers cas sont considérés comme des cas de terminaison. En effet, lorsque nous avons des variables d'entrées, la recherche d'auto-dépendance n'a aucun sens. Nous ne pouvons pas substituer ces variables par leur définition (qui n'existe pas, ce sont des variables d'entrée). Dans un tel cas, l'expression obtenue est une condition nécessaire sur les variables d'entrées. Pour le dernier cas, la feuille représente un sous-but non résolu dans le système de preuve. L'utilisateur doit intervenir pour permettre à la construction de continuer. Nous voyons donc que nous pouvons ici distinguer deux types de feuilles celles pour lesquelles la construction est réellement terminée, et celles pour lesquelles l'utilisateur doit encore intervenir. Nous définissons donc les notions suivantes de feuille fermée et de feuille pendante.

Définition 6.4 (feuille fermée) *Une feuille fermée est une feuille dans laquelle l'expression polyédrique est une constante.*

Définition 6.5 (feuille pendante) *Une feuille pendante est une feuille qui n'est pas une feuille fermée.*

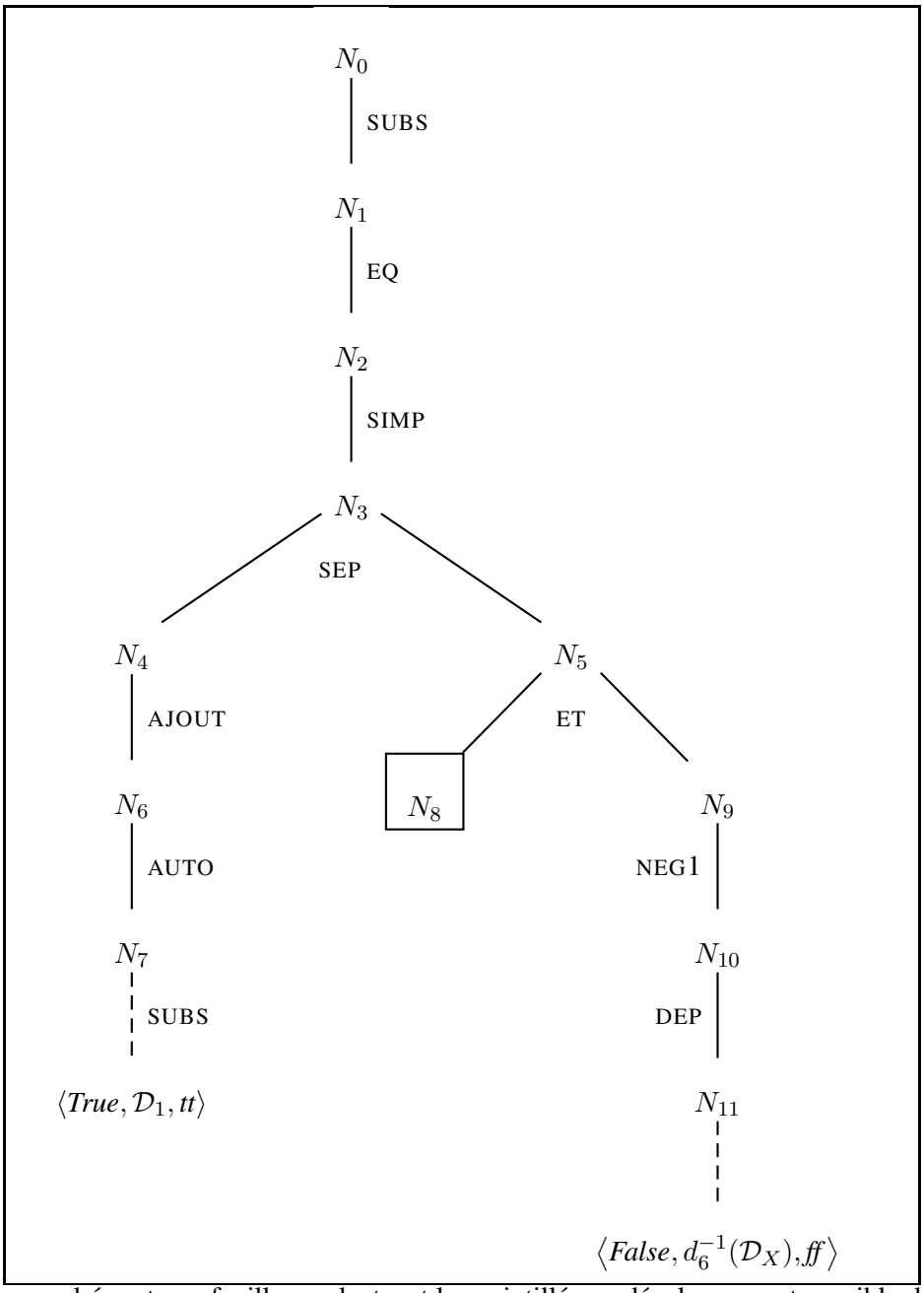
Ainsi, les feuilles contenant des variables d'entrées sont considérées comme des feuilles pendantes bien qu'aucune règle de preuve ne puisse être appliquée dessus. Nous ne pouvons pas les considérer comme des feuilles fermées car la preuve n'est pas terminée sur ces feuilles.

Dans la figure 6.2, nous reprenons l'exemple et construisons l'arbre de preuve. Nous avons tracé en pointillé un développement possible de l'arbre. Il y a deux feuilles pour lesquelles nous obtenons des constantes, ce sont donc des feuilles fermées. Nous constatons que les constantes sont bien les valeurs attendues et donc dans ces branches, la preuve a réussi. Il reste cependant une feuille pendante (encadrée) pour laquelle la preuve a échoué. Nous reviendrons par la suite sur cet échec.

Construction de l'arbre et existence d'une preuve. Jusqu'à présent nous avons étudié les règles de construction de l'arbre de preuve, et nous avons vu que ces règles de construction implémentaient les règles de preuve. On peut donc maintenant étudier le lien entre l'arbre ainsi construit et l'existence d'une preuve.

Pour que la construction de l'arbre constitue une preuve, il faut bien sûr que toutes les feuilles soient fermées.

Le théorème suivant énonce une condition suffisante sur les feuilles de l'arbre de preuve pour l'existence d'une preuve.



La feuille encadrée est une feuille pendante, et les pointillés un développement possible de l'arbre.

FIG. 6.2 – Arbre de preuve

Théorème 6.6 *Soit S un système, soit $\mathcal{D} : e \downarrow tt$ une formule sur ce système, soit un arbre de preuve dont la racine est $\langle e, \mathcal{D}, tt \rangle$. Si toutes les feuilles sont fermées, et si toutes les feuilles sont d'une des deux formes suivantes :*

$$\langle True, \mathcal{D}, tt \rangle$$
$$\langle False, \mathcal{D}, ff \rangle$$

alors l'arbre sera une preuve de la formule $\mathcal{D} : e$.

Démonstration : *La démonstration est simple, il suffit de remarquer qu'à chaque étape de la construction de l'arbre nous avons utilisé une règle de preuve et que les feuilles correspondent aux axiomes de notre système de preuve.* ❑

La condition que nous venons d'énoncer est extrêmement forte, elle ne tient pas compte par exemple du cas des variables d'entrée. Cependant, il ne faut pas perdre de vu que le système de preuve que nous utilisons ici n'est pas suffisant. En particulier, nous ne faisons pas intervenir le contexte (*cf.* page 59). De plus, le principe de substitution et celui de recherche de motifs (*cf.* page 66) sont mal exploités. Dans la section 2, nous donnerons un nouveau système de preuve exploitant mieux ces deux principes et permettant de continuer la preuve sur des formules contenants des variables d'entrées en utilisant quelques hypothèses sur les variables d'entrées. Nous ne parlerons pas dans ce chapitre du contexte car son utilisation n'est pas encore automatisé. Les règles de preuve concernant le contexte seront présentées dans le chapitre 9. Dans le cas des feuilles pendantes, l'utilisateur doit intervenir, et fournir par exemple des invariants, ou alors utiliser des techniques d'accélérations ou d'agrandissement de domaine que nous étudierons dans le chapitre suivant.

Nous avons donc ici présenté comment construire l'arbre de preuve et nous avons établi le lien entre construction de l'arbre et existence d'une preuve. Nous allons maintenant expliquer plus en détails comment construire automatiquement l'arbre, c'est-à-dire comment choisir automatiquement la règle à appliquer.

1.3 Algorithme de construction des arbres de preuve

Nous allons ici nous intéresser à la construction automatique de l'arbre de preuve. À chaque nœud de l'arbre, nous ajoutons un champ s représentant la règle à appliquer. Ce champ s est appelé statut. Un nœud a donc maintenant la forme suivante :

$$\langle e, \mathcal{D}, v, s \rangle$$

où e est une expression polyédrique, $\mathcal{D}$ un sous-domaine du domaine de e, et v une valeur booléenne. Nous avons présenté dans la section précédente les différentes règles utilisées pour construire l'arbre de preuve. Certaines règles sont toujours combinées ensemble. Le statut va donc représenter soit la combinaison de règles à appliquer soit la cause de l'arrêt. Ce statut peut avoir cinq valeurs différentes.

Signification de la valeur du statut

- **Variables d'entrée et Constantes** (*VarEnt*) : Cette valeur du statut représente les deux cas d'arrêt corrects. C'est-à-dire que nous avons soit une constante, soit une expression utilisant des variables d'entrées.
- **Substitution par constante** (*SubsCons*) : Pour un nœud dont le statut à cette valeur, nous devons appliquer une substitution par constante.
- **Séparation des opérandes** (*SepOp*) : Il faut ici appliquer les règles de séparation des opérandes et de négation. Ces règles sont combinées avec la règle de dépendance.
- **Recherche d'auto-dépendance** (*RecAuto*) : Cette valeur indique que nous devons effectuer une recherche d'auto-dépendance.

– **Échec de la recherche d'auto-dépendance** (*EchAuto*) : Cette valeur représente le cas d'échec : nous ne pouvons pas effectuer de recherche d'auto-dépendance et aucune autre règle ne peut être appliquée.

Nous allons présenter dans le paragraphe suivant, quelles sont les fonctions appliquées pour chaque statut et comment est calculé le statut de chaque nœud. Nous appellerons nœud intermédiaire un nœud pour lequel la valeur du statut n'est pas encore calculée.

Calcul des nœuds fils. Après application de la fonction indiquée par le statut du nœud, nous obtenons éventuellement des nouveaux sous-buts. Pour chacun des sous-buts, il faut calculer le nouveau statut. Soit $\langle e, \mathcal{D}, v, s\rangle$ le nœud étudié. Pour chacune des valeurs du statut, nous allons définir les opérations à effectuer sur le nœud, nous verrons comment calculer ses fils et comment calculer les statuts associés aux fils.

– **s =** *SubsCons***.** Ce statut ne peut être associé qu'à un nœud de la forme $\langle X, \mathcal{D}, v, \mathit{SubsCons}\rangle$ (l'expression présente dans le nœud est simplement une variable) afin de pouvoir appliquer la fonction **substitution_constante**. Nous allons tout d'abord commencer par calculer les nœuds fils, puis nous verrons quel est leur statut. Le calcul des fils se fait en plusieurs étapes :
  – Nous appliquons la fonction **substitution_constante**,
  – puis la fonction **équation**,
  – les fonctions **simplification** et **forme_normale**,
  – enfin, la fonction **séparation_branche**.

  Nous obtenons en sortie une liste de nœuds intermédiaires de la forme (le statut n'est pas encore déterminé)

$$\langle e', \mathcal{D}', v, ?\rangle$$

  ainsi qu'un nouveau système. Ce nouveau système sera utilisé pour continuer la construction de l'arbre.

  Pour chacun de ces nœuds intermédiaires, nous devons maintenant effectuer une séparation des opérandes. Pour cela, nous leur associons le statut *SepOp*. Les nœuds sont donc de la forme :

$$\langle e', \mathcal{D}', v, \mathit{SepOp}\rangle$$

– **s =** *SepOp***.** Nous avons encore plusieurs étapes à effectuer avant de pouvoir calculer le statut. Soit N le nœud défini par :

$$\langle e, \mathcal{D}, v, \mathit{SepOp}\rangle$$

  Nous supposons que e est définie de la manière suivante avec $\mathbb{X}_1, \mathbb{X}_2 \in \{\vee, \wedge\}$, $\mathbb{X}_1 \neq \mathbb{X}_2$ et v neutre pour l'opération booléenne $\mathbb{X}_1$:

$$e = \bigmathbb{X}_1^{n}{}_{i=1} e_i$$
$$e_i = \bigmathbb{X}_2^{n'}{}_{i=1} X_i.d_i$$

  – si $n > 1$, nous appliquons la fonction **séparation_opérandes**, nous générons n nœuds intermédiaires de la forme :

$$\left\langle \bigmathbb{X}_2^{n'}{}_{i=1} X_i.d_i, \mathcal{D}, v, ? \right\rangle$$

    En générant n nœuds, chaque nœud est ainsi ramené au cas où n vaut 1. Nous devons maintenant calculer le statut associé à chaque nœud intermédiaire.

– Si $n = 1$, l'expression e est de la forme $\bigwedge\!\!\!\bigwedge_{i=1}^{n'}{}_2\, X_i.d_i$. Pour calculer la valeur du statut, nous allons diviser notre calcul en plusieurs cas :
  – S'il existe j tel que X_j est une variable d'entrée, nous arrêtons la construction et nous associons le statut *VarEnt* au nœud. Sinon, nous distinguerons deux sous-cas suivant la valeur de n'.
  – Si $n' = 1$, l'expression e est alors d'une des trois formes suivantes :
    – $e =$ *True* ou $e =$ *False* : l'expression e est une constante, le statut associé au nœud est alors *VarEnt*.
    – $e = X_i.d_i$: il faut ici appliquer la fonction **dépendance**. Nous associerons le statut *SubsCons* au nouveau nœud.
    – $e = \neg X_i.d_i$: nous commençons par appliquer la fonction **négation** puis la fonction **dépendance**. Nous associerons le statut *SubsCons* au nouveau nœud.
  – Si $n' > 1$, nous sommes dans le cas où la recherche d'auto-dépendance peut être effectuée. Nous commençons par utiliser la fonction **ajout_variable**. Nous notons ici X' la nouvelle variable. Puis, nous regardons la structure de e : s'il n'y a que des occurrences d'une seule variable Y et si la variable Y respecte les critères énoncés dans les théorèmes 5.1 et 5.4 du chapitre 5 alors nous pouvons effectuer une recherche d'auto-dépendance. Le nouveau statut est *RecAuto*, sinon le statut est *EchAuto*. Le nouveau nœud est donc soit de la forme :

$$\langle X', \mathcal{D}, v, \textit{RecAuto} \rangle$$

    si la recherche d'auto-dépendance peut être effectuée, soit de la forme :

$$\langle X_i, \mathcal{D}, v, \textit{EchAuto} \rangle$$

    si cette recherche d'auto-dépendance conduit à une itération infinie. La variable X' est définie par e.
– **s =** ***RecAuto*****.** Nous devons effectuer une recherche d'auto-dépendance sur le nœud $\langle X, \mathcal{D}, v, \textit{RecAuto} \rangle$. Nous appliquons la fonction **recherche_auto_dépendance**. Le nouveau nœud est donc de la forme suivante :

$$\langle X, \mathcal{D}, v, \textit{SubsCons} \rangle$$

Nous avons supposé au moment de l'application de la fonction de séparation des opérandes que l'expression était normalisée. Cette hypothèse est correcte : toutes les expressions du système sont mises au départ sous forme normale conjonctive. Cependant du fait des négations, la forme normale d'une expression n'est pas toujours la forme conjonctive. Il est donc impératif d'utiliser la fonction **forme_normale** avant qu'une séparation des opérandes soit effectuée. Ici la seule possibilité pour effectuer une séparation des opérandes est d'effectuer juste avant une substitution par constante. Pendant cette phase de substitution, nous effectuons une mise sous forme normale.

L'algorithme de construction de l'arbre itère simplement les différentes règles de construction. Cependant si nous nous contentons simplement d'itérer ces règles, la construction est dans certains cas infinie. La section suivante sera donc consacrée à l'étude de la terminaison du processus de construction de l'arbre de preuve.

1.4 Terminaison et validité de la construction

Dans cette section, nous allons tout d'abord montrer sur un exemple qu'utiliser simplement les règles de construction précédentes peut mener à une itération infinie. Puis, nous reviendrons sur la validité de l'algorithme et nous montrerons que l'algorithme ainsi construit peut fournir des contre-exemples.

Terminaison Reprenons l'exemple présenté dans l'introduction (*cf.* chapitre 3 et figure 6.1).

La figure 6.3 représente la première étape de construction de l'arbre de preuve, ainsi qu'une étape de séparation des opérandes.

$$
\begin{aligned}
N_0 &= \langle X, \mathcal{D}, tt, \textit{SubsCons} \rangle \\
N_1 &= \left\langle X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2), \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, \textit{SepOp} \right\rangle \\
N_2 &= \left\langle Y.\delta_1 \vee Y.\delta_2, d_1^{-1}(\mathcal{D}) \cap \mathcal{D}_1 \cap \mathcal{D}, tt, s_1 \right\rangle \\
N_3 &= \left\langle X.d_2, \mathcal{D}_2 \setminus (d_2^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, \textit{SubsCons} \right\rangle \\
N_4 &= \left\langle X, tt, d_2^{-1}(\mathcal{D}) \cap \mathcal{D}_2 \cap \mathcal{D}, tt, \textit{VarEnt} \right\rangle \\
N_5 &= \left\langle X.d_1, \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, \textit{SubsCons} \right\rangle \\
N_6 &= \left\langle W, \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, s_2 \right\rangle
\end{aligned}
$$

où s_1 et s_2 sont des statuts appartenant à $\{\textit{RecAuto}, \textit{EchAuto}\}$.

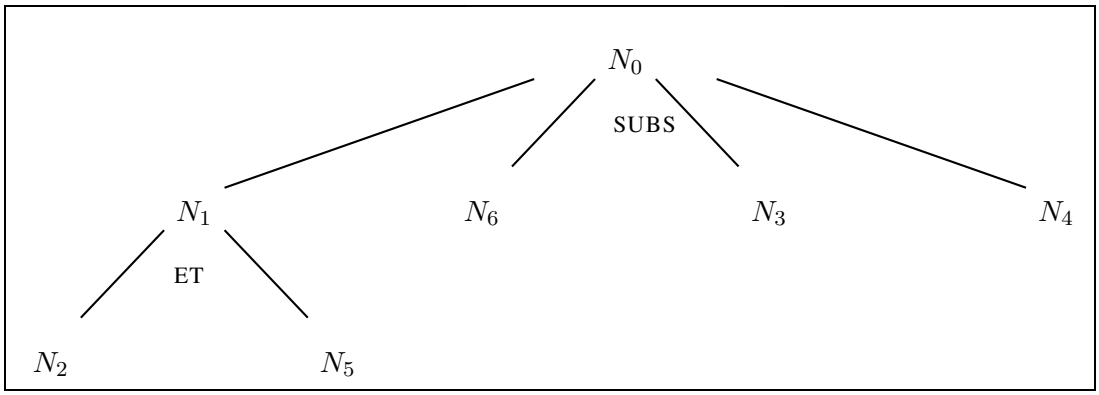


FIG. 6.3 – Début de la construction de l'arbre de preuve de l'exemple 3

Sur le nœud N_5, la substitution par constante est appliquée pour obtenir après séparation des opérandes, les deux nœuds suivants :

$$
\begin{aligned}
N_7 &= \left\langle X.d_1, \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D})) \cap \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, \textit{SubsCons} \right\rangle \\
N_8 &= \left\langle W, \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D})) \cap \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, s_2 \right\rangle
\end{aligned}
$$

Entre les nœuds N_5 et N_7, seul le domaine a changé. En itérant le processus, on construit une suite infinie de nœuds dont seuls les domaines diffèrent. Fixons pour cela les domaines et les dépendances qui suivent :

$$
\begin{aligned}
\mathcal{D} &= \{t, i | 0 \leq t \leq 3N; 0 \leq i \leq \min(2t, 2N, 6N - 2t)\} \\
\mathcal{D}_1 &= \{t, i | t \geq 1, i \geq 1\} \\
d_1 &= (t, i \rightarrow t - 1, i - 1) \\
d_2 &= (t, i \rightarrow t - 1, i)
\end{aligned}
$$

- Le domaine du nœud N_1 et du nœud N_5 est $\{t, i | i = 2t; 2 \leq i \leq 2N\}$.
- Celui du nœud N_7 est $\{t, i | i = 2t + 1; 1 \leq i \leq 2N - 1)\}$.
- En continuant la construction de l'arbre, le domaine du fils gauche de N_7 est $\{t, i | i = 2t + 2; 1 \leq i \leq 2N - 2)\}$.
- La k-ième substitution par constante permet d'obtenir le domaine $\{t, i | i = 2t + k; 1 \leq i \leq 2N - k)\}$.

Comme N est un paramètre, pour toute valeur de k, il existe une valeur de N telle que le domaine $\{t, i | i = 2t + k; 1 \leq i \leq 2N - k)\}$ est non vide. Ainsi, la construction de l'arbre est infinie.

Pour pallier ce problème, nous utilisons un système de traces. Soit $N = \langle e, \mathcal{D}, v, s\rangle$ un nœud, la trace correspond à l'ensemble des nœuds présents sur la branche allant de la racine à N. Soit $\{N_0, \ldots, N_i\}$ cette trace. S'il existe un nœud $N_j = \{e_j, \mathcal{D}_j, v_j, s_j\}$ appartenant à cette trace telle que $e_j = e$ et $v_j = v$, alors le nœud a déjà été rencontré, et continuer la construction de la branche conduit à une construction infinie. On associe à ce nœud un statut d'arrêt mais, pour le démarquer des autres causes d'arrêt, nous fixons la valeur du statut à *EchIter*. Cette valeur n'est associée qu'à des nœuds sur lesquels la règle de substitution par constante peut être appliquée. Si l'arbre contient des feuilles de statut *EchIter*, ces feuilles seront considérées comme pendantes. L'utilisateur devra intervenir pour continuer la construction de l'arbre en agrandissant le domaine de ces feuilles. En effet en agrandissant le domaine, nous supprimons l'itération infinie. Ce problème sera traité dans le chapitre 7.

Avant de donner l'algorithme, nous donnons la définition des nœuds semblables.

Définition 6.7 *Nous dirons que $N = \langle e, \mathcal{D}, v, s\rangle$ est semblable à un nœud d'une liste de nœuds L, s'il existe un nœud $N' = \langle e, \mathcal{D}', v, s\rangle$ (seul le domaine est différent).*

Algorithme 6.8 *Soit S un système, $N = \langle e, \mathcal{D}, v, s\rangle$ un nœud et L une liste de nœuds.*
L'algorithme de construction de l'arbre issu du nœud N est le suivant :
Si s est différent de VarEnt et de EchAuto
- *si N n'est semblable à aucun nœud de L faire :*
  - *Ajouter N à la liste L,*
  - *Appliquer la ou les fonctions correspondant au statut du nœud N. On obtient les nœuds $N_1, \ldots, N_p$ et le système S'.*
  - *Appliquer récursivement l'algorithme de construction sur le système S', chaque nœud N_i et sur la liste L .*
- *sinon N est semblable à un nœud de L, le statut de N est EchIter.*

Dans le théorème suivant, nous montrons que si nous utilisons l'algorithme avec le système de traces, la construction de l'arbre est finie.

Théorème 6.9 *L'algorithme 6.8 de construction de l'arbre de preuve termine.*

Démonstration : *Considérons une branche de l'arbre, nous savons que toutes les expressions apparaissant dans les différents nœuds de la branche sont distinctes sauf, peut-être, pour la dernière : dans ce cas, la feuille est semblable à un nœud de l'arbre et nous sommes dans un cas d'arrêt.*

Ces expressions peuvent être soit des expressions du système, soit des expressions apparaissant lors de la recherche d'auto-dépendance.

- *Expressions du système : le nombre de sous-expressions du système étant fini, nous ne pouvons utiliser qu'un nombre fini de telles sous-expressions.*
- *Recherche d'auto-dépendance : nous ne l'appliquons que dans le cas où la recherche est finie.*

Nous ajoutons dans les deux cas un nombre fini d'expressions, la construction de l'arbre termine donc.
□

Validité Dans ce paragraphe, nous allons nous intéresser à la validité de l'algorithme ; c'est-à-dire, à montrer que l'arbre construit permet de prouver la formule à sa racine. La validité repose sur le théorème 6.6. Un corollaire important est le suivant qui fournit l'existence d'un contre-exemple.

Corollaire 6.10 *Soit $R = \langle e, \mathcal{D}, tt, s\rangle$ la racine d'un arbre de preuve. S'il existe une feuille de la forme :*

$$\langle \mathit{True}, \mathcal{D}, \mathit{ff}, \mathit{VarEnt}\rangle$$

ou de la forme

$$\langle \mathit{False}, \mathcal{D}, \mathit{tt}, \mathit{VarEnt} \rangle$$

alors il est impossible de construire une preuve pour la formule $\mathcal{D} : e \downarrow \mathit{tt}$ *et nous avons construit un contre-exemple.*

Démonstration : *Comme toutes les règles de preuve sont inversibles, lorsque nous obtenons une contradiction, nous avons obtenons une preuve de la contraposée de la formule. Le contre-exemple s'obtient de la manière suivante : il suffit de prendre le père de cette feuille. Notons* $\langle e, \mathcal{D}, v, s \rangle$ *ce père. En générant la feuille, nous avons calculé la valeur de l'expression* e *et cette valeur est en contradiction avec la valeur* v*. L'expression* e *sur le domaine* $\mathcal{D}$ *est un contre-exemple.* ❑

Ce corollaire aura une autre utilité : il pourra être utilisé pour effectuer des démonstrations par l'absurde, ce que nous verrons dans le chapitre 9.

1.5 Conclusion

Cette première implémentation a été utilisée pour prouver l'exemple du DLMS présenté dans le chapitre 8 section 1. Cependant, très rapidement, cette implémentation s'est montrée limitée. L'utilisateur devait intervenir régulièrement pour permettre aux preuves de terminer. Dans cette implémentation, les branches de l'arbre sont toutes construites de manière indépendante. Pour illustrer ces propos, prenons l'exemple suivant :

Exemple 6.11

$$\begin{array}{rcl} W & = & \{t, i \mid t > 0; i > 0\} : X.(t, i \rightarrow t-1, i-1) \wedge Y.(t, i \rightarrow t, i) \\ Y & = & \{t, i \mid t > 0; i > 0\} : X.(t, i \rightarrow t-1, i) \vee Z.(t, i \rightarrow t-1, i) \\ X & = & \left\{ \begin{array}{l} \{t, i \mid t = 0\} : \mathit{True} \\ \{t, i \mid t > 0; i \geq 0\} : X.(t, i \rightarrow t-1, i) \end{array} \right. \end{array}$$

Le but est ici de prouver la formule

$$\{t, i \mid t > 0; i > 0\} : W \downarrow \mathit{tt}$$

La règle de séparation des opérandes engendre deux branches :

- *la première concerne la formule* $\{t, i \mid t > 0; i > 0\} : X.(t, i \rightarrow t-1, i)$*. Après application de la règle des dépendances, le nouveau sous-but concerne la formule* $\{t, i \mid t \geq 0; i \geq 0\} : X$*. En continuant la construction de cette branche, nous n'obtenons que des feuilles fermées. Lors de la construction de cette branche, il a été prouvé que* X *était vraie sur tout son domaine.*
- *la seconde branche n'engendre qu'une feuille pendante pour la formule* $\{t, i \mid t > 0; i > 0\} : X.(t, i \rightarrow t-1, i) \vee Z.(t, i \rightarrow t-1, i)$*. Or, si nous pouvions utiliser le fait que* X *est vraie sur tout son domaine, cette feuille serait en fait une feuille fermée.*

Si nous utilisons les informations obtenues lors des constructions des branches, nous pourrons obtenir des constructions plus précises, c'est -à-dire que certaines feuilles potentiellement pendantes pourront être fermées. Pour cela, nous devons modifier légèrement nos règles de preuve et faire intervenir de manière plus importante les connaissances issues du système dans nos preuves.

2 Un nouveau système de preuve

Dans cette section, nous allons fournir un nouveau système de preuve. Les règles que nous définirons s'appuieront sur les règles définies dans le premier système. Nous allons ici accroître l'utilisation de l'information contenue dans le système sur les substitutions qui ont déjà eu lieu et nous allons cette fois-ci utiliser la notion de liste de motifs. La liste de motifs contiendra des formules spécifiant les variables d'entrées ainsi que les motifs rencontrés lors de la construction de l'arbre de preuve.

Reprenons l'exemple présenté en conclusion de la section précédente. Pour transformer la feuille pendante en feuille fermée, nous devons pouvoir utiliser le fait que X vaut vrai sur tout son domaine. Le principe est le suivant. Puisque X vaut vrai sur tout son domaine, nous pouvons substituer toutes les instances de X par vrai. Ainsi, la variable Y se réécrit de la manière suivante :

$$Y = \{t, i \mid t > 0; i > 0\} : \textit{True}$$

Pour prouver la formule concernant W, la construction de l'arbre engendre le sous-but sur la formule $\{t, i \mid t > 0; i > 0\} : Y$, et si Y a été réécrit, la preuve de ce sous-but mène directement à une feuille fermée. Notre règle de substitution par constante devra donc permettre la substitution de toutes les instances de X définies sur le domaine de la formule $\{t, i \mid t > 0; i > 0\} : X.(t, i \rightarrow t - 1, i)$, dans toutes les variables.

Cette modification n'est cependant pas suffisante. Reprenons la règle de preuve concernant la séparation de l'opérande et :

$$\frac{S \vdash \mathcal{D} : e_1 \downarrow \mathit{tt}, \qquad S \vdash \mathcal{D} : e_2 \downarrow \mathit{tt}}{S \vdash \mathcal{D} : e_1 \wedge e_2 \downarrow \mathit{tt}} \text{ ET}$$

Pour construire les arbres de preuve concernant les deux sous-buts engendrés, nous devons utiliser le système du nœud père. Ainsi dans le cas de notre exemple, pour prouver la formule $\{t, i \mid t > 0; i > 0\} : Y \downarrow \mathit{tt}$, nous devons utiliser le système initial S dans lequel la substitution de X par la valeur vraie n'a pas été effectuée. Ainsi, la construction de l'arbre mène à une feuille pendante. Il faut donc pouvoir utiliser le système obtenu après avoir substitué X. L'idée est donc de faire « remonter » dans les nœuds antécédents les systèmes utilisés au niveau des feuilles. La règle de décomposition ET serait alors de la forme suivante :

$$\frac{S \vdash \mathcal{D} : e_1 \downarrow \mathit{tt}, S', \qquad S' \vdash \mathcal{D} : e_2 \downarrow \mathit{tt}, S''}{S \vdash \mathcal{D} : e_1 \wedge e_2 \downarrow \mathit{tt}, S''} \text{ ET}$$

où S' correspond au système obtenu lors de la construction de la dernière feuille de l'arbre de preuve de la formule $\mathcal{D} : e_1 \downarrow \mathit{tt}$.

Les règles que nous présenterons dans la suite seront un peu plus complexes, car elles feront intervenir le contexte.

Nous allons maintenant formaliser ce nouveau système de preuve.

2.1 Les règles de base

Dans cette section nous allons redéfinir toutes les règles de base, et nous donnerons une nouvelle définition de la sûreté des règles.

Nous définissons tout d'abord la notion de liste de motifs. Une liste de motifs est un ensemble de formules étendues. Ces formules étendues correspondent soit aux formules obtenues dans les feuilles pendantes, soit à des hypothèses faites sur les variables d'entrées. Elles pourront être utilisées pour poursuivre la construction de feuilles pendantes contenant des variables d'entrées. Ces listes ne seront utilisées que par une seule règle : la règle de recherche de motif que nous présenterons en section 2.3.

L'existence d'une preuve pour une formule étendue $\mathcal{D} : e \downarrow v$ d'un système S sous la liste de motifs $\mathcal{L}$ est notée

$$S, \mathcal{L} \vdash \mathcal{D} : e \downarrow v, S', \mathcal{L}'$$

où S' est un système et $\mathcal{L}'$ une liste de motifs obtenus à partir des règles de preuve.

Définition 6.12 (règles de preuve étendues aux systèmes et aux listes de motifs) *Dans la suite, e_1, e_2, e désigneront des expressions polyédriques, v une valeur booléenne, $\mathcal{D}$ un domaine, X une variable, S, S', S'' des systèmes et $\mathcal{L}, \mathcal{L}', \mathcal{L}''$ des listes de motifs. Les règles de preuve de base sont définies de la manière suivante :*

– *Axiomes :*

$$\frac{}{S, \mathcal{L} \vdash \mathcal{D} : \mathit{True} \downarrow \mathit{tt}, S, \mathcal{L}}\ \text{AX1} \qquad \frac{}{S, \mathcal{L} \vdash \{\} : e \downarrow v, S, \mathcal{L}}\ \text{AX2}$$

$$\frac{}{S, \mathcal{L} \vdash \mathcal{D} : \mathit{False} \downarrow \mathit{ff}, S, \mathcal{L}}\ \text{AX3}$$

– *Règles de décompositions des opérandes :*

$$\frac{S, \mathcal{L} \vdash \mathcal{D} : e_1 \downarrow \mathit{tt}, S', \mathcal{L}', \quad S', \mathcal{L}' \vdash \mathcal{D} : e_2 \downarrow \mathit{tt}, S'', \mathcal{L}''}{S, \mathcal{L} \vdash \mathcal{D} : e_1 \wedge e_2 \downarrow \mathit{tt}, S'', \mathcal{L}''}\ \text{ET}$$

$$\frac{S, \mathcal{L} \vdash \mathcal{D} : e_1 \downarrow \mathit{ff}, S', \mathcal{L}', \quad S', \mathcal{L}' \vdash \mathcal{D} : e_2 \downarrow \mathit{ff}, S'', \mathcal{L}''}{S, \mathcal{L} \vdash \mathcal{D} : e_1 \vee e_2 \downarrow \mathit{ff}, S'', \mathcal{L}''}\ \text{OU}$$

– *Règle de la négation :*

$$\frac{S, \mathcal{L} \vdash \mathcal{D} : e \downarrow \mathit{ff}, S', \mathcal{L}'}{S, \mathcal{L} \vdash \mathcal{D} : \neg e \downarrow \mathit{tt}, S', \mathcal{L}'}\ \text{NEG1} \qquad \frac{S, \mathcal{L} \vdash \mathcal{D} : e \downarrow \mathit{tt}, S', \mathcal{L}'}{S, \mathcal{L} \vdash \mathcal{D} : \neg e \downarrow \mathit{ff}, S', \mathcal{L}'}\ \text{NEG2}$$

– *Règle de la dépendance :*

$$\frac{S, \mathcal{L} \vdash d(\mathcal{D}) : e : v, S', \mathcal{L}'}{S, \mathcal{L} \vdash \mathcal{D} : e.d \downarrow v, S', \mathcal{L}'}\ \text{DEP}$$

– *Règle de l'équation :*

$$\frac{S, \mathcal{L} \vdash \mathcal{D} : \left\{ \begin{array}{l} \mathcal{D}_1 : e_1 \\ \vdots \qquad \downarrow v, S', \mathcal{L}' \\ \mathcal{D}_n : e_n \end{array} \right.}{S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v, S', \mathcal{L}'}\ \text{EQ} \quad \textit{si} \quad X = \left\{ \begin{array}{l} \mathcal{D}_1 : e_1 \\ \vdots \qquad \textit{est une équation de } S \\ \mathcal{D}_n : e_n \end{array} \right.$$

– *Règle de séparation des branches :*

$$\frac{S, \mathcal{L} \vdash \mathcal{D} \cap \mathcal{D}_1 : e_1 \downarrow v, S_1, \mathcal{L}_1 \quad \ldots \quad , S_{n-1}, \mathcal{L}_{n-1} \vdash \mathcal{D} \cap \mathcal{D}_n : e_n \downarrow v, S_n, \mathcal{L}_n}{S, \mathcal{L} \vdash \left\{ \begin{array}{l} \mathcal{D}_1 : e_1 \\ \vdots \qquad \downarrow v, S_n, \mathcal{L}_n \\ \mathcal{D}_n : e_n \end{array} \right.}\ \text{SEP}$$

– *Règle de l'ajout de variable :*

$$\frac{S', \mathcal{L} \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}'}{S, \mathcal{L} \vdash \mathcal{D} : e \downarrow v, S'', \mathcal{L}'}\ \text{NOUV}$$

où S' est le système S auquel on a ajouté la variable X définie par l'expression e sur le domaine de e.

– *Règle de la simplification :*

$$\frac{S', \mathcal{L} \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}'}{S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}'} \text{ SIMP}$$

Le système S' est le système S dans lequel toutes les constantes ont été supprimées en appliquant récursivement les techniques de simplification vues dans le chapitre 4.

Définition 6.13 (Sûreté) *Soit une règle de preuve,*

$$\frac{\mathcal{L}', S' \vdash \mathcal{D}' : e' \downarrow v', S'', \mathcal{L}''}{\mathcal{L}, S \vdash \mathcal{D} : e \downarrow v, S'', \mathcal{L}''}$$

Cette règle est dite sûre *si et seulement si*

$$\begin{array}{ll} & S' \cong S'' \implies S \cong S'' \\ \wedge & \forall \rho_{S'} \propto S', \forall \rho_S \propto S, \rho_{S'} \cong \rho_S, \models_{\rho_{S'}} \mathcal{D}' : e' \downarrow v' \implies \models_{\rho_S} \mathcal{D} : e \downarrow v \\ \wedge & \mathcal{L} \subset \mathcal{L}' \end{array}$$

Théorème 6.14 *Les nouvelles règles de preuve sont sûres.*

Démonstration : *La démonstration est similaire à celle du chapitre 4. Pour chaque règle, il faut de plus prouver que $S' \cong S'' \implies S \cong S''$ et $\mathcal{L} \subset \mathcal{L}'$.*

– *$\mathcal{L} \subset \mathcal{L}'$: aucune des règles présentées ici ne modifie la liste de motifs. L'inclusion est donc triviale.*
– *$S' \cong S'' \implies S \cong S''$: il suffit de remarquer que $S' \cong S$ en appliquant la définition de $\cong$. Comme $\cong$ est une relation d'équivalence, nous aurons même montré que $S' \cong S'' \iff S \cong S''$.*

❑

2.2 Règle de substitution par constante

Nous allons maintenant définir la règle liée à la substitution par constante. Cette définition sera faite en plusieurs étapes. Nous commençons par modifier légèrement la règle de substitution par constante donnée dans le premier système de preuve. Nous rappelons que dans le premier système de preuve (section 3, chapitre 4), cette règle de substitution était la suivante :

$$\frac{S' \vdash \mathcal{D} : X_{subs} \downarrow v}{S \vdash \mathcal{D} : X \downarrow v} \text{ SUBS}$$

où le système S' contenait toutes les variables de S et leurs définitions ainsi qu'une variable X_{subs}. La première règle que nous donnons permet de continuer la preuve sur la variable X dont la définition dans S' est celle de la variable X_{subs}.

Cette règle est la suivante :

$$\frac{\mathcal{L}, S' \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}'}{\mathcal{L}, S \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}'} \text{ SUBS1}$$

où la variable X du système S' est définie par l'équation :

$$X = \left\{ \begin{array}{l} \mathcal{D}' \setminus (\mathcal{D}' \cap d^{-1}(\mathcal{D}) \cap \mathcal{D}) : g \\ \mathcal{D}_X \setminus \mathcal{D}' : f \\ \mathcal{D}' \cap d^{-1}(\mathcal{D}) \cap \mathcal{D} : g[v/X.d] \end{array} \right. \tag{6.1}$$

si X était définie dans le système S par

$$X = \left\{ \begin{array}{l} \mathcal{D}' : g \\ \mathcal{D}_X \setminus \mathcal{D}' : f \end{array} \right.$$

Le système S' correspond simplement au système S excepté pour la définition de X. Cette règle est sûre : si l'on note X_S (respectivement $X_{S'}$) la variable X du système S (respectivement S') alors, nous montrons que $X_S \cong_{\mathcal{D}_X} X_{S'}$ en utilisant la proposition 4.15 page 63 et la définition de l'équivalence des environnements.

Cependant cette définition n'est pas particulièrement satisfaisante. Nous ne tirons pas parti de l'existence d'une preuve $S' \vdash \mathcal{D} : X \downarrow v$. Ainsi, en reprenant l'exemple 6.11, la variable X n'est pas substituée par sa valeur dans l'expression définissant Y.

La règle que nous allons présenter permet de substituer simultanément toutes les occurrences de X. Cette règle s'énonce exactement comme la précédente (subs1). La différence vient du système S'. Ce système est défini à l'aide de toutes les variables du système S, mais les définitions de ces variables sont modifiées : toutes les occurrences de X dans le domaine $\mathcal{D}$ sont substituées par la constante v. Cette règle est notée :

$$\frac{S' \vdash \mathcal{D} : X \downarrow v, S''}{S \vdash \mathcal{D} : X \downarrow v, S''} \text{ SUBS2}$$

Plus formellement, la variable X du système S' (notée $X_{S'}$) est définie par (pour simplifier l'expression de $X_{S'}$ nous ne considérons qu'une auto-dépendance de X) :

$$X_{S'} = \left\{ \begin{array}{l} \mathcal{D}' \setminus (\mathcal{D}' \cap d^{-1}(\mathcal{D})) : g \\ \mathcal{D}_X \setminus \mathcal{D}' : f \\ \mathcal{D}' \cap d^{-1}(\mathcal{D}) : g[v/X.d] \end{array} \right. \tag{6.2}$$

La définition de cette variable est légèrement différente de celle donnée dans la règle SUBS1. La substitution n'est plus restreinte au seul domaine de la formule étudiée, (la substitution n'était effectuée que dans la définition des instances de X appartenant au domaine de la formule étudiée) mais toutes les instances de X appartenant à ce domaine sont substituées et ce dans la définition de toutes les occurrences de X.

Proposition 6.15

$$\forall \rho \propto S', \models_{\rho_{S'}} \mathcal{D} : X_{S'} \downarrow v \implies X \cong_{\mathcal{D}_X} X_{S'}$$

Démonstration : *La variable $X_{S'}$ est définie par l'équation (6.2). L'équivalence sémantique est évidente pour tous les points z appartenant à $\mathcal{D}_X \setminus (\mathcal{D}' \cap d^{-1}(\mathcal{D}))$ ou pour tous ceux appartenant à $\mathcal{D}$. Pour ceux appartenant à $\mathcal{D}' \cap d^{-1}(\mathcal{D}) \setminus \mathcal{D}$, il suffit d'appliquer le lemme 4.13 qui permet de substituer une expression par une constante. Cette proposition s'étend facilement au cas où X est définie par plusieurs auto-dépendances en appliquant récursivement cette proposition pour chacune des autos-dépendances.* ❑

Substitution de X par une constante dans toutes les variables d'un système. Le même principe s'applique à toute variable définie à l'aide d'instances de X. Soit Y une variable dépendant de X, et d la dépendance entre ces deux variables. Il existe donc un sous-domaine $\mathcal{D}'$ de $\mathcal{D}_Y$ tel que

$$\forall z \in \mathcal{D}', Y[z] \rightarrow X[d(z)]$$

Soit f l'expression définissant Y. On définit la variable $Y_{d,S'}$ par

$$Y_{d,S'} = \left\{ \begin{array}{l} \mathcal{D}_Y \setminus (d^{-1}(\mathcal{D}) \cap \mathcal{D}') : f \\ (d^{-1}(\mathcal{D}) \cap \mathcal{D}') : f[v/X.d] \end{array} \right. \tag{6.3}$$

Alors si $\mathcal{D} : X_{S'} \downarrow v$ est sémantiquement valide, la variable Y et la variable $Y_{d,S'}$ dans laquelle les occurrences de X ont été substituées par v seront sémantiquement équivalentes.

Proposition 6.16 *Soit ρ un environnement sémantiquement correct de S'. Si $\models_\rho \mathcal{D} : X_{S'} \downarrow v$, alors*

$$\models_{\rho_{S'}} \mathcal{D} : X_{S'} \downarrow v \implies Y_{d,S'} \cong_{\mathcal{D}_Y} Y$$

Démonstration : *Il suffit d'appliquer le lemme 4.13 page 63.* ❑

Le système S' est donc défini par les mêmes variables que le système S mais les définitions de ces variables sont celles données par les équations $X_{S'}$ et $Y_{S'}$.

Ce qui nous permet maintenant de prouver la sûreté de la nouvelle règle de substitution.

Proposition 6.17 *La règle de substitution par constante est sûre.*

Démonstration : *La proposition découle de la proposition 6.16.* ❑

2.3 Recherche de motifs

Jusqu'à présent la recherche de motif n'a été utilisée que dans le cadre de la recherche d'auto-dépendance, nous allons maintenant pouvoir définir une règle de preuve spécifique. La règle que nous allons fournir est un peu particulière dans le sens où seuls le système et la liste de motifs seront modifiés par la règle :

$$\frac{S', \mathcal{L}' \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}''}{S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}''} \text{ MOTIF}$$

Sur le domaine $\mathcal{D}$ la variable X est définie par un motif e. Le système S' correspond au système dans lequel toutes les occurrences du motif e ont été substituées par X, puis une substitution par constante de X sur le domaine $\mathcal{D}$ a été effectuée. La liste de motifs $\mathcal{L}'$ contient la liste $\mathcal{L}$ et le motif associé à X.

Cette règle est sûre, il faut prouver que $S \cong S'$. C'est une application du théorème 4.24 du chapitre 4. Il faut de plus prouver que $\mathcal{L} \subset \mathcal{L}'$, ce qui correspond à la définition de $\mathcal{L}'$.

La règle de recherche d'auto-dépendance est définie par :

$$\frac{S', \mathcal{L}' \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}''}{S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v, S'', \mathcal{L}''} \text{ AUTO}$$

où S' est le système S dans lequel la définition de la variable X est substituée par la définition récursive obtenue par la recherche d'auto-dépendance et $\mathcal{L}'$ contient la liste $\mathcal{L}$ et le motif associé à X.

3 Vers un algorithme plus performant

Dans cette section, nous allons présenter deux nouveaux algorithmes pour automatiser la construction d'un arbre de preuve. Ces algorithmes sont fondés sur les règles de preuves définies dans la section précédente. Le premier algorithme que nous présenterons est très proche de l'algorithme 6.8, les fonctions tiennent compte des nouvelles règles et seules les fonctions de substitution par constante et de recherche d'auto-dépendance sont sensiblement changées. Le deuxième algorithme que nous présenterons permet une construction plus précise de l'arbre de preuve et fait intervenir une deuxième itération lors de la construction.

3.1 Implémentation d'un nouvel algorithme de construction d'arbres de preuve

Nous allons donc construire un nouvel algorithme en tenant compte de ces nouvelles règles de preuve.

Ces nouvelles définitions vont légèrement changer la définition de certaines des fonctions associées aux règles, et nous allons ajouter une nouvelle fonction.

- La fonction **substitution_constante_2** : Quand cette fonction est appliquée sur le nœud $\langle X, \mathcal{D}, v, \mathit{SubsCons}\rangle$ toutes les instances de X appartenant au domaine dans toutes les variables du système sont substituées par la constante v. Cette fonction sera utilisée à la place de la fonction **substitution_constante**.
- La fonction **recherche_motif_2** :

$$\textbf{recherche\_motif\_2}(\langle X, \mathcal{D}, v, \mathit{SubsCons}\rangle, S, \mathcal{L}) = \left(\langle X, \mathcal{D}, v, \mathit{SubsCons}\rangle, S', \mathcal{L}'\right)$$

  Le motif défini par e est recherché dans tout le système S et substitué par X. Nous obtenons le système S' et la liste $\mathcal{L}'$.

Nous allons maintenant étudier les opérations effectuées lorsque le statut vaut *RechInv, EchInv*.

Statut *RechInv* La première étape est d'appliquer la fonction **recherche_auto_dépendance**, puis nous appliquons la fonction **recherche_motif**. Le statut associé au nouveau nœud reste *SubsCons*.

Statut *EchIter* Le statut *EchIter* est un cas d'arrêt. Cependant, nous pouvons effectuer une recherche de motif dans le reste du système et substituer le motif par sa valeur. Cette opération permet de simplifier la construction des autres branches.

Algorithme de construction d'arbres de preuve Nous devons tenir compte du fait que lorsqu'une branche d'arbre est construite nous utilisons le système obtenu à la fin de la construction pour continuer les constructions des autres branches.

Algorithme 6.18 *Soit S un système, $N = \langle e, \mathcal{D}, v, s\rangle$ un nœud et L une liste de nœuds. L'algorithme de construction de l'arbre issu du nœud N est le suivant :*

- *Si s vaut VarEnt ou EchAuto retourner le système S.*
- *Sinon,*
  - *si N est semblable à un nœud de la liste L alors associer le statut EchIter et retourner le système S*
  - *Sinon,*
    - *$S' \leftarrow S$,*
    - *Ajouter N à la liste L,*
    - *Appliquer la ou les fonctions correspondant au statut du nœud N. On obtient les nœuds $N_1, \ldots, N_p$.*
    - *Pour i allant de 1 à p faire*
      - *Construire l'arbre issu du nœud N_i en utilisant le système S et la liste L. À la fin de la construction, on obtient le système S' et la liste L'.*
      - *$S \leftarrow S'$, $L \leftarrow L'$.*

Conclusion Les preuves de la terminaison et de validité de cet algorithme sont similaires aux preuves effectuées pour l'algorithme 6.8. L'ajout des motifs ne modifie pas l'algorithme, seule la règle de recherche de motifs a été ajoutée. Cette règle n'est utilisée que sur des feuilles pendantes, elle ne modifie pas la feuille, mais simplement le système en le simplifiant. Le système contient donc moins d'expressions, et nous avons vu que la terminaison de l'algorithme 6.8 reposait sur le fait que le nombre de

sous-expressions du système était fini, comme ici nous diminuons ce nombre, notre algorithme est aussi fini.

Cependant, ces modifications posent problème, car selon l'ordre de construction des fils d'un nœud, l'arbre résultant de cette construction peut contenir des feuilles pendantes. Nous allons donc apporter une nouvelle modification à l'algorithme pour que l'ordre de construction des fils n'intervienne pas dans la présence de feuilles pendantes.

3.2 Itération lors de la construction de l'arbre

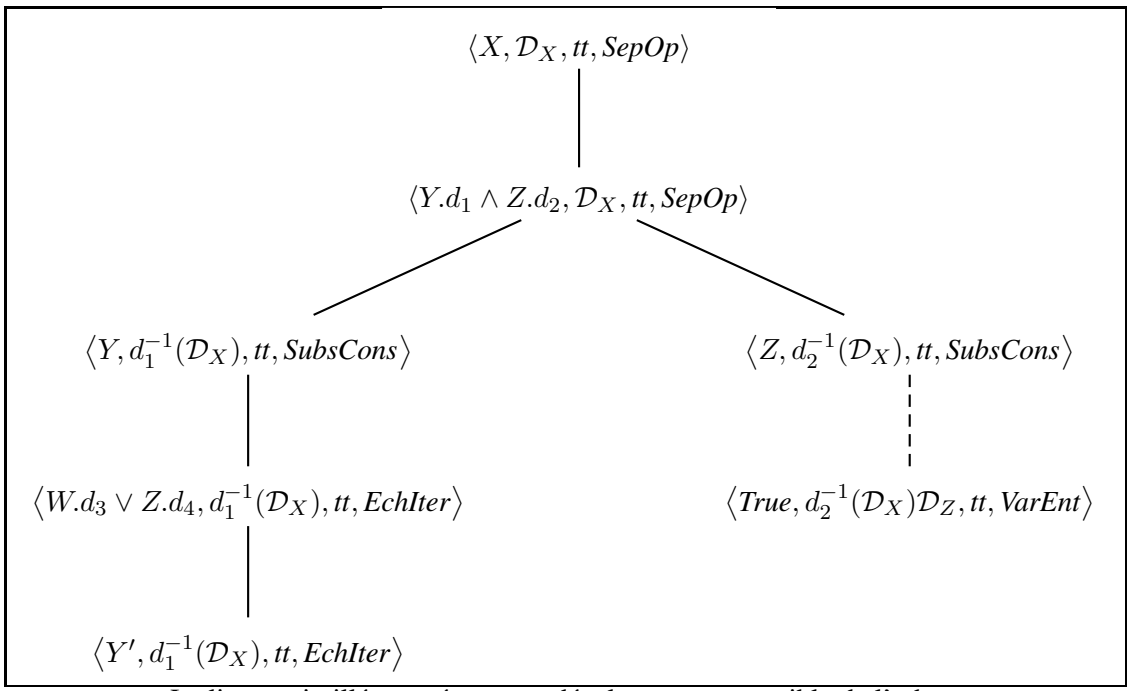
Nous commencerons cette section en illustrant le rôle de l'ordre de la construction des fils d'un nœud.

Exemple 6.19 *Soient X, Y, Z, W quatre variables polyédriques d'un système S. Soient D_X et $\mathcal{D}_Y$ les domaines de définition de X et de Y. On suppose que X et Y sont définies de la manière suivante :*

$$\begin{aligned} X &= Y.d_1 \wedge Z.d_2 \\ Y &= W.d_3 \vee Z.d_4 \end{aligned}$$

Nous voulons prouver que

$$\forall \rho \propto S, \models_\rho \mathcal{D}_X : X$$



La ligne pointillée représente un développement possible de l'arbre.

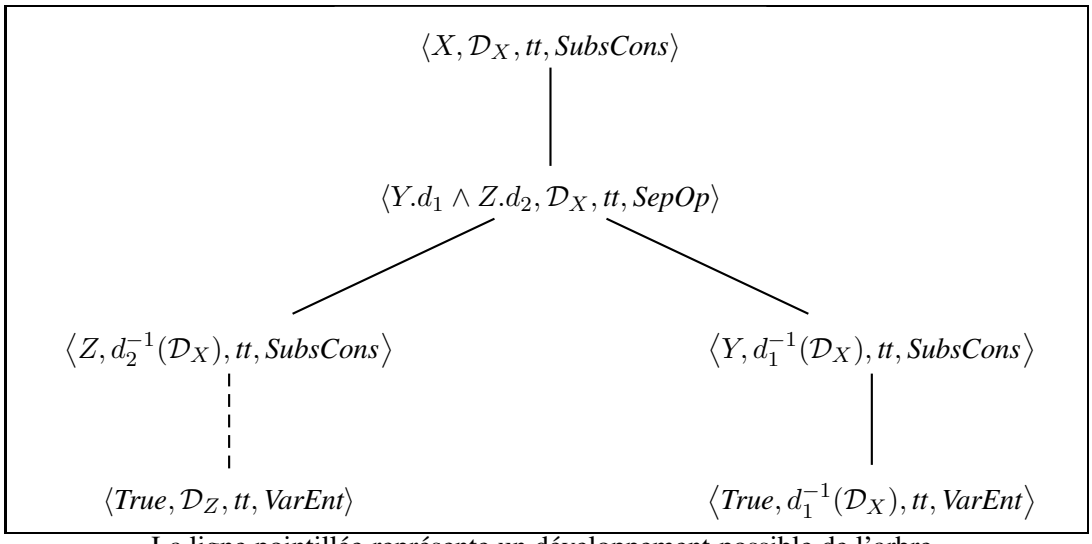
FIG. 6.4 – Construction automatique de l'arbre de preuve en utilisant l'algorithme 6.18

Dans la figure 6.4, nous représentons l'arbre de preuve. Nous supposons que $\mathcal{D}_Z = d_2^{-1}(\mathcal{D}_X)$ et que le sous-arbre construit sur la formule $d_2^{-1}(\mathcal{D}_X) : Z$ ne possède que des feuilles fermées. Nous avons représenté en pointillé une branche de ce sous-arbre dans la figure 6.4 menant à une feuille fermée. Voici la définition des variables X, Y, Z dans le système obtenu

à l'arrêt de la construction de l'arbre :

$$\begin{aligned} X &= Y.d_1 \\ Y &= \mathit{True} \\ Z &= \mathit{True} \end{aligned}$$

Pour une des feuilles de l'arbre de preuve de la formule $\mathcal{D}_X : X \downarrow tt$, nous n'avons pas réussi à finir la preuve. Cependant, si nous regardons l'expression de Y' dans le système qui résulte de la construction de cet arbre, nous voyons que Y' est définie par True : comme nous avons réussi à prouver que Z était vrai sur tout son domaine, les instances de Z ont toutes été substituées par la constante True. Cette information n'était pas connue au moment de la construction du sous-arbre de preuve de la formule $d_1^{-1}(\mathcal{D}_X) : Y \downarrow tt$, la construction s'est donc arrêtée sur une feuille pendante. Si nous avions commencé par construire le sous-arbre concernant la formule $d_2^{-1}(\mathcal{D}_X) : Z \downarrow tt$ puis celui de la formule $d_1^{-1}(\mathcal{D}_X) : Y \downarrow tt$, toutes les feuilles auraient été fermées. Nous avons illustré cet arbre dans la figure 6.5.



La ligne pointillée représente un développement possible de l'arbre.

FIG. 6.5 – Construction automatique de l'arbre de preuve en utilisant l'algorithme 6.18, la branche concernant Z est construite en premier.

La solution que nous proposons est la suivante. Nous commençons par construire l'arbre de preuve. Une fois cet arbre construit, nous avons un nouveau système et nous vérifions si sur les nœuds pendants, il ne serait pas possible de reprendre la construction de l'arbre. Nous itérons le principe jusqu'à ce que l'arbre ne soit plus modifié.

Algorithme 6.20 *Répéter :*

Soient $F_1, \ldots, F_n$ les feuilles pendantes, pour toute feuille pendante faire :
- *si la construction de l'arbre peut reprendre, l'effectuer en utilisant le système S. À l'arrêt de la construction, on obtient le système S'.*
- $S \leftarrow S'$

tant que l'arbre de preuve contient de nouvelles feuilles. Initialement le système S est le système sur lequel la racine doit être prouvée.

Cet algorithme est construit à partir de l'algorithme précédent, cependant nous introduisons une nouvelle itération, il faut donc s'assurer que l'algorithme termine.

Terminaison Comme précédemment si nous utilisons l'algorithme de cette manière, l'itération est infinie. La raison est similaire à celle donnée pour les précédents algorithmes ; nous l'illustrons par l'exemple suivant.

Exemple 6.21 *Soit $F_1 = \langle Z, \mathcal{D}_1, tt, \mathit{EchAuto}\rangle$ une feuille où Z est définie par $X.d_1 \vee Y.d_2$. Supposons que dans la suite de la construction de l'arbre la formule $\mathcal{D}_2 : X \downarrow \mathit{ff}$ soit prouvée. Une fois la construction de l'arbre arrêtée, il faut vérifier s'il est possible de reprendre la construction de l'arbre à partir de la feuille F_1.*

- *si $d_1^{-1}(\mathcal{D}_1) \cap \mathcal{D}_2$ est vide, alors l'expression de la feuille F_1 est identique, le sous-arbre n'est pas modifié,*
- *sinon, deux cas se présentent :*
  - *soit $d_1^{-1}(\mathcal{D}_1) \subset \mathcal{D}_2$, la construction de l'arbre peut continuer et nous obtenons un fils représenté par le nœud $\langle Y, d_2^{-1}(\mathcal{D}_1), tt, \mathit{SubsCons}\rangle$.*
  - *soit $d_1^{-1}(\mathcal{D}_1) \not\subset \mathcal{D}_2$, nous avons dans ce cas deux fils*

$$N_1 = \langle Z, \mathcal{D}_1 \setminus (d_1^{-1}(\mathcal{D}_1) \cap \mathcal{D}_2), tt, \mathit{EchAuto}\rangle$$
$$N_2 = \langle Y, d_2^{-1}((d_1^{-1}(\mathcal{D}_1) \cap \mathcal{D}_2 \cap \mathcal{D}_1), tt, \mathit{SubsCons}\rangle$$

Le nœud N_1 est une feuille pendante. Mais pour le nœud N_2, il est possible de construire un sous-arbre. Soit A_2 ce sous-arbre. Une fois la construction de l'arbre arrêté, il faut vérifier que l'expression de la feuille N_1 n'a pas été modifiée. Supposons que la construction de l'arbre A_2 a permis de prouver la formule $\mathcal{D}_3 : X \downarrow \mathit{ff}$. Nous retrouvons alors le même problème que précédemment et il peut être possible de construire un arbre sur la feuille N_1. En itérant la construction on peut donc construire un arbre infini. Cet arbre est représenté dans la figure 6.6. Nous avons représenté en pointillé les branches contenant les formules $\mathcal{D}_{X,i} : X \downarrow \mathit{ff}$.

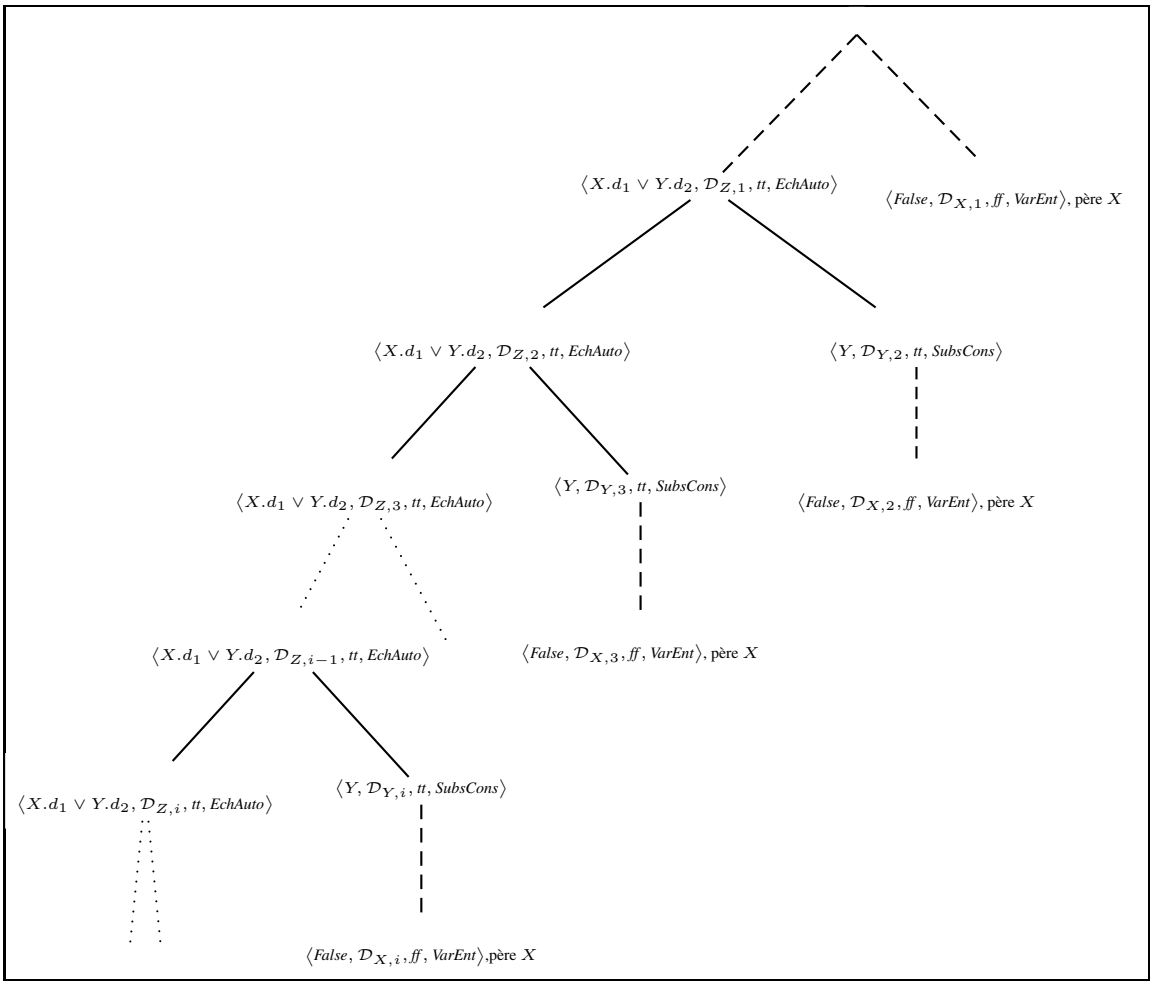
La solution est donc là encore d'utiliser un système de traces. Nous pouvons en fait reprendre la trace utilisée pour la construction de l'arbre de preuve, seul l'algorithme doit être modifié. En effet, nous remarquons dans la figure 6.6, que la feuille pendante $\langle Z, \mathcal{D}_{Z,i}, tt, \mathit{EchAuto}\rangle$ engendre lors de la reprise de la construction la feuille pendante $\langle Z, \mathcal{D}_{Z,i+1}, tt, \mathit{EchAuto}\rangle$. Une étape de construction de l'arbre est finie, mais l'on peut reprendre indéfiniment la construction de l'arbre. La solution est donc d'interdire qu'il y ait deux expressions identiques dans une même branche, et de les marquer d'un statut *EchIter*. Il suffit alors d'interdire la construction sur un nœud de statut *EchIter*.

Théorème 6.22 *L'algorithme 6.20 termine.*

Démonstration : *Nous savons que la construction en profondeur d'un arbre termine. Lorsque nous revisitons un arbre, seules les feuilles de statut EchAuto sont visitées. Soit A un arbre construit sur une telle feuille F. Les feuilles de l'arbre A seront soit :*

- *de statut VarEnt ou EchIter. Pour de telles feuilles, la construction ne pourra pas être continuée plus tard.*
- *soit de statut EchAuto. Dans ce cas, l'expression e d'une telle feuille est différente de l'expression de F. Comme nous avons un nombre fini d'expressions différentes (cf. démonstration du théorème 6.9), la construction s'arrête forcément.*

□



$$\begin{aligned} \mathcal{D}_{Y,i} &= d_1^{-1}(\mathcal{D}_{Z,i-1}) \cap \mathcal{D}_{Z,i-1} \cap \mathcal{D}_{X,i-1} \\ \mathcal{D}_{X,i} &= \mathcal{D}_{X,i-1} \setminus \mathcal{D}_{Y,i} \end{aligned}$$

Pour les feuilles fermées, nous avons précisé le nom de la variable du nœud père. Les traits $- - -$ représentent un développement possible de l'arbre, les traits $\cdots\cdots$ représentent les itérations de la construction de l'arbre.

FIG. 6.6 – Construction d'un arbre infini

3.3 Liste de motifs

Le dernier point que nous aborderons ici concerne encore l'ordre des opérations. Lorsque nous effectuons une recherche de motif m dans le système S_i, nous effectuons la recherche dans les expressions de toutes les variables du système. Cependant, nous avons vu dans les sections précédentes que le système se modifiait au cours de la construction de l'arbre. Supposons que pour le système S_k ($k > i$), une recherche d'auto-dépendance doive être effectuée. Lors de cette recherche, les variables du motif candidat pour être une auto-dépendance sont substituées par leur définition, et l'expression est transformée en forme normale. Cette substitution et cette transformation ont fait apparaître de nouvelles expressions. Comme ces expressions sont nouvelles, elles n'étaient pas présentes dans le système S_i et la recherche du motif m n'y a pas été effectuée. Il faut donc effectuer cette recherche et pour cela, il faut conserver la liste de tous les motifs recherchés. Nous construisons donc une liste de tous les motifs ainsi que leur valeur et leur domaine de validité que nous rencontrons au cours de la construction de l'arbre. Ainsi, lorsque nous obtenons une nouvelle expression, nous recherchons les motifs de la liste. Si ces motifs sont présents dans l'expression, nous les substituons par leur valeur et simplifions l'expression.

4 Conclusion

Nous avons présenté dans ce chapitre l'algorithme utilisé pour vérifier des propriétés. Cette version permet actuellement de prouver un certain nombre de propriétés, mais l'utilisateur doit encore trop souvent intervenir. Pour que l'algorithme termine, nous avons été obligés d'arrêter la construction de certaines branches. Pour ces cas d'arrêt, la construction de l'arbre n'est pas terminée, nous pouvons encore appliquer des règles. L'idée que nous proposons est d'utiliser des techniques d'accélération pour permettre à l'algorithme de continuer la construction de l'arbre de manière finie. De même pour certains cas d'arrêt correspondant aux feuilles de statuts *EchIter*, nous utiliserons des techniques d'accélération. La technique d'accélération que nous allons présenter s'appuie sur les agrandissements de domaine. Ce sera le sujet du chapitre suivant. Cependant, les techniques d'accélération ne pourront s'appliquer que sur certains types de nœuds ; dans les autres cas et en particulier pour les feuilles contenant des variables d'entrées, il faudra utiliser d'autres techniques. Ces techniques passeront entre autres par l'utilisation du contexte, mais aussi par l'intervention de l'utilisateur qui sera amené à fournir un invariant, le but étant que l'utilisateur intervienne le plus rarement possible.

Complexité. Nous allons dire quelques mots de la complexité de cet algorithme. Nous commencerons par détailler la complexité des règles de preuve, puis nous parlerons de la complexité globale de l'algorithme.

- **Substitution par constante.** Cette règle utilise des opérations de calcul de préimages, d'union et d'intersection de domaines. La complexité de ces calculs est selon l'opération effectuée soit linéaire en nombre d'indices, soit linéaire en nombre de contraintes. Il nous faut donc utiliser selon les opérations la représentation des polyèdres par contraintes, ou la représentation géométrique. La librairie polyédrique utilise en fait la double représentation. Ainsi pour chaque opération et sur sur chaque polyèdre, la meilleure représentation est utilisée, l'opération effectuée sur cette représentation puis le résultat transformé dans l'autre représentation. Dans tous les cas, le coût de l'opération est donc exponentiel soit en nombre de contraintes, soit en nombre de dimensions. Nous nous demandons actuellement s'il ne serait pas plus efficace d'utiliser une librairie n'utilisant pas la double représentation.
- **Recherche de motifs – Recherche d'auto-dépendance.** La technique que nous avons implémentée correspond au cas général. Cette règle se décompose en deux parties. Nous avons tout d'abord une partie qui consiste à rechercher toutes les injections d'un ensemble dans un autre, cette opération est donc exponentielle par rapport à la taille du motif. Puis, pour chaque injection, nous

effectuons un calcul de résolution de système d'équations. La technique de résolution implémentée correspond au cas général, et ne prend pas en compte le fait que nous avons des dépendances uniformes : le coût de cette résolution est ainsi cubique en nombre d'indices. Cependant, nous pouvons très facilement simplifier nos calculs en utilisant les différentes heuristiques vues dans le chapitre 5, la complexité deviendrait ainsi linéaire en nombre d'indices.

- **Les autres règles de preuve.** La complexité des autres règles de preuve est constante excepté pour la règle de la dépendance qui est exponentielle en nombre de contraintes car nous effectuons un calcul de préimage (le calcul des préimages est linéaire en nombre de contraintes, mais il faut ensuite transformer le résultat sous forme géométrique).
- **Agrandissement de domaines.** La complexité de l'opérateur d'agrandissement est là encore exponentielle en nombre de dimensions à cause de la double représentation des polyèdres.

Nous n'avons pas effectué une étude approfondie de la complexité des algorithmes, nous nous contentons ici de donner une idée intuitive de cette complexité : elle semble être quadratique par rapport aux nombre de sous-termes du système que nous notons n. Si nous étudions le premier algorithme où l'on ne revient pas dans la construction de l'arbre, la complexité de cette construction est donc au pire linéaire par rapport aux nombres de feuilles de cet arbre. Le nombre de feuilles dans l'arbre est au plus égale à n le nombre de sous-termes du système. Ainsi, cet algorithme est au pire linéaire par rapport au nombre de sous-termes du système. Pour le deuxième algorithme, nous reprenons la construction de l'arbre jusqu'à obtenir soit des feuilles fermées soit rencontrer une seconde fois une même expression. Nous pouvons donc rencontrer au plus deux fois chaque sous-termes du système. Nous reprenons donc dans le pire des cas n fois la construction de l'arbre dont la complexité est au pire linéaire par rapport à n. La complexité semble donc être au pire quadratique. Cependant, l'implémentation que nous en avons fait est une implémentation naïve qui est exponentielle. Nous considérons donc pour le moment que la complexité de notre algorithme est exponentielle.

Malgré les cas d'arrêt encore fréquents et une complexité exponentielle, cet algorithme a été appliqué avec succès sur des exemples concrets. Nous étudierons ces exemples dans le chapitre 8.

Chapitre 7

Accélération de la construction des preuves : utilisation d'un opérateur d'agrandissement

Dans ce chapitre, nous allons nous intéresser aux feuilles pendantes obtenues à l'arrêt de la construction d'un arbre de preuve, c'est-à-dire lorsque plus aucune ne peut être appliquée. Nous allons fournir des heuristiques pour tenter de reprendre la construction de l'arbre sur ces feuilles. Rappelons tout d'abord les différentes valeurs possibles des statuts associés aux feuilles pendantes.

- Statut *EchAuto* : il représente le cas où la recherche d'auto-dépendance ne peut être effectuée,
- Statut *EchIter* : il représente le cas où l'expression étiquetant la feuille a déjà été rencontrée dans la branche allant de la racine à la feuille.

Détaillons maintenant les expressions étiquetant les feuilles pendantes.

- **Feuille de statut** *EchAuto* **:** Lors d'une substitution par constante d'une variable X sur un domaine $\mathcal{D}$, seules les instances de X dans le domaine $\mathcal{D}$ sont substituées par la constante. Ainsi, en continuant la construction de l'arbre, certains sous-buts contiennent encore des instances de X.

  Exemple 7.1 *Soient X et Y deux variables polyédriques. Soit $\mathcal{D}_X$ le domaine de X. La variable X est définie par l'équation suivante :*

  $$X = \begin{cases} \mathcal{D}_1 : X.d_1 \vee Y.d_2 \\ \mathcal{D}_2 : \mathit{True} \end{cases}$$

  Soit $\mathcal{D}$ un sous-domaine de $\mathcal{D}_X$ tel que son intersection avec $\mathcal{D}_1$ est non vide. Supposons que l'on veuille prouver la formule $\mathcal{D} : X \downarrow tt$. La première étape est d'appliquer une substitution par constante, ce qui génère les deux nœuds suivants :

  $$N_1 = \{\mathit{True}, d_1^{-1}(\mathcal{D}_1) \cap \mathcal{D}, tt, \mathit{VarEnt}\}$$
  $$N_2 = \{X.d_1 \vee Y.d_2, (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D})) \cap \mathcal{D}, tt, \mathit{EchAuto}\}$$

  On remarque que le nœud N_2 est une feuille pendante, et qu'il y a encore une occurrence de X. L'idée est ici de se dire que X est peut-être vrai sur un domaine plus grand que le domaine $\mathcal{D}$, plus précisément sur le domaine $(\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}_1)) \cap \mathcal{D}$.

  Nous allons développer pour de telles feuilles, des heuristiques particulières. L'idée est d'agrandir le domaine $\mathcal{D}$ pour que toutes les instances de X dans les sous-buts non résolus soient substituées par une constante. En reprenant l'exemple précédent, le domaine $\mathcal{D}$ est agrandi pour devenir le

domaine $\mathcal{D}'$ tel que $\mathcal{D} \subset \mathcal{D}'$ et pour tout $z \in \mathcal{D}' \cap \mathcal{D}_1$, $d_1(z) \in \mathcal{D}'$ (le point z doit appartenir à $\mathcal{D}_1$ si l'on veut substituer toutes les occurrences de $X.d_1$ par la constante *True*). Après substitution par constante, nous obtenons les deux nœuds suivants :

$$N'_1 = \{\mathit{True}, d_1^{-1}(\mathcal{D}_1) \cap \mathcal{D}', \mathit{tt}, \mathit{VarEnt}\}$$
$$N'_2 = \{X.d_1 \vee Y.d_2, (\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}')) \cap \mathcal{D}', \mathit{tt}, \mathit{EchAuto}\}$$

Le domaine de la feuille N'_2 est vide. En effet, si z appartient à $(\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}')) \cap \mathcal{D}'$ alors :
– z n'appartient pas à $d_1^{-1}(\mathcal{D}')$,
– $d_1(z)$ appartient à $\mathcal{D}'$.
Par définition de $\mathcal{D}'$, z appartient à $d_1^{-1}(\mathcal{D}')$. Ainsi, z appartient à $\mathcal{D}_1 \cap d_1^{-1}(\mathcal{D}')$, ce qui est impossible.
Nous avons donc montré que le domaine était vide, il suffit maintenant d'appliquer sur ce nœud la règle de construction correspondant à l'axiome AX2 (qui concerne les domaines vides) et nous obtenons ainsi le nœud N_2 est transformé en feuille fermée. En agrandissant le domaine, nous avons réussi à transformer une feuille pendante en une feuille fermée.

– **Nœud de statut** *EchIter* **:** Cette méthode d'agrandissement va être aussi utilisée dans le cas où un sous-but a déjà été rencontré. Reprenons l'exemple présenté dans le chapitre d'introduction 3 page 51 et repris dans le chapitre 6. La variable X est définie sur le domaine $\mathcal{D}_X = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{D}_3$ par l'équation suivante :

$$X = \begin{cases} \mathcal{D}_1 : X.d_1 \wedge (Y.\delta_1 \vee Y.\delta_2) \\ \mathcal{D}_2 : X.d_2 \\ \mathcal{D}_3 : \mathit{True} \end{cases}$$

Nous voulons montrer la validité de la formule $\mathcal{D} : X \downarrow \mathit{tt}$ où $\mathcal{D}$ est un sous domaine de $\mathcal{D}_X$. La figure 3.1 page 51 représente ces différents domaines. La construction de l'arbre est présentée dans la section 1.4 du chapitre 6. De plus, dans cette section, nous illustrons le problème de l'itération infinie : si le statut *EchIter* n'était pas associé à la feuille N_5, à chaque étape, nous devrions prouver que X est vraie sur un nouveau domaine : à chaque étape, il faudrait agrandir le domaine de validité $\mathcal{D}$. Nous avons représenté ces agrandissements successifs dans la figure 7.1.

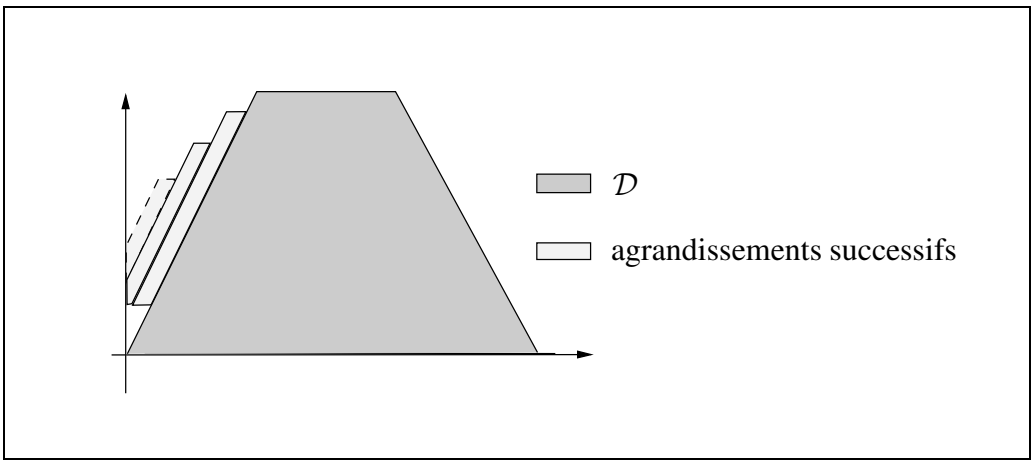


FIG. 7.1 – Agrandissements successifs de $\mathcal{D}$

Comme les domaines sont paramétrés, la construction de l'arbre est potentiellement infinie. Nous allons donc ici accélérer la construction, et pour cela agrandir le domaine de validité pour que la construction converge. Nous avons représenté sur la figure 7.2 un agrandissement possible.

Ces techniques d'accélération et d'agrandissement sont des techniques classiques : nous commencerons donc par un rapide état de l'art sur le sujet dans la section 1, puis dans la section 2 nous définirons

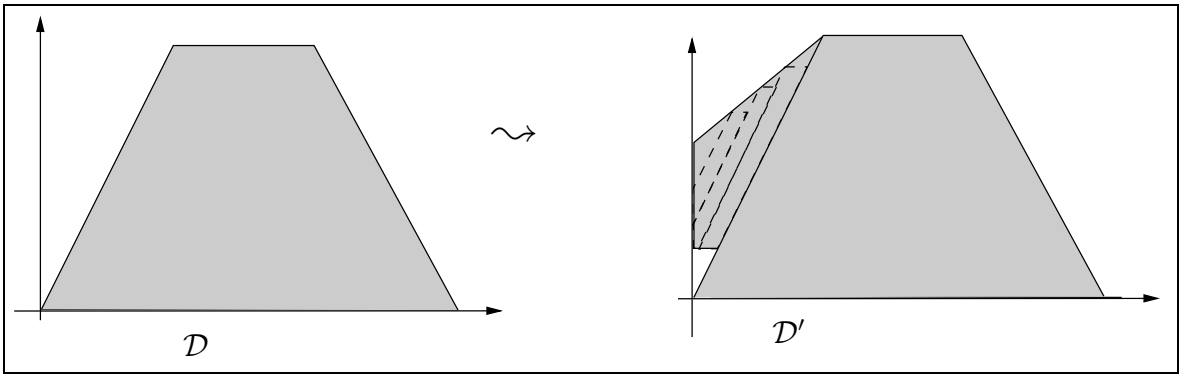


FIG. 7.2 – Agrandissement de $\mathcal{D}$

l'opérateur d'agrandissement que nous utiliserons. Les deux dernières sections seront consacrées à l'utilisation de cet opérateur d'agrandissement. Nous commencerons par appliquer cet opérateur sur des feuilles dont l'expression est une variable ayant une structure particulière et dans la section 4, nous élargirons notre étude à l'utilisation de l'opérateur sur des feuilles dont les expressions sont quelconques.

1 Opérateurs d'agrandissement pour les polyèdres

Cette section s'appuie sur les travaux de Bagnara *et al.* [7]. Depuis les travaux de Cousot et Halbwachs [27] dans le domaine de l'analyse de flots de données sur la recherche automatique de contraintes linéaires sur les variables d'un programme, les polyèdres sont une structure abstraite très souvent utilisée. Ils sont entre autres utilisés dans les domaines suivants :

- Analyse de flots de données : vérification des bornes d'un tableau, calculs d'invariants de boucles, etc.
- Analyse et vérification de langages synchrones [10, 95] : ils représentent les contraintes sur les valeurs des signaux, et permettent de générer des invariants.
- Analyse et vérification formelle d'automates hybrides linéaires [45] : ces automates sont une extension des automates à états finis qui modélisent des systèmes utilisant des variables dont la valeur évolue continûment par rapport au temps. Les polyèdres sont utilisés pour fournir une approximation de l'ensemble de valeurs numériques.

Pour toutes ces applications, l'utilisation de polyèdres conduit à construire des chaînes ascendantes de polyèdres. Ces chaînes peuvent être infinies, comme nous l'illustrons dans la figure 7.3. Dans cette figure, la chaîne (P_i) de polyèdres est infinie, mais elle converge vers le polyèdre P. Un opérateur d'agrandissement permet d'accélérer la convergence de ces chaînes. Ces opérateurs sont nombreux et les définitions peuvent varier. Cet état de l'art ne se voulant pas exhaustif, nous ne donnerons ici que la définition la plus fréquente des opérateurs d'agrandissement. Soit L un ensemble de polyèdres ordonné par l'inclusion $\subseteq$. Un opérateur d'agrandissement noté ∇ est une fonction partielle de $L \times L \to L$ telle que :

1. Pour tout couple de polyèdres X et Y dans L, tels que $X \nabla Y$ est défini, nous avons $X \subseteq X \nabla Y$ et $Y \subseteq X \nabla Y$.
2. Pour toute chaîne croissante $Y_0 \subseteq Y_1 \subseteq \ldots$, la chaîne croissante définie par $X_0 = Y_0, \ldots, X_{i+1} = X_i \nabla Y_{i+1}$ n'est pas strictement croissante.

Cet opérateur fournit une sur-approximation du polyèdre vers lequel la chaîne (P_i) converge. L'opérateur doit avoir deux qualités : la précision et l'efficacité. Il n'est cependant pas toujours possible d'allier les deux : lorsque l'on souhaite accélérer la convergence d'une chaîne, la précision est souvent mise de côté. Cependant, pour certaines applications on préférera un opérateur précis, quitte à avoir une convergence

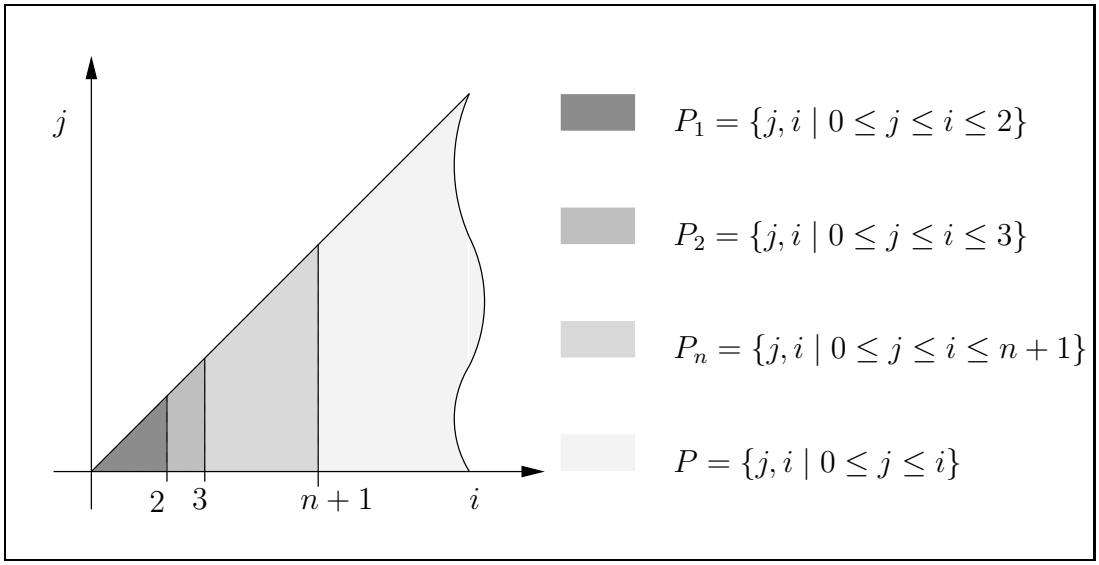


FIG. 7.3 – Chaîne ascendante infinie de polyèdres convergeant vers P.

plus lente. Selon l'application choisie, il faudra trouver un opérateur possédant le juste équilibre. Il peut aussi être utile d'avoir plusieurs opérateurs.

L'opérateur d'agrandissement ∇' de Cousot et Halbwachs [27, 43] est considéré comme l'opérateur standard. De manière intuitive, l'agrandissement de P_2 par P_1 est le polyèdre défini par toutes les contraintes de P_1 qui sont satisfaites par les points de P_2. Cet opérateur est illustré figure 7.4.

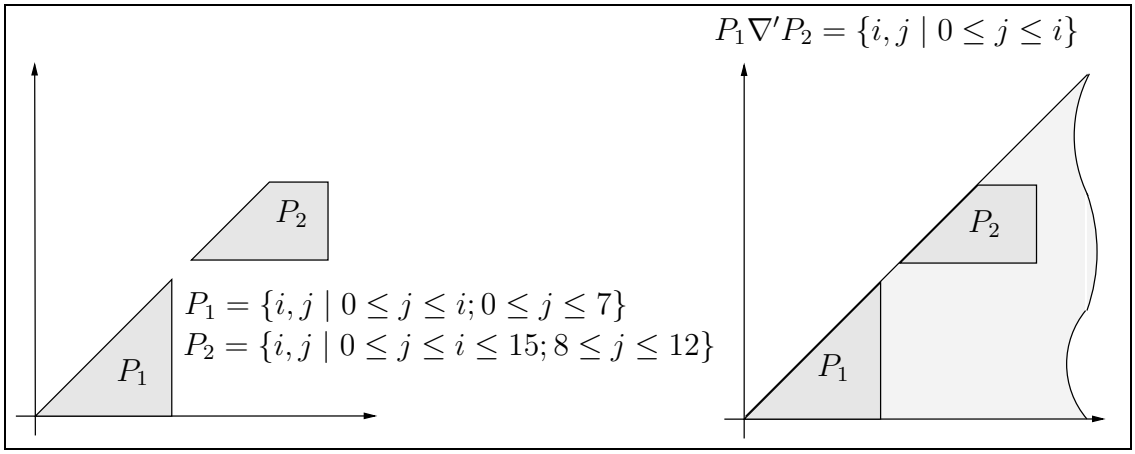


FIG. 7.4 – Opérateur d'agrandissement (remarque :∇' n'est pas symétrique)

D'autres opérateurs concernant la précision ont été proposés par exemple par Henzinger et Ho pour le vérificateur de modèle HYTECH [49, 48]. Enfin, toujours pour les opérateurs d'agrandissement privilégiant la précision, nous citerons les travaux de Bagnara *et al.* [7] qui, à partir d'une idée de Besson *et al.* [10], ont développé une technique permettant de définir de manière systématique et à partir d'opérateurs d'agrandissement déjà existants de nouveau opérateurs au moins aussi précis que ceux existants, mais en pratique plus précis. Cette technique s'appuie sur la définition d'une relation de préordre calculable entre polyèdres et qui respecte la condition de chaîne ascendante, elle est ensuite alliée à plusieurs heuristiques qui améliorent la précision des opérateurs.

2 Choix de l'opérateur d'agrandissement

Nous allons présenter dans cette section, l'opérateur d'agrandissement que nous utiliserons par la suite. Il se distingue des opérateurs présentés dans la section précédente dans le sens où nous n'agrandissons pas deux domaines entre eux mais un domaine dans la direction d'une dépendance. Le domaine agrandi contient le domaine initial ainsi que les images de ce domaine par la composition itérative de la dépendance.

Exemple 7.2 *Dans l'exemple présenté dans le chapitre introductif 3 page 51 à cette partie, la suite de domaines $(\mathcal{D}'_i)$ que nous obtenons est définie de la manière suivante :*
– $\mathcal{D}'_0 = \mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}_1) \cap (\mathcal{D})$
– $\mathcal{D}'_i = d_1(\mathcal{D}_{i-1})$
Nous voulons donc prouver que X est vrai sur $\bigcap_i d_1^i(\mathcal{D}_1 \setminus d_1^{-1}(\mathcal{D}_1) \cap (\mathcal{D})) \cap \mathcal{D}$. À chaque étape, nous avons ajouté l'image de $\mathcal{D}$ par d_1 (en fait seulement la partie appartenant aussi à $\mathcal{D}_1$).

Soit W une variable définie récursivement, soit $\mathcal{D}$ un sous domaine de $\mathcal{D}_W$, soit d une auto-dépendance de W et $\mathcal{D}_1$ le domaine où cette dépendance est utilisée. Le résultat de l'agrandissement du domaine $\mathcal{D}$ par la dépendance d est l'enveloppe convexe de l'ensemble :

$$\{z \mid \exists z_0 \in \mathcal{D}, \exists i \in \mathbb{N},\ z = d^i(z_0)\}$$

Nous avons illustré sur la figure 7.5 le résultat de l'agrandissement du domaine $\mathcal{D}$ par la dépendance d.

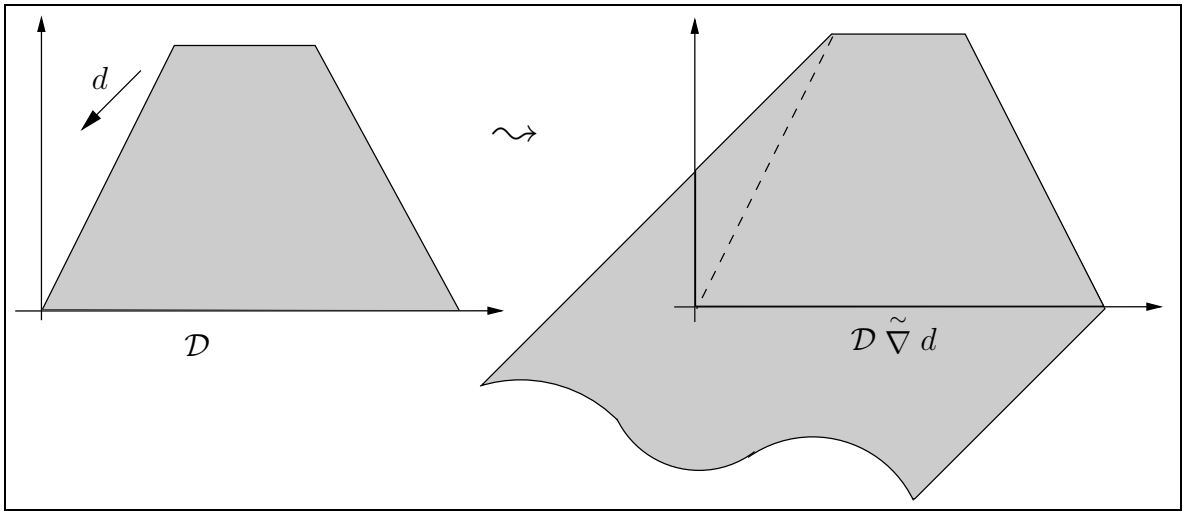


FIG. 7.5 – Agrandissement de $\mathcal{D}$

L'opérateur d'agrandissement se définit donc de la manière suivante :

Définition 7.3 (Agrandissement) *Soit $\mathcal{D}$ un domaine de dimension n et d une dépendance de $\mathbb{Z}^n$ dans $\mathbb{Z}^n$. Nous noterons $\mathcal{D} \widetilde{\nabla} d$, l'agrandissement du domaine $\mathcal{D}$ par la dépendance d. Ce domaine est défini de la manière suivante :*

$$\mathcal{D} \widetilde{\nabla} d = enveloppe.convexe(\{z \mid \exists z_0 \in \mathcal{D}, \exists i \in \mathbb{N},\ z = d^i(z_0)\})$$

Remarque : L'ensemble $\{z \mid \exists z_0 \in \mathcal{D}, \exists i \in \mathbb{N},\ z = d^i(z_0)\}$ est une union infinie de polyèdres. Mais comme d est une dépendance uniforme, nous retrouvons ici un $\mathbb{Z}$-polyèdre (*cf.* annexe D). Pour

obtenir un polyèdre, nous devons donc prendre l'enveloppe convexe. La librairie polyédrique permet de calculer l'enveloppe convexe d'un tel ensemble.

L'implémentation de cet opérateur est faite en utilisant la représentation par sommets, et par rayons : soit $\mathcal{D}$ un domaine et d une dépendance, l'agrandissement $\mathcal{D} \widetilde{\nabla} d$ se fait en ajoutant un rayon dans le direction de d au domaine $\mathcal{D}$.

Nous nous intéressons maintenant à la précision de notre opérateur, c'est-à-dire savoir s'il fournit un domaine sur lequel notre propriété sera valide ou si le domaine est trop grand.

Définition 7.4 *Nous dirons que l'opérateur d'agrandissement est précis pour une variable X d'un système S sur un domaine $\mathcal{D}$ si*

$$\forall \rho \propto S, \models_\rho \mathcal{D} : X \downarrow v \implies \models_\rho \mathcal{D} \widetilde{\nabla} d : X \downarrow v$$

où d est une auto-dépendance de X, sinon nous dirons qu'il fournit une approximation.

L'opérateur d'agrandissement que nous venons de définir n'est pas précis dans le cas général. Nous verrons dans le chapitre suivant sur l'application au produit matrice vecteur que cet opérateur peut fournir des domaines trop grands. Nous allons tout d'abord étudier des signaux sur lesquels l'opérateur d'agrandissement est précis. Puis nous verrons dans le cas général comment allier ces signaux à l'opérateur d'agrandissement pour obtenir des résultats plus précis.

3 Pseudos-pipelines et propagation de valeurs connues

Les signaux que nous allons étudier ici sont appelés pseudos-pipelines. Ils sont la généralisation de la notion de pipeline. Nous commencerons par donner une définition de la notion de pseudo-pipeline, puis nous étudierons les propriétés des pseudos-pipelines et nous verrons les liens avec l'opérateur d'agrandissement.

3.1 Définition

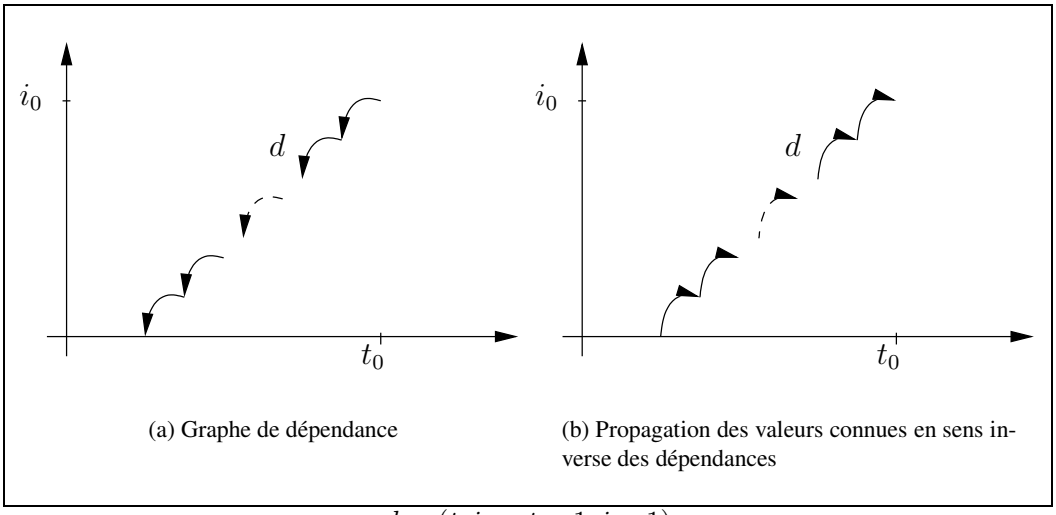
Dans cette section nous nous intéresserons à un type particulier de variables que nous appelons pseudo-pipeline. Ces signaux jouent un rôle particulier dans les démonstrations. Les pseudos-pipelines sont une généralisation de la notion de pipeline : les pipelines sont utilisés pour transmettre des valeurs d'un registre d'une cellule d'un système au registre d'une autre cellule. Dans le modèle polyédrique, les pipelines sont des signaux définis uniquement par une instance d'eux-mêmes. Prenons par exemple la définition suivante d'un pipeline $init$:

$$init = \begin{cases} \{t, i | t \geq 1; N \geq i \geq 1\} : init(t, i \rightarrow t-1, i-1) \\ \{t, i | i = 0\} : \textit{True} \\ \{t, i | 1 \geq i \geq N; t = 0\} : \textit{False} \end{cases}$$

Le signal $init$ permet ici d'initialiser un registre, sa valeur est pipelinée d'une cellule à l'autre d'un réseau. C'est-à-dire qu'une instance de $init$ au point $(t, i) \in \{t, i | t \geq 1; i \geq 1\}$:

- ne dépend que de l'instance de $init$ au point $(t-1, i-1)$ (une seule dépendance, et cette dépendance est une auto-dépendance),
- et sa valeur est celle de l'instance de $init$ au point $(t-1, i-1)$ (pas de modification de la valeur, la valeur est propagée en sens inverse des dépendances).

Nous avons représenté son graphe de dépendance et la propagation des valeurs connues sur la figure 7.6



(a) Graphe de dépendance

(b) Propagation des valeurs connues en sens inverse des dépendances

$$d = (t, i \to t-1, i-1)$$

FIG. 7.6 – Graphe de dépendance et propagation des valeurs connues du pipeline $init$.

Dans le cas des pipelines, la propagation des valeurs se fait tout aussi bien dans les sens des dépendances. Prenons par exemple un pipeline défini de la manière suivante :

$$X = \begin{cases} \{t, i | t \geq 1; N \geq i \geq 1\} : X.(t, i \to t-1, i-1) \\ \{t, i | i = 0\} : Y \\ \{t, i | 1 \geq i \geq N; t = 0\} : \textit{False} \end{cases}$$

Si nous supposons connue la valeur de X au point (t_0, N), nous pouvons en déduire la valeur de X sur tout un domaine en propageant la valeur connue dans le sens des dépendances. La propagation des valeurs connues se fait donc dans le sens des dépendances et dans le sens inverse.

Le pseudo-pipeline est une généralisation du pipeline. Comme dans le cas des pipelines les valeurs sont propagées d'une instance à l'autre. Mais cette fois-ci la propagation ne peut se faire que dans un unique sens : soit dans le sens des dépendances, soit dans le sens inverse. Quant au graphe de dépendance, il est cette fois-ci plus complexe : le pseudo-pipeline peut dépendre d'un autre signal.

L'idée générale est donc de supposer la valeur connue en un point et de la propager soit dans le cas d'un pipeline dans le sens des dépendances et dans le sens inverse des dépendances, soit dans le cas des pseudos-pipelines uniquement dans le sens des dépendances, soit uniquement dans le sens inverse.

Voici un exemple de pseudo-pipeline :

Exemple 7.5

$$Or\_req = \begin{cases} \{t, i | N \geq i \geq 1\} : Or\_req(t, i \to t, i-1) \vee req(t, i \to t, i) \\ \{t, i | i = 0\} : \textit{False} \end{cases}$$

Nous avons illustré en figure 7.7, le graphe de dépendance, et la propagation des valeurs connues dans le sens inverse des dépendances du signal : nous avons supposé que l'instance $req[t_0, i_0]$ *valait vrai, ainsi* $Or\_req[t_0, i_0]$ *est vrai, et d'après la définition de* Or_req*, l'instance* $Or\_req[t_0, i_0 + 1]$ *est vraie. La propagation de la valeur vraie a donc lieu en sens inverse des dépendances. Nous verrons dans la section suivante le cas de la propagation dans le sens des dépendances.*

Nous allons maintenant définir formellement la notion de pseudo-pipeline.

Définition 7.6 *On appelle pseudo-pipeline un signal polyédrique* X *telle que l'une des branches de* X *contient une expression* e *de la forme suivante :*

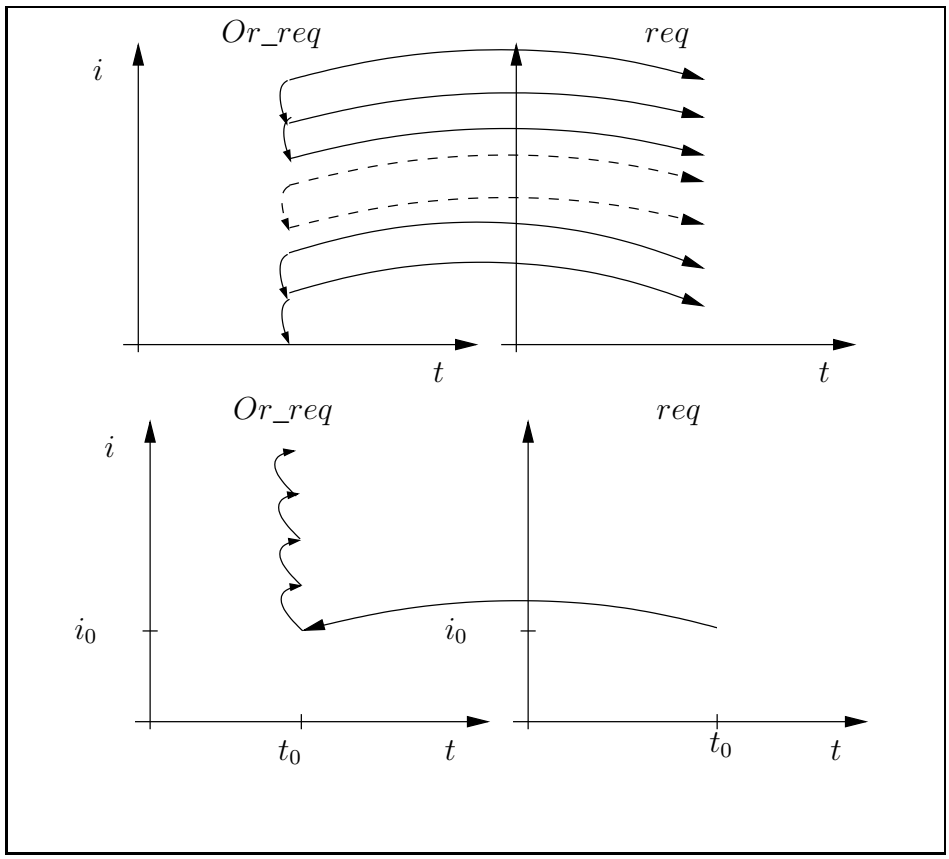


FIG. 7.7 – Graphe de dépendance et graphe de propagation des valeurs du pseudo-pipeline Or_req avec $req[t_0, i_0] = tt$

- *e est sous forme $\barwedge$-normale ($\barwedge \in \{\vee, \wedge\}$)*
- *les opérandes de e sont :*
  - *soit des expressions de la forme $X.d$,*
  - *soit des expressions polyédriques sans occurrence de X.*

Les pseudos-pipelines sont des signaux qui apparaissent souvent dans les programmes ALPHA. Comme nous l'avons dit précédemment, les pseudos-pipelines sont une généralisation de la notion de pipeline. Comme nous travaillons sur des systèmes bas-niveau, les systèmes contiennent beaucoup de signaux pipelinés (en particulier lors de la transformation des dépendances affines en dépendances uniformes de nombreux pipelines sont introduits). De plus, les pseudos-pipelines sont utilisés pour effectuer des conjonctions ou des disjonctions de N instances. C'est le cas du pseudo-pipeline Or_req. Au point(t, N), sa valeur correspond à la disjonction des valeurs de toutes les instances de req à l'instant t :

$$val(\rho(Or\_req))[t, N] = \vee_{i=1}^{N} val(\rho(Or\_req))[t, i]$$

Les pseudos-pipelines se prêtent bien à l'agrandissement de leur domaine. Nous pouvons prédire si le résultat de l'application de l'opérateur d'agrandissement sur un sous-domaine du pseudo-pipeline sera précis (*cf.* section 3.2). Ainsi, nous pouvons dans les cas où le résultat sera précis automatiser le processus d'agrandissement (ce que nous verrons dans la section 4).

3.2 Propriétés des pseudos-pipelines

Le but de cette section est d'étudier la propagation des valeurs dans le cas des pseudos-pipelines en fonction de la valeur propagée et de $\barwedge$ l'opérateur booléen du pseudo-pipeline.

Soit X un pseudo-pipeline défini sur le domaine $\mathcal{D}$ par l'équation suivante :

$$X = \begin{cases} \mathcal{D}_1 : X.d \barwedge e \\ \mathcal{D}_2 : f \end{cases} \tag{7.1}$$

où e et f sont des expressions polyédriques. Sans perte de généralité, nous supposerons que e ne contient aucune occurrence de X (dans le cas contraire, il suffit d'appliquer les différents résultats étudiés dans ce chapitre à chacune des dépendances).

Soit z_0 un point de $\mathcal{D}_1$, nous définissons les domaines $\mathcal{D}_{X,d,z_0}$ et $\mathcal{D}_{X,d^{-1},z_0}$ correspondant respectivement à une propagation dans le sens des dépendances, et à une propagation dans le sens inverse des dépendances.

$$\begin{aligned} \mathcal{D}_{X,d,z_0} &= \{z \mid z \in \mathcal{D}_1 \wedge \exists j \in \mathbb{N},\ z = d^j(z_0)\} \\ \mathcal{D}_{X,d^{-1},z_0} &= \{z \mid z \in \mathcal{D}_1 \wedge \exists j \in \mathbb{N},\ z = d^{-j}(z_0)\} \end{aligned}$$

Les domaines $\mathcal{D}_{X,d,z_0}$ et $\mathcal{D}_{X,d^{-1},z_0}$ sont des Z-polyèdres (*cf.* annexe D). Ces différents domaines sont illustrés sur la figure 7.8.

Proposition 7.7 *Soient S un système et ρ un environnement sémantiquement correct pour S, soit X le pseudo-pipeline défini par l'équation (7.1), soit z_0 un point de $\mathcal{D}_1$, et soit v la valeur de X au point z_0.*

Si $val(\rho(X))[z_0] = v$ alors

$$\text{soit} \quad \models_\rho \mathcal{D}_{X,d,z_0} : X \downarrow v \tag{7.2}$$

$$\text{soit} \quad \models_\rho \mathcal{D}_{X,d^{-1},z_0} : X \downarrow v \tag{7.3}$$

Remarque : Nous avons illustré dans la figure 7.9, les deux expressions (7.2) et (7.3). La première 7.2 correspond à une propagation dans le sens des dépendances, la seconde correspond à une propagation en sens inverse des dépendances (comme celles vues dans les exemples précédents).

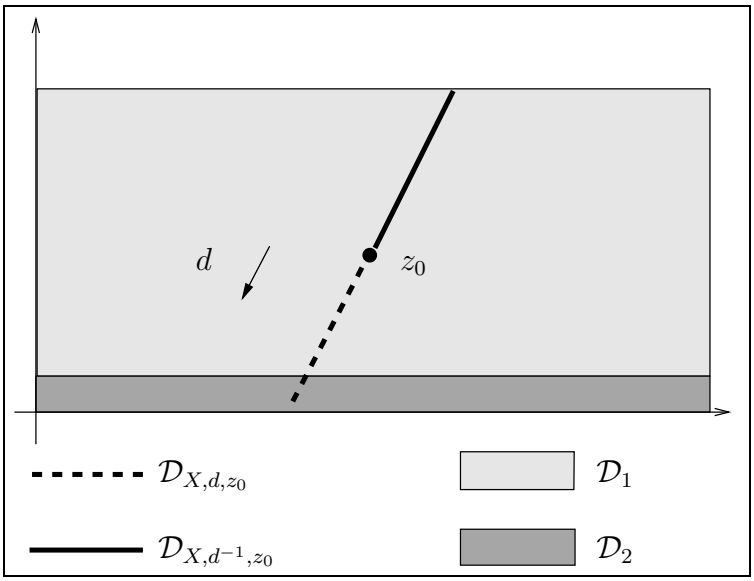
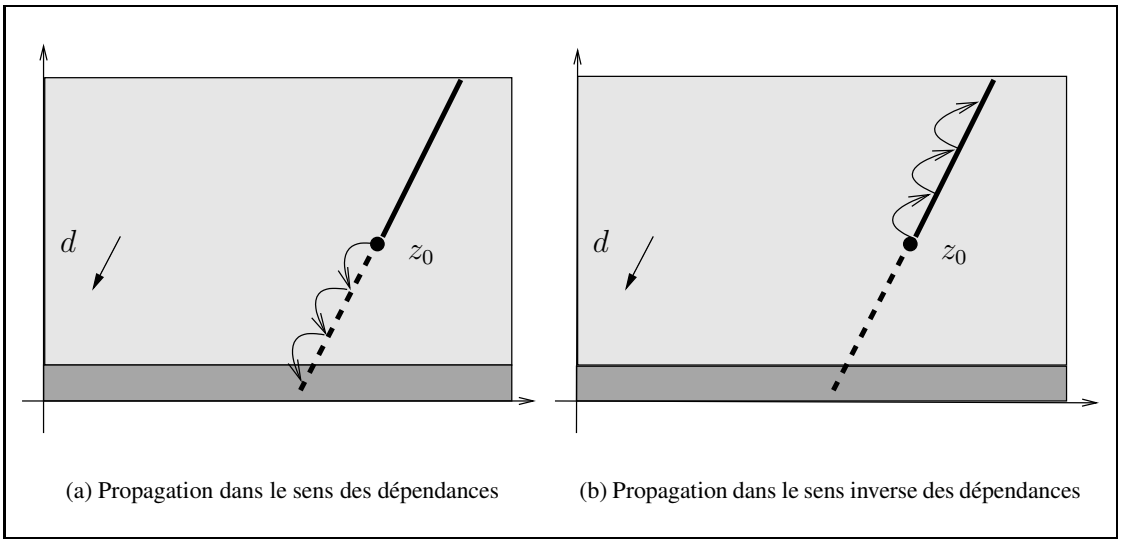


FIG. 7.8 – Représentation des domaines $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_{X,d,z_0}, \mathcal{D}_{X,d^{-1},z_0}$



(a) Propagation dans le sens des dépendances

(b) Propagation dans le sens inverse des dépendances

FIG. 7.9 – Propagation des valeurs.

Démonstration : *Nous séparons notre démonstration en deux. Tout d'abord, nous étudions le cas où v est neutre, puis le cas où v est absorbante.*

- **Premier cas : v neutre.** *Ce cas correspond à la propagation des valeurs dans le sens des dépendances (cf. figure 7.9(a)). Soit z un point de $\mathcal{D}_1$, alors*

$$val(\rho(X))[z] = v \iff val(\mathcal{E}(X.d \Rrightarrow e)(\rho))[z] = v \tag{7.4}$$

En appliquant les règles de définition de la sémantique, et par récurrence sur j on montre que si $val(\rho(X))[z] = v$ alors $val(\rho(X))[d^j(z_0)] = v$ pour tout j appartenant à $\mathbb{N}$. Ce qui revient à montrer $\models_\rho D_{X,d,z_0} : X \downarrow v$.

- **Deuxième cas : v absorbante.** *Ce cas correspond à la propagation des valeurs dans le sens inverse des dépendances (cf. figure 7.9(b)). Soit z un point de $\mathcal{D}_1$, le point $d^{-1}(z)$ est unique et s'il appartient à $\mathcal{D}_1$, en effectuant un changement de base, nous obtenons :*

$$X[d^{-1}(z)] = X[z] \Rrightarrow e[d^{-1}(z)]$$

ainsi,

$$val(\rho(X))[d^{-1}(z)] = v \iff val(\mathcal{E}(X[z] \Rrightarrow e)(\rho))[d^{-1}(z)] = v$$

En appliquant les règles de la sémantique, nous avons :

$$val(\rho(X))[z] = v \implies val(\rho(X))[d^{-1}(z)] = v \tag{7.5}$$

Par récurrence sur j nous montrons alors que si $val(\rho(X))[d^{-1}(z_0)] = v$ alors $val(\rho(X))[d^{-j}(z_0)] = v$ pour tout j appartenant à $\mathbb{N}$, c'est-à-dire $\models_\rho \mathcal{D}_{X,d^{-1},z_0} : X \downarrow v$.

□

De la proposition 7.7, nous allons en déduire quelques corollaires. Mais avant, nous posons la définition suivante :

Définition 7.8 *Nous parlerons de propagation de la valeur neutre, si la propagation des valeurs se fait sur le domaine $\mathcal{D}_{X,d,z_0}$. Sinon, si elle se fait sur le domaine $\mathcal{D}_{X,d^{-1},z_0}$, nous parlerons de propagation de la valeur absorbante.*

La figure 7.9(a) représente une propagation de la valeur neutre, et la figure 7.9(b) représente une propagation de la valeur absorbante.

Dans le cas d'une propagation de valeur neutre, nous ne nous sommes intéressés qu'aux valeurs des instances de X dans la proposition 7.2. Cette proposition correspond, dans le cas de la propagation dans le sens des dépendances, à un agrandissement. Si lors de la construction d'un arbre nous obtenons une feuille correspondant à la formule $\{z \mid z = z_0\} : X \downarrow v$, où v est neutre pour X, l'agrandissement du domaine $\{z \mid z = z_0\}$ correspond simplement au domaine $\mathcal{D}_{X,d,z_0}$.
Cependant cette proposition ne donne aucun résultat sur les expressions e et f. Ce sera donc l'objet de la proposition suivante qui nous permet de déduire les valeurs des expressions de e et de f sur certains domaines si l'on connaît la valeur du pseudo-pipeline en un point.

Proposition 7.9 *Si v est une valeur neutre pour le pseudo-pipeline X alors,*

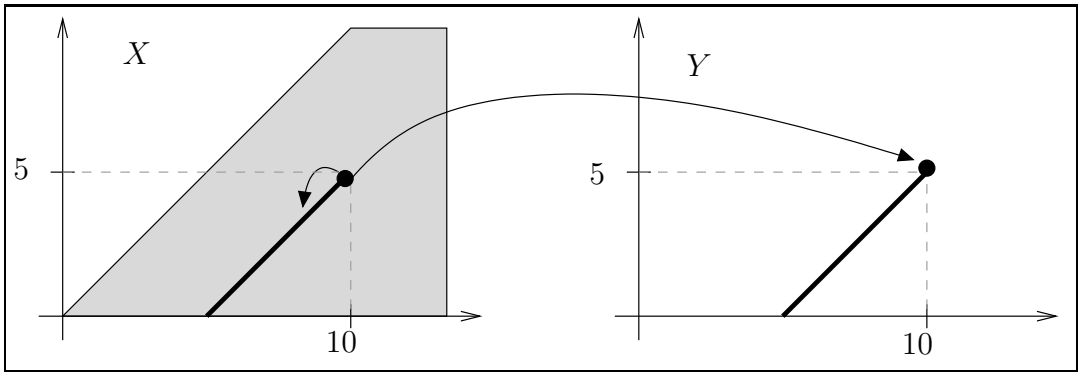
$$\begin{aligned} val(\rho(X))[z_0] = v \iff \quad & \exists z \in \mathcal{D}_2, \exists i \in \mathbb{N}, z = d^i(z_0), \\ & val(\mathcal{E}(f)(\rho))[z] = v \wedge \models_\rho \mathcal{D}_{X,d,z_0} : e \downarrow v \end{aligned}$$

Cette proposition est illustrée dans l'exemple suivant.

Exemple 7.10 *Soient X et Y deux variables polyédriques telles que :*

$$X = \begin{cases} \{t,i \mid t > i > 0\} : X.(t, i \to t-1, i-1) \wedge Y.(t, i \to t, i) \\ \{t,i \mid t = i\} \cup \{t,i \mid i = 0\} : Y.(t, i \to t, i) \end{cases}$$

Le domaine $\mathcal{D}_1$ est représenté par $\{t,i \mid t > i > 0\}$, le domaine $\mathcal{D}_2$ correspond à $\{t,i \mid t = i\} \cup \{t,i \mid i = 0\}$, les expressions e et f correspondent à $Y.(t, i \to t, i)$.
Supposons que $val(\rho(X))[10,5] =$ tt. Comme X est un pseudo-pipeline que $\Join$ est $\wedge$ et que v est tt, nous sommes donc dans le cas de la propagation de la valeur neutre. Nous savons que :
– d'après la proposition 7.7, $\models_\rho \{t,i \mid t = i + 5; 0 \leq t < 10\} : X \downarrow$ tt,
– et d'après la proposition 7.9, $\models_\rho \{t,i \mid t = i + 5; 0 \leq t < 10\} : Y \downarrow$ tt
Nous avons illustré ces domaines sur la figure 7.10.



Les traits en gras représentent les domaines où X et Y valent vrai, et $val(\rho(X))[10,5] =$ tt

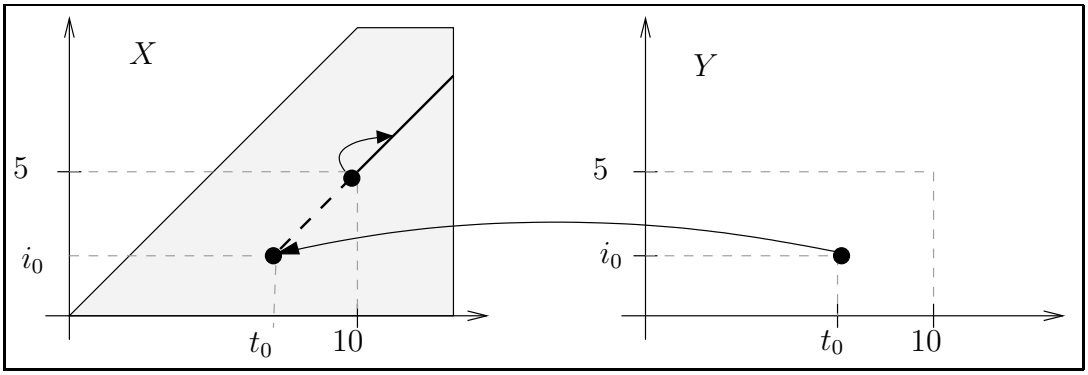
FIG. 7.10 – Propagation de la valeur neutre : étude du domaine $\mathcal{D}_{X,(t,i\to t-1,i-1),(10,5)}$.

Plaçons nous maintenant dans le cas où v est absorbante et supposons que nous ayons une feuille pendante correspondant à la formule $\{z \mid z = z_0\} : X \downarrow v$, cette fois-ci le domaine $\mathcal{D}_{X,d^{-1},z_0}$ de propagation de la valeur ne correspond pas à l'agrandissement obtenu par l'opérateur. L'opérateur d'agrandissement donne le domaine $\mathcal{D}_{X,d,z_0}$. Sur ce domaine nous ne connaissons pas le valeur de X. Le but de l'exemple est d'étudier comment agrandir le domaine dans le sens des dépendances dans le cas d'une valeur absorbante.

Exemple 7.11 *Nous reprenons la définition de X présentée dans l'exemple précédent et nous supposons que avons une feuille pendante correspondant à la formule $F = \{t,i \mid t = 10, i = 5\} : X \downarrow$ ff. Comme X est un pseudo-pipeline, nous sommes donc dans le cas de la propagation de la valeur absorbante. D'après la proposition 7.7, nous savons que si nous réussissons à prouver la formule F alors $\models_\rho \{t,i \mid t = i + 5; t > 10\} : X \downarrow$ ff. Mais comme nous souhaitons obtenir un agrandissement de domaine pour supprimer la feuille pendante, il faut agrandir dans le sens des dépendances. Le but est donc de connaître la valeur de X sur le domaine $\{t,i \mid t = i + 5; 0 \leq t < 10\}$.*
– Il est clair que s'il existe un point (t_0, i_0) appartenant au segment $\{t,i \mid t = i + 5; 0 \leq t < 10\}$ tel que $val(\rho(X))[t_0, i_0] =$ tt alors $\models_\rho \{t,i \mid t = i + 5; 0 \leq t \leq t_0\} : X \downarrow$ tt. Il suffit en effet de propager la valeur neutre tt dans le sens des dépendances sur le segment où t est compris entre 0 et T_0 et i vaut $t - 5$, nous nous retrouvons dans l'exemple 7.10.
– De même, s'il existe un point (t_1, i_1) appartenant à $\{t,i \mid t = i + 5; 0 \leq t < 10\}$ tel que $val(\rho(X))[t_1, i_1] =$ ff alors $\models_\rho \{t,i \mid t = i + 5; t \geq t_1\} : X \downarrow$ ff. Il suffit dans ce cas de propager la valeur absorbante ff dans le sens inverse des dépendances sur la demi-droite engendrée par l'inverse de la dépendance $(t, i \to t-1, i-1)$.

– *Si maintenant, nous supposons que* $(t_1, i_1) = (t_0 + 1, i_0 + 1)$*, comme nous savons que*
  – $val(\rho(X))[t_0, i_0] =$ *tt,*
  – $val(\rho(X))[t_1, i_1] =$ *ff,*
  – $X[t, i] = X[t-1, i-1] \wedge Y[t, i]$ *pour tout* $(t, i) \in \{t, i \mid t > i > 0\}$*,*
  nous pouvons en déduire que $val(\rho(Y))[t_0, i_0] =$ *ff.*

La valeur de X *sur le domaine* $\{t, i \mid t = i + 5; 0 \leq t < 10\}$ *ne dépend que de la valeur de* Y *sur ce même domaine. Ainsi il existe un point* (t_0, i_0) *appartenant à* $\{t, i \mid t = i + 5; 0 \leq t < 10\}$ *tel que* $val(\rho(Y))[t_0, i_0] =$ *ff. Ce que nous avons illustré dans la figure 7.11. Ainsi, si nous souhaitons agrandir le domaine* $\{t, i \mid t = 10; i = 5\}$*, nous devons étudier la valeur de* Y*. Cependant, dans cet exemple, les formules* e *et* f *sont égales : elles correspondent à l'expression* $Y.(t, i \rightarrow t, i)$ *sont égales. Ainsi, nous n'avons ici qu'à étudier la valeur de* Y*. Dans le cas général, il faudrait étudier la valeur de l'expression* e *et celle de l'expression* f*.*



Le trait en gras représente le domaine $\mathcal{D}_{X,(t,i\rightarrow t-1,i-1)^{-1},(10,5)}$, et le trait pointillé représente une extension de ce domaine si $val(\rho(Y))[t_0, i_0] =$ *ff*. Sur ces deux domaines, X vaut faux.

FIG. 7.11 – Propagation de la valeur absorbante : étude du domaine $\mathcal{D}_{X,(t,i\rightarrow t-1,i-1),(10,5)}$.

Nous reprenons dans le corollaire suivant les résultats énoncés dans le cas d'une propagation de valeur absorbante.

Corollaire 7.12 *Si* v *est une valeur absorbante pour le pseudo-pipeline* X *alors,*

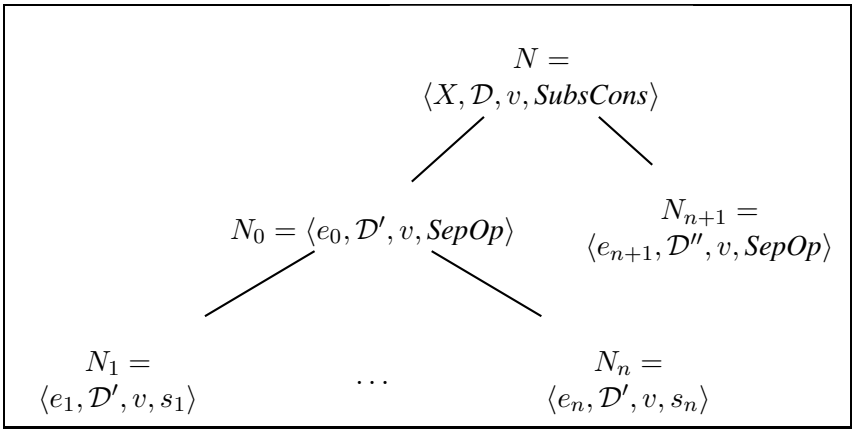
$$\begin{aligned} val(\rho(X))[z_0] = v \implies \quad & (\exists z \in \mathcal{D}_{X,d,z_0},\ val(\mathcal{E}(e)(\rho))[z] = v \\ & \quad \wedge val(\rho(X))[z] = \neg v) \vee \\ & \exists z \in \mathcal{D}_{X,d,z_0} \cap \mathcal{D}_2,\ val(\mathcal{E}(f)(\rho))[z] = v \end{aligned}$$

Nous avons énoncé quelques propriétés sur les pseudos-pipelines, et nous avons vu que ces propriétés permettaient d'agrandir les domaines. Dans le cas d'une propagation de la valeur neutre, l'agrandissement est ici proposé en un point. Or, nous serons intéressés par l'agrandissement d'un domaine. Cela ne pose pas de problème et les résultats sont identiques. Pour obtenir le domaine agrandi, il suffit d'utiliser l'opérateur d'agrandissement $\widetilde{\nabla}$. Dans le cas d'une valeur neutre, l'opérateur d'agrandissement est précis, la démonstration repose simplement sur la définition 7.4 et sur la proposition 7.7.

Cependant dans le cas de la valeur absorbante, le problème de l'agrandissement est plus difficile. Nous ne savons pas jusqu'où agrandir. Nous avons représenté la borne de l'agrandissement à l'aide d'une quantification existentielle dans le corollaire 7.12. De plus, si nous souhaitons agrandir un domaine et non plus agrandir un domaine restreint à un point, les bornes peuvent être différentes d'un point du domaine à l'autre. Dans la section suivante, nous verrons cependant comment utiliser des pseudos-pipelines pour effectuer des agrandissements sur des variables définies par des expressions plus complexes.

4 Heuristiques d'agrandissement

Dans cette section, nous allons voir comment utiliser l'opérateur d'agrandissement. Soit X une variable définie sur un domaine $\mathcal{D}_X$. Nous voulons prouver la formule $\mathcal{D}_X : X \downarrow v$ où v est une constante booléenne. Nous construisons un arbre de preuve. Dans la figure 7.12, nous représentons une partie de cet arbre.



$s_i \in \{\mathit{EchAuto}, \mathit{EchIter}, \mathit{SubsCons}\}$

FIG. 7.12 – Structure partielle d'un arbre contenant des feuilles pendantes parmi les nœuds $\langle e_i, \mathcal{D}', v, s_i \rangle$.

Soient $e_{i_1}, \ldots, e_{i_l}$ les expressions des nœuds $N_{i_1}, \ldots, N_{i_l}$ de l'arbre représenté dans la figure 7.12 contenant des occurrences de X où pour tout j appartenant à $\{1, \ldots, l\}$, i_j appartient à $\{1, \ldots, n\}$. Pour alléger l'écriture, nous supposerons que $i_1 = 1, \ldots, i_l = l$. Nous pouvons donc appliquer sur ces nœuds l'opérateur d'agrandissement. Soient $\barwedge_1, \barwedge_2 \in \{\vee, \wedge\}$ deux opérateurs tels que v est neutre pour $\barwedge_1$ et absorbante pour $\barwedge_2$. Comme la séparation des opérandes vient d'être effectuée, ces expressions sont d'une des deux formes suivantes :

$$\text{pour tout } j \text{ appartenant à } \{1, \ldots, l\} \qquad e_j = X.d_j \quad \text{ou} \quad e_j = X.d_j \barwedge_2 e'_j$$

L'expression e_0 est quant à elle de la forme :

$$e_0 = \bigbarwedge_{i=1}^{n}{}_1 \, e_i$$

Nous notons f_i les formules obtenues à partir des nœuds N_i de l'arbre 7.12 :

$$f_i = \mathcal{D}' : e_i \downarrow v$$

Il nous faut donc prouver les formules $f_i,\ i \in \{1, \ldots, l\}$. Pour chacune de ces formules nous devons agrandir le domaine $\mathcal{D}$ de sorte que la substitution par constante puisse être effectuée. Cependant, il n'est pas possible d'étudier séparément ces expressions : ces expressions dépendent toutes de la variable X. Regardons pour cela l'exemple suivant :

Exemple 7.13 *Soit W une variable polyédrique définie sur un domaine $\mathcal{D}_W$. Soit $\mathcal{D}$ un sous-domaine de $\mathcal{D}_W$. Le but est de prouver la formule $\mathcal{D} : W \downarrow tt$. Nous supposons qu'à la fin de la construction de l'arbre la variable W est définie sur un domaine $\mathcal{D}'$ par l'expression suivante :*

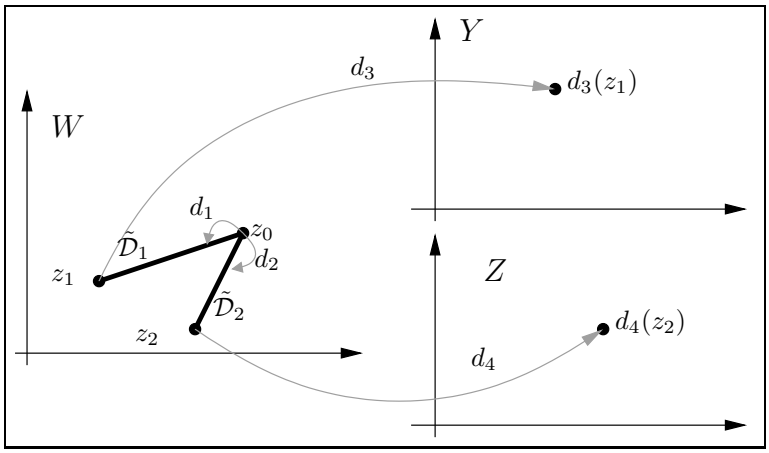
$$(W.d_1 \vee Y.d_3) \wedge (W.d_2 \vee Z.d_4)$$

Cette expression correspond à e_0. Nous notons e_1 l'expression $W.d_1 \vee Y.d_3$ et e_2 l'expression $W.d_2 \vee Z.d_4$. Les expressions e_1 et e_2 contiennent encore des occurrences de W, nous devons donc effectuer un agrandissement de domaine. Commençons par traiter les expressions e_1 et e_2. Pour simplifier le problème, nous supposons que le domaine $\mathcal{D}'$ est réduit à un point z_0 :

$$\mathcal{D}' = \{z \mid z = z_0\}$$

- *Pour l'expression e_1, nous devons agrandir le domaine dans la direction de d_1. Cet agrandissement est délimité par un point z_1 tel que l'instance de Y au point $d_3(z_1)$ est vraie (cf. corollaire 7.12). Nous obtenons alors l'agrandissement $\widetilde{\mathcal{D}}_1$.*
- *De même pour l'expression $W.d_2 \vee Z.d_4$, nous devons agrandir le domaine $\mathcal{D}'$ dans la direction de d_2. Cet agrandissement est délimité par un point z_2 tel que l'instance de Z au point $d_4(z_2)$ est vraie. Nous obtenons cette fois-ci, l'agrandissement $\widetilde{\mathcal{D}}_2$.*

Nous avons représenté sur la figure 7.13 les deux agrandissements.

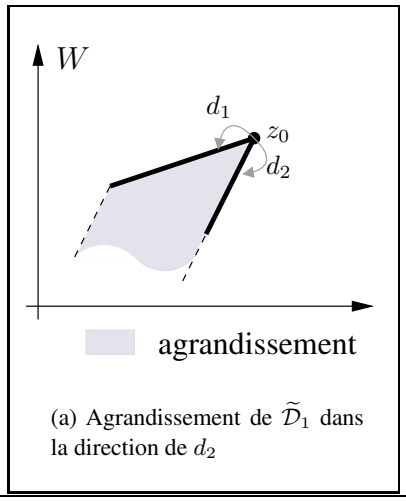


Les points et traits en gras représentent les domaines où les instances des variables W, Y, Z valent vrai.

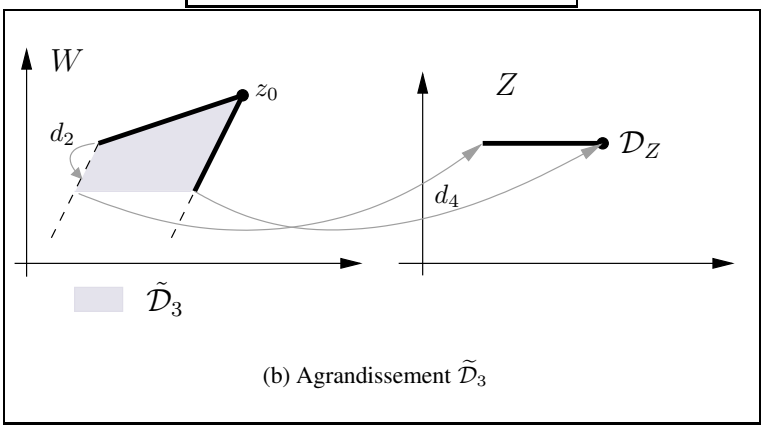
FIG. 7.13 – Agrandissements $\widetilde{\mathcal{D}}_1$ et $\widetilde{\mathcal{D}}_2$ du domaine $\mathcal{D}'$.

Limitons nous pour le moment à l'agrandissement donné par le domaine $\widetilde{\mathcal{D}}_1$ et construisons l'arbre de preuve de la formule $\widetilde{\mathcal{D}}_1 : W \downarrow tt$. Pour tout les points z de ce domaine, nous avons soit $d_1(z)$ qui appartient à $\widetilde{\mathcal{D}}_1$ soit $d_3(z) = d_3(z_1)$, donc l'expression $W.d_1 \vee Y.d_3$ est substituée par la constante tt. En ce qui concerne l'expression $W.d_2 \vee Z.d_4$, la substitution de W ne peut pas se faire car $d_2(z)$ n'appartient pas à $\widetilde{\mathcal{D}}_2$. Nous obtenons donc la formule : $\widetilde{\mathcal{D}}_1 : W.d_2 \vee Z.d_4 \downarrow tt$. Nous devons donc effectuer un agrandissement du domaine $\widetilde{\mathcal{D}}_1$ dans la direction de d_2. Nous avons représenté cet agrandissement dans la figure 7.14(a). Cependant cet agrandissement peut être trop grand, c'est-à-dire que pour certains points z de ce domaine nous pouvons avoir $val(\mathcal{E}(W.d_2 \vee Z.d_4)(\rho))[z] = ff$. De la même manière que nous avons délimité l'agrandissement de $\mathcal{D}'$ dans la direction de d_2 par un point z_2, nous pouvons délimiter cet agrandissement par un ensemble de points tel que les instances de $Z.d_4$ en ces points valent vrai. Nous supposons donc que sur le domaine $\widetilde{\mathcal{D}}_Z$ représenté sur la figure 7.14(b), la variable Z vaut vrai. Sur le domaine $d_4^{-1}(\widetilde{\mathcal{D}}_Z)$, l'expression $W.d_2 \vee Z.d_4$ vaut vrai. Nous pouvons donc réussi à délimité l'agrandissement de $\widetilde{\mathcal{D}}_1$. Nous l'appelons $\widetilde{\mathcal{D}}_3$. Il est représenté sur la figure 7.14(b).

Si maintenant nous utilisons ce domaine pour faire la substitution par constante, toutes les occurrences de $W.d_2$ seront substituées. Mais, sur le domaine $\mathcal{D}_2 \subset d_1^{-1}(\mathcal{D}_2)$ il restera des occurrences de $W.d_1$. Sur la figure 7.15, l'image du point z par la dépendance d_1 n'est pas dans $\widetilde{\mathcal{D}}_3$. L'instance de W en ce point sera vraie seulement si nous agrandissons le domaine



(a) Agrandissement de $\widetilde{\mathcal{D}}_1$ dans la direction de d_2



(b) Agrandissement $\widetilde{\mathcal{D}}_3$

FIG. 7.14 – Deux étapes d'agrandissement

$\widetilde{\mathcal{D}}_3$ *dans la direction de* d_1 *ou si l'instance de* Y *au point* $d_3(z)$ *est vraie.*

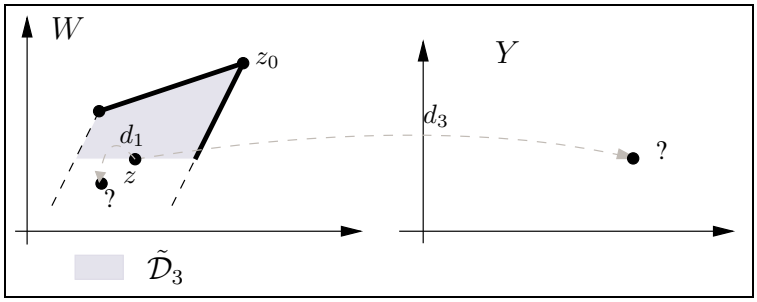


FIG. 7.15 – L'image de z par d_1 n'est pas dans $\widetilde{\mathcal{D}}_3$

Nous devons donc répéter le processus d'agrandissement pour le domaine $\widetilde{\mathcal{D}}_3$ *dans la direction de* d_1*. Cette technique ne converge pas toujours, et quand elle converge ce n'est pas en un nombre fini d'étapes. Dans le cas que nous étudions, nous ne savons pas si elle converge vers un domaine mais nous savons que l'ensemble de convergence sera inclus dans* $((\{z \mid z = z_0\} \mathbin{\widetilde{\nabla}} d_1) \mathbin{\widetilde{\nabla}} d_2)$ *et nous aurons ainsi obtenu une borne supérieure. Nous ne pouvons donc pas étudier séparément les deux sous-expressions de* W *pour effectuer l'agrandissement.*

Comme nous l'avons vu dans cet exemple, il n'est pas possible d'étudier séparément les expressions e_1 et e_2, nous devons donc remonter dans l'arbre de preuve et étudier l'expression e_0 qui correspond à la conjonction des expressions e_1 et e_2.

On pourrait proposer comme première heuristique d'agrandissement, l'agrandissement suivant :

$$\widetilde{\mathcal{D}} = (\mathcal{D} \mathbin{\widetilde{\nabla}} d_1) \ldots \mathbin{\widetilde{\nabla}} d_l$$

Mais cet agrandissement est généralement imprécis. Pour s'en convaincre, il suffit simplement de reprendre l'exemple précédent et de supposer que sur un domaine $\mathcal{D}_2$ tel que $\mathcal{D}_2 \cap ((\{z \mid z = z_0\} \mathbin{\widetilde{\nabla}} d_1) \mathbin{\widetilde{\nabla}} d_2)$ est non vide, la variable W est définie par *False*. L'agrandissement proposé sera trop grand.

Nous allons donc dans la suite essayer de déterminer des heuristiques d'agrandissement plus précises en fonction de la structure de e_0. Cette structure est d'une des trois formes suivantes :

- $e_0 = X.d \bowtie_1 e'$,
- $e_0 = X.d \bowtie_2 e'$,
- $e_0 = (X.d \bowtie_2 e') \bowtie_1 e''$.

4.1 Premier cas : $e_0 = X.d \bowtie_1 e'$

Ce cas correspond exactement à un pseudo-pipeline. La valeur neutre est propagée. Pour tout point z du domaine $\mathcal{D}$, la propagation se fait sur le domaine $\mathcal{D}_{X,d,z}$. Ce qui revient en fait à calculer le domaine $\cup_i d^i(\mathcal{D})$ qui est un $\mathbb{Z}$-polyèdre.

Proposition 7.14 *Si* $\cup_i d^i(\mathcal{D})$ *est un polyèdre, alors*

$$\cup_i d^i \mathcal{D} = \mathcal{D} \mathbin{\widetilde{\nabla}} d$$

Dans ce cas l'opérateur d'agrandissement est précis (cf. définition 7.4).

Dans la pratique, nous supposerons que nous obtenons un polyèdre et nous agrandirons à l'aide de notre opérateur d'agrandissement. Le risque d'utiliser une telle approximation est d'obtenir un domaine trop grand et donc d'obtenir des absurdités dans la suite de la preuve. Ce risque se révèle en pratique très limité mais afin de pouvoir détecter si les absurdités obtenues dans la construction d'une preuve après un agrandissement sont liées à l'agrandissement ou à la propriété à prouver, nous devons reconnaître les cas où nous obtenons une approximation du domaine, c'est-à-dire reconnaître les cas où l'agrandissement est effectivement un polyèdre de ceux où cet agrandissement est normalement un $\mathbb{Z}$-polyèdre. Il suffit pour cela de calculer l'union du domaine à agrandir avec l'image de ce même domaine.

Proposition 7.15 *Si l'union de $\mathcal{D}$ et de son image par la dépendance d est égale à son enveloppe convexe, alors $\cup_i d^i(\mathcal{D})$ sera un polyèdre.*

Démonstration : *La démonstration se fait par récurrence sur i en utilisant la définition des polyèdres dans $\mathbb{Q}$, elle s'appuie sur le fait que les dépendances sont uniformes : ce sont donc des translations.* ❑

Nous représentons sur la figure 7.16 un exemple d'application de cette proposition.

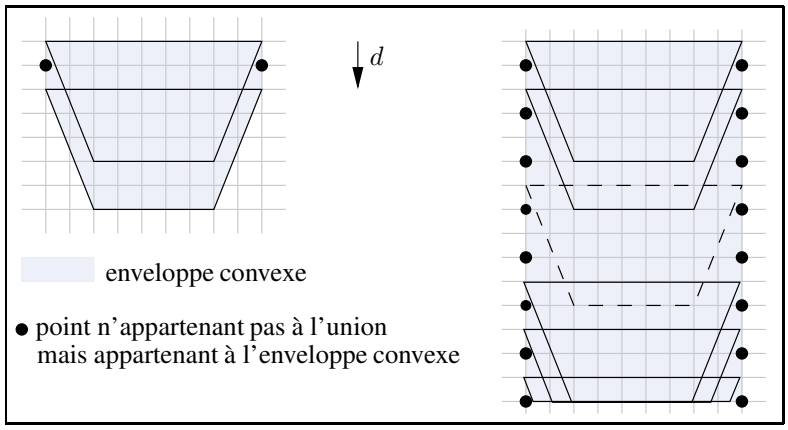


FIG. 7.16 – Application de la proposition 7.15

Dans le cas où nous n'obtenons pas un polyèdre, nous utiliserons tout de même l'opérateur d'agrandissement. Cette approximation ne posera en pratique pas de problème. Si cependant, après un tel agrandissement l'arbre de preuve donnait un nœud faux, nous pourrions toujours revenir sur l'approximation et travailler sur des $\mathbb{Z}$-polyèdres à la place. Il faudrait pour cela modifier l'implémentation de l'opérateur d'agrandissement afin d'obtenir des $\mathbb{Z}$-polyèdres, ce que nous n'avons pour le moment pas étudié.

Ce cas peut être étendu au cas où l'expression e_0 contient plusieurs instances de X.

Proposition 7.16 *Soit e l'expression définie par $\bigvee\!\!\!\bigvee_{k}^{1} X.d_k \between_1 e'$. On note $\widetilde{\mathcal{D}}$ le domaine suivant :*

$$\widetilde{\mathcal{D}} = \{z \mid \exists z_0 \in \mathcal{D}, \exists i_1, \ldots, i_l \in \{1, \ldots, j\},\ z = d_{i_1} \circ \cdots \circ d_{i_l}(z_0)\}$$

C'est à dire que tout point de $\widetilde{\mathcal{D}}$ est l'image d'un point de $\mathcal{D}$ par une combinaison d'auto-dépendances de X. Alors,

$$\models_\rho \mathcal{D} : X \downarrow v \implies \models_\rho \widetilde{\mathcal{D}} : X \downarrow v$$

Démonstration : *Il suffit d'appliquer la proposition 7.7 pour chaque dépendance.* ❑

Le domaine $\widetilde{\mathcal{D}}$ est un $\mathbb{Z}$-polyèdre. Notre première heuristique est donc la suivante :

Heuristique 7.17 *Soient $d_1, \ldots, d_l$ les auto-dépendances et soit $\mathcal{D}$ le domaine. Dans le cas d'un pseudo-pipeline et d'une valeur neutre, l'agrandissement se fait de la manière suivante :*

$$(\ldots((\mathcal{D} \mathbin{\widetilde{\nabla}} d_1) \mathbin{\widetilde{\nabla}} d_2) \mathbin{\widetilde{\nabla}} \ldots) \mathbin{\widetilde{\nabla}} d_l$$

Cet agrandissement peut donc être automatisé (nous l'avons implémenté et il est actuellement utilisé automatiquement).

4.2 Deuxième cas : $e_0 = X.d \mathbin{\mathbb{X}_2} e'$

Nous avons toujours un pseudo-pipeline mais cette fois-ci, c'est la valeur absorbante qui est propagée. La propagation ne se fait donc pas dans le « bon » sens, c'est-à-dire le sens des dépendances. Notre opérateur d'agrandissement ne sera pas précis. Nous pouvons cependant développer des heuristiques pour améliorer son utilisation. D'après le corollaire 7.12, nous savons que nous pouvons agrandir tout de même le domaine, mais nous ne savons pas jusqu'où.

Nous supposons donc X définie par l'équation suivante (nous n'exprimons que les deux branches qui nous intéressent et nous supposons pour le moment une seule occurrence de X dans l'expression e_0) :

$$X = \begin{cases} \mathcal{D}' : e_0 \\ \mathcal{D}'' : f \\ \ldots \end{cases}$$

où $\mathcal{D}''$ est tel que pour tout $z \in \mathcal{D}'$, il existe $z_0 \in \mathcal{D}''$ et il existe $i \in \mathbb{N}$ tels que $z = d^i(z_0)$.

L'agrandissement se fait en deux étapes : nous commençons par un premier agrandissement de $\mathcal{D}'$ dans la direction de la dépendance d et nous notons $\widetilde{\mathcal{D}'}$ cet agrandissement. Puis, nous essayons d'affiner le résultat à l'aide de la stratégie suivante qui consiste à étudier e'. L'expression e' est une $\mathbb{X}_2$-expression (car e_0 est sous forme $\mathbb{X}_1$-normale). Cependant avant d'effectuer cet agrandissement il est utile de regarder l'expression f.

– Étude de f. Si f est la constante absorbante, alors il suffit d'appliquer la proposition 7.7. Comme la constante est absorbante nous la propageons dans le sens inverse des dépendances. Nous obtenons le domaine $\mathcal{D}'' \mathbin{\widetilde{\nabla}} d^{-1}$. Cet agrandissement dans le sens inverse des dépendances contient l'agrandissement $\widetilde{\mathcal{D}'}$. Nous pouvons donc utiliser l'opérateur d'agrandissement qui dans ce cas sera précis puisqu'il est contenu dans le domaine sur lequel s'est propagé la valeur absorbante. Ce cas est assez courant. Nous l'avons illustré dans la figure 7.17. Nous l'utiliserons dans une version un peu plus compliquée dans le chapitre suivant.

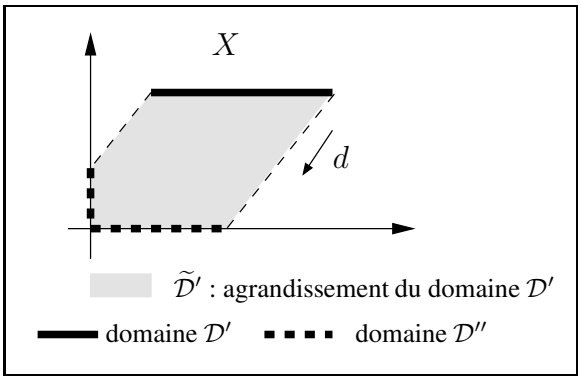


FIG. 7.17 – Agrandissement de domaine dans le cas où l'expression f est la constante absorbante.

– Si l'expression e' fait intervenir au moins un pseudo-pipeline, alors il est possible d'affiner le domaine fourni par l'opérateur d'agrandissement. Notons W ce pseudo-pipeline. L'idée est de se ramener au cas précédent. Nous supposons que l'expression e' est définie par l'expression suivante :

$$e' = W.d' \mathbin{\mathbb{X}_2} e''$$

où e'' est une expression polyédrique.

– Si en un point z la valeur de W est la constante v, alors nous propageons cette valeur suivant la dépendance d' (ou en sens inverse selon la valeur v). Nous obtenons un domaine $\widetilde{\mathcal{D}}_W$ où les instances de W valent la constante v. L'expression e_0 correspond à l'expression $X.d \mathbin{\mathbb{X}_2} W.d' \mathbin{\mathbb{X}_2} e''$, ainsi, si le domaine $d'^{-1}(\widetilde{\mathcal{D}}_W)$ est inclus dans le domaine $\widetilde{\mathcal{D}'}$, nous pouvons simplifier l'expression de X qui vaut alors v. Nous nous retrouvons donc dans le cas précédent. Nous illustrons ce cas dans la figure 7.18.

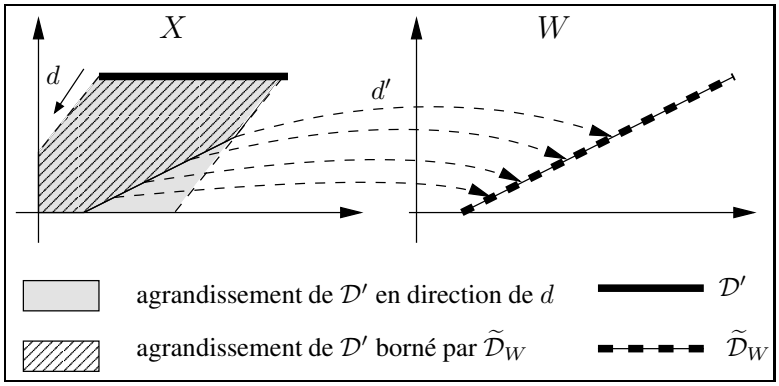


FIG. 7.18 – Agrandissement de domaine dans le cas où l'un des termes de l'expression est un pseudo-pipeline.

En théorie cependant, il est possible que certains points de $\mathcal{D}'$ n'aient pas d'image par une dépendance d^i dans le domaine $d'^{-1}(\widetilde{\mathcal{D}}_W)$. Il faudra donc traiter ces points séparément et appliquer la technique suivante, correspondant au cas où nous ne connaissons pas la valeur de W en un point.

– Sinon, c'est-à-dire si nous ne connaissons pas directement la valeur de W en un point, nous essayons alors d'en découvrir une afin de la propager. Pour cela nous supposons que W vaut v en un point $d'(z)$, où z appartient à $\mathcal{D}' \mathbin{\widetilde{\nabla}} d$ et nous essayons de déterminer la valeur de z. Cette technique s'appuie sur celle de l'ajout des paramètres qui sera présentée dans le chapitre 9.

– Si e' ne contient pas de pseudo-pipeline. Nous ne nous sommes pour le moment pas intéressé à ce problème. L'idée serait cependant d'essayer de découvrir le domaine où l'expression e' vaut la valeur absorbante de X. Il faudrait donc rechercher dans la définition des variables composant X des pseudos-pipelines pour propager les valeurs.

Nous avons jusqu'à présent supposé que l'expression définissant X ne contenait qu'une occurrence de X. Supposons maintenant que l'expression définissant X contient plusieurs occurrences de X, c'est-à-dire que sur le domaine $\mathcal{D}'$, domaine qui doit être agrandi, la variable X est définie par :

$$\mathop{\mathbb{X}}_i{}_2 \, X.d_i \mathbin{\mathbb{X}_2} e'$$

Nous pouvons donc agrandir le domaine de X dans les directions d_i. En un point z il suffit qu'une des instances $X.d_i[z]$ soit vraie pour que l'instance de X au point z soit vrai. Il n'est donc pas nécessaire d'agrandir dans toutes les directions. Il suffit en fait de prendre l'intersection des agrandissements $\mathcal{D} \mathbin{\widetilde{\nabla}} d_i$. Une fois l'intersection calculée nous nous retrouvons dans le cas précédent et nous effectuons

une étude de e' pour obtenir un domaine d'agrandissement plus petit. Dans la figure 7.19, nous avons supposé que X était définie par $X.d_1 \mathbin{\mathbb{X}_2} X.d_2 \mathbin{\mathbb{X}_2} e'$, et nous avons représenté l'intersection des deux agrandissements $\mathcal{D} \mathbin{\widetilde{\nabla}} d_1$ et $\mathcal{D} \mathbin{\widetilde{\nabla}} d_2$. Pour tout point appartenant à cette intersection, soit son image par d_1, soit son image par d_2 est dans l'intersection.

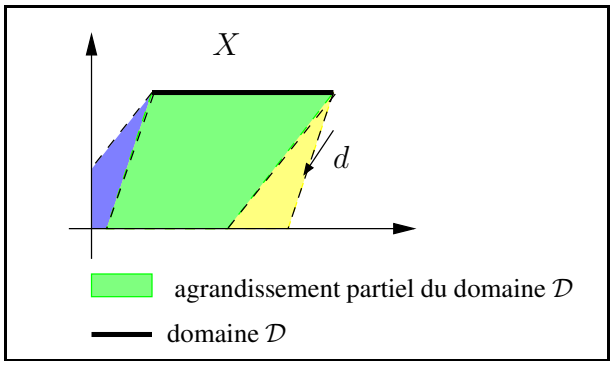


FIG. 7.19 – Agrandissement partiel du domaine.

4.3 Troisième cas : $e_0 = (X.d \mathbin{\mathbb{X}_2} e') \mathbin{\mathbb{X}_1} e''$

Ce dernier cas est le plus complexe. Nous n'avons plus de pseudo-pipeline. C'est le cas illustré dans l'exemple 7.13. Nous décrirons ici la démarche proposée, mais nous ne rentrerons pas dans les détails.

Ce cas se décompose en deux cas dont l'un se ramène au cas précédent.

- Si l'expression e_0 ne contient des occurrences de X que dans l'expression $X.d \mathbin{\mathbb{X}_2} e'$ alors nous pouvons nous ramener au cas précédent (section 4.2). En effet, nous ne pouvons agrandir le domaine que dans une seule direction, celle de d.
- Sinon, ce problème ne s'est encore jamais présenté lors de l'étude des diverses applications. Nous proposons donc une idée générale.
  - Nous commençons par rechercher les pseudos-pipelines et par propager leur valeur. Cette méthode permet de simplifier l'expression.
  - Pour chaque $\mathbb{X}_2$-terme, nous proposons un agrandissement. Soit e' un tel $\mathbb{X}_2$-terme, e' est donc de la forme $\bigtimes_{i}{}_2 X.d_i \mathbin{\mathbb{X}_1} e''$ où e'' ne contient plus d'occurrence de X. L'agrandissement associé au terme e' correspond à l'intersection des images du domaine $\mathcal{D}'$ par les dépendances d_i. Nous calculons alors l'enveloppe convexe de l'union de tous ces agrandissements. Cet agrandissement peut être la encore trop grand, ce sera alors à l'utilisateur de produire un agrandissement plus fin.

5 Conclusion

Dans ce chapitre nous avons étudié une technique permettant soit d'accélérer la construction de l'arbre de preuve dans le cas où nous avions une feuille de statut *EchIter*, soit de reprendre la construction de l'arbre lorsque nous avons une feuille de statut *EchAuto*. Cette technique s'appuie sur les agrandissements de domaine. Comme nous l'avons vu dans la section 1, le problème de l'agrandissement est un problème difficile. Dans ce chapitre, nous avons tout d'abord défini un opérateur d'agrandissement. Cet opérateur consiste en des manipulations de polyèdres (ajout de rayon, intersection), sa complexité est donc celle de ces deux opérations c'est-à-dire exponentielle (soit en nombre de contraintes pour l'ajout de rayon, soit par rapport à la dimension du polyèdre pour l'intersection). Nous avons ensuite présenté plusieurs heuristiques selon la structure de l'expression dont le domaine doit être agrandi. La technique

générale est la suivante : nous commençons par utiliser notre opérateur d'agrandissement pour obtenir une première approximation, puis nous utilisons les différentes heuristiques pour affiner le domaine.

Toutes ces heuristiques reposent sur la même idée qui est :

Repérer les pseudos-pipelines et propager les valeurs.

Nous n'avons pas parlé ici de la méthode pour repérer les pseudos-pipelines. Cela peut se faire automatiquement, sans aucune difficulté. En effet, les pseudos-pipelines ont été définis à l'aide d'un critère syntaxique.

L'opérateur d'agrandissement que nous proposons donne en résultat un polyèdre. Cependant, le résultat attendu est en théorie un $\mathbb{Z}$-polyèdre. Cette approximation mène dans certains cas (rares en pratique) à des constructions de preuves absurdes. Ces constructions contiennent des feuilles fermées non valides, ce qui peut conduire de manière erronée à la conclusion que la propriété vérifiée n'est pas valide. Nous proposons donc un test pour vérifier si l'opérateur fournit une approximation, et si nous obtenons des feuilles non valides lors de la construction, il faut alors fournir un autre agrandissement s'appuyant sur les $\mathbb{Z}$-polyèdres. Cet opérateur n'a pas encore été développé.

Ces techniques d'agrandissement sont donc indispensables dans le processus automatisation. Sans ces techniques, la construction de la preuve échoue. L'utilisateur doit alors intervenir pour fournir pour agrandir lui même les domaines. Dans les exemples que nous avons étudiés et que nous présenterons dans les deux prochains chapitres, ces techniques se sont avérées efficaces. Une fois leur implémentation effectuée elles permettront de prouver automatiquement par exemple le produit matrice-vecteur du chapitre 8.

Chapitre 8

Applications

Dans ce chapitre, nous allons présenter deux applications. La première est un filtre adaptatif paramétré. Nous nous intéresserons à la preuve d'une propriété sur un signal de contrôle introduit automatiquement (mais par une transformation ne préservant pas la sémantique) lors de la synthèse du dispositif. Le second exemple sera un circuit permettant de calculer le produit d'une matrice et d'un vecteur. Là encore, nous travaillerons sur un système paramétré.

1 DLMS

Cette première section est consacrée à l'étude d'un filtre adaptatif avec délai. Nous présenterons tout d'abord le rôle des filtres, puis nous nous intéresserons à la preuve d'une propriété sur un des signaux de contrôle d'un tel filtre.

1.1 Les filtres adaptatifs

Les filtres sont des dispositifs utilisés dans les applications de contrôles et les traitements de signaux : communications, radar, sonar, sismologie, ingénierie biomédicale. Bien que ces applications soient très différentes elles ont toutes en commun les caractéristiques suivantes : l'utilisation d'un vecteur d'entrée et d'un vecteur de « sortie désirée » pour calculer une estimation de l'erreur. Cette estimation est utilisée pour ajuster dynamiquement les poids du filtre.

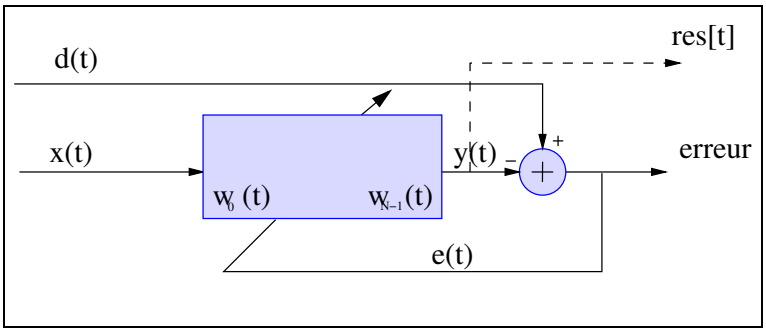


FIG. 8.1 – Filtre adaptatif

Nous ne rentrerons pas dans les détails du fonctionnement du filtre. Haykin [46] a effectué une étude approfondie du sujet. Nous en fournissons une description paramétrée par N, le nombre de poids du filtre.

– $x(t)$ correspond au signal d'entrée à l'instant t,

– $d(t)$ est la sortie désirée à l'instant t,,
– $y(t)$ est la sortie à l'instant t,,
– $e(t)$ correspond à l'erreur à l'instant t,,
– $w_0(t), \ldots, w_{N-1}(t)$ sont les poids du filtre à l'instant t,.

Les signaux sont définis de la manière suivante :

$$\begin{aligned} y(t) &= \sum_{i=0}^{N-1} x(n-i)w_i(t) \\ e(t) &= d(t) - y(t) \end{aligned}$$

Les coefficients w_i sont calculés par la méthode des moindres carrés et dépendent de $e(n)$.

L'objectif du filtre adaptatif (*cf.* figure 8.1) est, comme son nom l'indique, de filtrer le signal $x(t)$ de sorte que ce signal ressemble au signal désiré $d(t)$. Le signal d'erreur $e(t)$ est la différence entre le signal désiré et la sortie $y(t)$. Cette différence est renvoyée au filtre qui évalue ainsi la similarité des deux signaux par rapport à certains critères de performances et modifie la valeur de ces poids pour augmenter la similarité.

Les filtres peuvent être utilisés par exemple pour supprimer le bruit dans les signaux de paroles. Considérons la situation suivante : un pilote parlant par radio dans un cockpit. Le filtre aura pour but de supprimer le bruit de fond. Pour cela, il y a deux micros. L'un (micro 2) n'enregistre que le bruit de fond. C'est dans l'autre micro (micro 1) que le pilote parle, ce micro enregistre de plus le bruit de fond tel que perçu par le pilote. Le bruit de fond entre les deux micros est légèrement différent. Le signal désiré $d(t)$ correspond à la voix du pilote avec le bruit de fond (micro 1), le signal $x(t)$ correspond au bruit de fond (micro 2). L'erreur correspondra à la voix du pilote sans le bruit de fond qui sera renvoyée à l'utilisateur. Les poids du filtre évolueront pour que le bruit de fond du micro 2 soit filtré de manière à être similaire au bruit de fond du micro 1.

1.2 Filtre adaptatif avec délai

Description du Filtre Nous allons ici nous intéresser à un filtre adaptatif avec délai (*cf.* figure 8.2) : l'erreur est utilisée par le filtre après D instants. Le délai est utilisé pour diminuer la complexité d'un temps de cycle. Le filtre est un système de N cellules, chacune contenant dans un registre la valeur

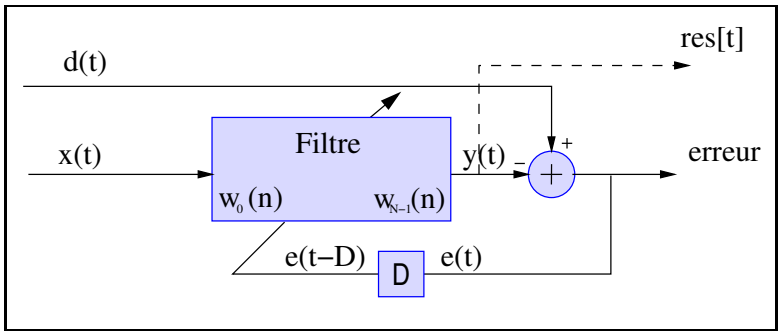


FIG. 8.2 – Filtre adaptatif avec délai

d'un des poids du filtre. Les cellules forment un réseau linéaire. Le réseau a deux entrées x et d et deux sorties e et y (nous avons repris les mêmes notations que précédemment pour les signaux). Le système est paramétré par N,M, D, ou N est la longueur du réseau (nombre de poids), M est la longueur du signal d'entrée x, et D est le délai. Ces paramètres sont définis sur le domaine :

$$\{N, M, D \mid 3 \leq N \leq \min(M - D - 1, D - 1)\}$$

Dans cet exemple, à cause de la présence du délai, certains registres ne sont pas accessibles tant que la sortie y n'est pas définie. Pour contrôler l'accès à ces registres, un signal *Wctl1P* a été introduit automatiquement lors de la synthèse. Ce signal est défini sur le domaine

$$\{p, t \mid p \leq t \leq p - N + M; 0 \leq p \leq N - 1\}$$

Tant que le système est en phase d'initialisation, les poids ne sont pas correctement définis à cause de la présence du délai. Le signal *Wctl1P* doit donc valoir faux pour empêcher l'accès à ces poids. Ensuite, le signal doit devenir et rester vrai pour permettre l'accès au registre contenant les poids. Les valeurs initiales du signal sont définies dans la première cellule puis pipelinées dans tout le réseau. Le signal *Wctl1P* est défini par l'équation suivante :

$$\begin{aligned} \textit{Wctl1P} \quad = \quad & \{t, p \mid t \leq D - 1; p = 0\} : \textit{False} \\ & \{t, p \mid D \leq t; p = 0\} : \textit{True} \\ & \{t, p \mid 1 \leq p\} : \textit{Wctl1P}.(t, p \rightarrow t - 1, p - 1) \end{aligned}$$

Les propriétés Le but ici est de vérifier que le signal *Wctl1P* vérifie bien la spécification que nous avons donnée plus haut, à savoir que pendant la phase d'initialisation le signal vaut faux, et qu'après cette phase, il vaut vrai. De manière plus formelle, voici les deux propriétés que nous voulons prouver :

$$\{\textit{Wctl1P}, \{t, p \mid p \leq t \leq p + D - 1; 0 \leq p \leq N - 1\}, \textit{ff}, \textit{SubsCons}\}$$
$$\{\textit{Wctl1P}, \{t, p \mid p + D \leq t \leq p - N + M; 0 \leq p \leq N - 1\}, \textit{tt}, \textit{SubsCons}\}$$

Les deux propriétés sont fournies à notre outil. Les deux arbres de preuve sont construits automatiquement. Pour le construire, seules deux règles sont utilisées :

- la substitution par constante,
- la séparation des domaines.

La vérification réussit immédiatement : toutes les feuilles de l'arbre sont fermées.

Cet exemple est extrêmement simple, mais c'est typiquement le genre de propriétés que notre outil de vérification sera amené à prouver.

2 Produit Matrice Vecteur

Le second exemple que nous présentons est un circuit permettant de calculer le produit d'une matrice et d'un vecteur. Nous commencerons par présenter le système et la propriété que nous souhaitons prouver puis nous verrons quelle est la stratégie de preuve que nous avons choisie et comment elle a été mise en œuvre.

2.1 Présentation

Le système est un réseau linéaire (*cf.* figure 8.3) de N cellules séparées les unes des autres par un registre.

Chaque cellule a (*cf.* figure 8.4) en entrée :

- deux données (a et b),
- trois signaux de contrôle (*init*, *empty*, *accum*)

et en sortie :

- une donnée (c),

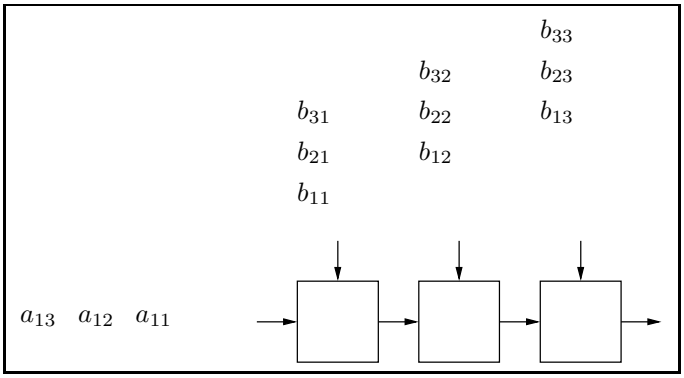


FIG. 8.3 – Structure d'un réseau calculant un produit matrice vecteur

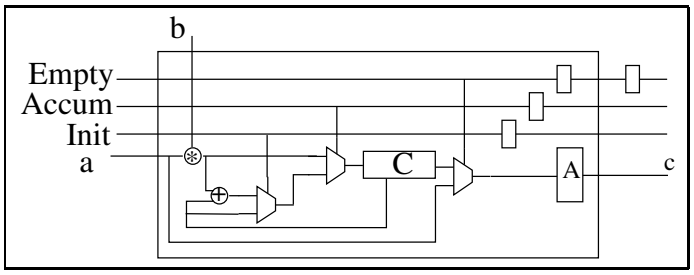


FIG. 8.4 – Structure d'une cellule

– les trois signaux de contrôle pipelinés : le décalage est produit par un registre pour les signaux *init* et *accum* et par deux registres pour le signal *empty*. Ainsi les signaux *init* et *accum* sont pipelinés deux fois plus vite que le signal *empty*.

L'entrée a représente les coefficients du vecteur, l'entrée b correspond aux coefficients de la matrice, chaque colonne de la matrice étant associée à une cellule du réseau. Une cellule se compose d'un registre C qui contient un calcul intermédiaire d'un coefficient du vecteur résultat. Le signal de contrôle *init* est utilisé pour initialiser le registre C à 0, le signal *accum* permet d'accumuler les produits des coefficients dans ce registre et le signal *empty* vide ce registre dans le registre A. Le registre A quant à lui contient soit un coefficient du vecteur d'entrée a (si le signal est faux), soit le résultat du calcul d'un coefficient. Les signaux a et b sont des séquences de N coefficients suivies de N valeurs non significatives.

Nous ne sommes pas ici intéressés par la preuve que le système effectue bien le produit matrice vecteur, mais nous voulons prouver que le résultat en sortie est toujours significatif, c'est-à-dire que le résultat dépend seulement des valeurs significatives de a et b.

Puisque nous sommes simplement intéressés par la présence ou l'absence d'un résultat significatif, nous commençons par modéliser les variables entières par des variables booléennes. De plus, nous modélisons les opérations sur les variables entières par une conjonction sur les signaux booléens correspondants : le résultat d'une opération est significatif si et seulement si tous les opérandes ont une valeur significative. La syntaxe ALPHA du système se trouve dans la figure 8.5.

La formule exacte que nous voulons prouver est la suivante :

$$\{t \mid N < t\} : c \downarrow \mathit{tt} \tag{8.1}$$

De plus, nous avons une spécification externe qui est la suivante :

– $a[1]$ est vrai, c'est-à-dire que la formule $\{t \mid t = 1\} : a \downarrow \mathit{tt}$ est valide pour tout environnement sémantiquement correct

```
system matvectmult: {N|N>1}  (a : {i|0<=i} of boolean;
                              b : {t,i|0<=t; 1<=i<=N} of boolean)
returns (c: {i|1<=i} of boolean);
var  init: {t,i|0<=t;1<=i<=N} of boolean;
  empty: {t,i|0<=t;1<=i<=N} of boolean;
  accum:{t,i|0<=t;1<=i<=N} of boolean;
  C: {t,i|1<=i<=N;0<=t} of boolean;
  A: {t,i|0<=i<=N;0<=t} of boolean;
let
  C =
    case
      {t,i|t=0}: init.(t,i -> t,i);
      {t,i|t>=1}: init.(t,i -> t,i) and   A.(t,i -> t-1,i-1) and b.(t,i -> t-1,i) or
              (not init.(t,i -> t,i) and accum.(t,i -> t,i) and C.(t,i -> t-1,i)
                 and  A.(t,i -> t-1,i-1) and b.(t,i -> t-1,i)) or
              (not init.(t,i -> t,i) and not accum.(t,i -> t,i) and C.(t,i -> t-1,i));
    esac;

  A =
    case
      {t,i| t=0; i>=1}: False;
      {t,i|t>0;i>=1}:(empty.(t,i -> t,i) and C.(t,i -> t-1,i)) or
                  ( not empty.(t,i -> t,i) and A.(t,i -> t-1,i-1));
      {t,i|t>=0;i=0}: a[t];
    esac;

  init =
    case
     {t,i|t=0}:False;
     {t,i|t=1;i=1}:True;
     {t,i|1<t<=2N;i=1}:False;
     {t,i|t>2N;i=1}: init.(t,i -> t-2N,i);
     {t,i|t>0;1<i<=N}: init.(t,i -> t-1,i-1);
    esac;

  accum =
    case
     {t,i|0<=t<=1}:False;
     {t,i|2<=t<=N;i=1}:True;
     {t,i|N<t<=2N;i=1}:False;
     {t,i|t>2N;i=1}: accum.(t,i -> t-2N,i);
     {t,i|t>0;1<i<=N}: accum.(t,i -> t-1,i-1);
    esac;

  empty =
    case
      {t,i|t<=N}: False;
      {t,i|N+1<=t<=2N; i=1}:True;
      {t,i|t>2N;i=1}:empty.(t,i -> t-2N,i);
      {t,i|t>N; 1<i<=N}:empty.(t,i -> t-2,i-1);
    esac;

  c = A.(t -> t,N);
tel;
```

FIG. 8.5 – Syntaxe ALPHA du produit matrice-vecteur

– pour t compris entre 1 et N, $b[t,t]$ est la première valeur vraie (pour tout $t' < t$, $b[t',t]$ est faux), autrement dit les deux formules suivantes sont valides pour tout environnement sémantiquement correct :

$$\{t,i \mid t = i; 1 \leq i \leq N\} : b \downarrow tt$$
$$\{t,i \mid t < i; 1 \leq i \leq N\} : b \downarrow ff$$

– b et a sont des signaux périodiques,
– ils ont une période commune valant $2N$,
– a et b sont des alternances de N valeurs vrai et N valeurs faux.

2.2 Preuves sur les signaux de contrôle

Le problème que nous étudions ici est beaucoup plus complexe que celui du filtre. Si nous donnons à notre prouveur la propriété (8.1), la preuve échoue après application de la substitution par constante. Toutes les feuilles que nous obtenons sont pendantes de statut *EchIter*. Cependant, si nous regardons les formules de ces nœuds, nous remarquons que le problème a été transformé en prouver que la variable A vaut la valeur vrai sur le domaine $\mathcal{D}_A = \{t,i \mid N < t; i = N\}$. Nous ne pouvons donc pas générer directement une preuve de $\mathcal{D}_A : A$. Nous devons prouver un résultat plus général en agrandissant le domaine $\mathcal{D}_A$. Si nous agrandissons simplement le domaine $\mathcal{D}_A$ dans la direction de la dépendance $(t, i \rightarrow t-1, i-1)$ comme proposé dans le chapitre 7, nous devrons prouver que A est vraie sur tout son domaine de définition, ce qui est évidemment faux (puisque la variable d'entrée a n'est pas vraie sur tout son domaine). Nous devons donc choisir un agrandissement plus fin. Le seul agrandissement qui convienne engendre une union infinie de polyèdres (un polyèdre pour chaque période). Comme les unions infinies ne sont pas supportées par notre modèle, nous décomposons notre preuve en deux parties :

– nous prouverons tout d'abord que tous les signaux sont périodiques,
– puis nous nous restreindrons à l'étude d'une unique période.

2.2.1 Périodicité

La périodicité du problème peut s'exprimer comme les autres propriétés et ne fait pas intervenir de nouvelle technique. Pour chacun des signaux, nous allons définir une équation de périodicité. Soit e un signal booléen défini sur un domaine $\mathcal{D}$. Soit ρ un environnement sémantiquement correct pour le système auquel appartient e. Pour alléger l'écriture, nous noterons $e[t]$ l'expression $val(\mathcal{E}(e)(\rho))[t,z]$ où (t,z) appartient au domaine $\mathcal{D}$. Le signal e sera périodique de période P si et seulement si :

$$e[t] \iff e[t-P]$$

Cette équivalence peut être réécrite sous forme normale conjonctive :

$$(e[t] \vee \neg e[t-P]) \wedge (e[t-P] \vee \neg e[t])$$

Ainsi, prouver que e est périodique pour tout environnement ρ équivaut à prouver :

$$\forall \rho, \forall (t-P,z) \in \mathcal{D},\ val(\mathcal{E}((e \vee \neg e.(t \rightarrow t-P)) \wedge (e(t \rightarrow t-P) \vee \neg e))(\rho))(t)$$

ou encore, si l'on sépare des opérandes :

$$\begin{aligned} \forall \rho, \forall (t-P,z) \in \mathcal{D}, \quad & val(\mathcal{E}((e \vee \neg e.(t \rightarrow t-P))(\rho))(t) \\ \wedge \quad & val(\mathcal{E}(e(t \rightarrow t-P) \vee \neg e)(\rho))(t) \end{aligned}$$

Nous ajoutons au système étudié les deux équations suivantes définies sur le domaine $\mathcal{D}'$ correspondant à l'intersection de $\mathcal{D}$ avec $\{t, z \mid t \geq P\}$:

$$\begin{aligned} eper1 &= e \vee \neg e.(t \rightarrow t - P) \\ eper2 &= e.(t \rightarrow t - P) \vee \neg e \end{aligned}$$

Et ainsi, prouver que e est périodique revient à prouver :

$$\begin{aligned} &\forall \rho,\ \models_\rho \mathcal{D}' : eper1 \downarrow \mathit{tt} \\ &\forall \rho,\ \models_\rho \mathcal{D}' : eper2 \downarrow \mathit{tt} \end{aligned}$$

Nous allons donc devoir construire une preuve pour les formules $\mathcal{D}' : eper1 \downarrow \mathit{tt}$ et $\mathcal{D} : eper2 \downarrow \mathit{tt}$.

Pour chacun des signaux du système nous définissons donc deux équations de périodicité (*Aper1, Aper2, Cper1, Cper2, cper1, cper2, initper1, initper2, emptyper1, emptyper2, accumper1, accumper2*) et nous associons ainsi à chaque signal deux formules. Il nous faut donc construire douze arbres de preuve. Les douze formules sont données simultanément à l'outil de preuve qui va construire les douze preuves. Quand l'outil de preuve aura terminé la construction d'un arbre de preuve, il reviendra dans les arbres de preuve précédents pour essayer de continuer la construction des nœuds pendants. De plus, nous donnons à l'outil de preuve en entrée les motifs correspondant à la périodicité des variables d'entrées.

Nous allons esquisser ici une des preuves : nous nous intéresserons à la variable *Aper1*. La preuve est construite automatiquement, nous montrerons simplement comment est construit le résultat. La première règle de construction utilisée est celle de recherche d'auto-dépendance. Elle permet de réécrire cette variable récursivement. Rappelons que simultanément à l'étape de recherche d'auto-dépendance, une recherche des motifs déjà rencontrés est effectuée. Selon les arbres de preuve déjà construits, les motifs définis par *Emptyper1*, *Cper1* et *aper1* peuvent être trouvés dans la variable *Aper1*. Ces motifs sont substitués par la constante *tt*. L'arbre se construit entièrement au fur et à mesure que les arbres des autres formules sont construits. À la fin de cette construction, les feuilles sont toutes fermées et ne contiennent que des constantes. Cependant, la construction de la preuve de la formule concernant *Aper1* dépend des constructions des autres arbres de preuve. Il faut que pour tout les arbres les feuilles soient fermées pour pouvoir conclure.

Lorsque la construction de tous les arbres de preuve s'arrête, certaines feuilles sont pendantes. Ces feuilles concernent les variables *init*, *empty* et *accum*. Pour la variable *empty*, nous avons par exemple la formule suivante à prouver :

$$\{t, i \mid 2N \leq t < 3N; i = 1\} : \neg \mathit{empty}.(t, i \rightarrow t - 2, i - 1)$$

Comme ces trois signaux sont des pipelines, il suffit de propager leur valeur dans le sens inverse des dépendances, et ces feuilles sont transformées en feuilles fermées. Cependant comme la propagation n'est pas encore implémentée, nous ne pouvons donc pas conclure pour le moment. Nous remarquons que ces feuilles auraient été facilement transformées en feuille fermée si l'agrandissement de domaine dans le cas des pseudo-pipelines avait été implémenté. En effet, les signaux *init*, *empty* et *accum* sont des pseudo-pipelines. Nous devons donc agrandir les domaines des feuilles en choisissant manuellement une heuristique. Nous obtenons ainsi des nouvelles formules à prouver. Cette dernière étape nécessite actuellement une intervention de l'utilisateur, mais dès que l'agrandissement automatique sera implémenté, cette étape sera automatisée et toute la construction des preuves sera faite automatiquement.

2.2.2 Étude d'une période

Puisque nous travaillons avec des variables périodiques, nous devons donc prouver la formule suivante :

$$\{t, i \mid N < t \leq 3N; i = N\} : A \downarrow \mathit{tt}$$

Comme précédemment, à la fin de la construction de l'arbre de preuve de cette formule, il y a des feuilles pendantes. Les expressions de ces feuilles font intervenir le signal A des cellules précédentes. Concrètement, l'échec vient du fait que nous étudions le registre A de la dernière cellule. La valeur de ce registre dépend bien sûr des valeurs des registres A dans les cellules précédentes, et ces valeurs sont nécessaires pour connaître la valeur de A dans le dernier registre. Comme la variable A apparaît dans les formules des feuilles pendantes, nous devons agrandir le domaine de A. Pour cela, nous utilisons les heuristiques définies dans le chapitre 7 de la partie II. Sur le domaine $\{t, i \mid N < t \leq 3N; i = N\}$, la variable A est définie par l'équation (mise sous forme normale conjonctive)

$$A[t,i] = (\textit{empty}[t,i] \vee A[t-1,i-1]) \wedge (C[t-1,i] \vee \neg \textit{empty}[t,i]) \wedge (C[t-1,i]) \vee A[t-1,i-1]);$$

Il faut donc prouver que chaque terme de la conjonction est vrai. Nous utilisons une des heuristiques proposées dans le chapitre 7 que nous devons choisir nous même en l'état actuel de l'implémentation.

Nous avons ici plusieurs termes contenant une occurrence de A bien qu'il n'y ait qu'une seule fonction de dépendance. Nous pourrions donc nous ramener au cas ou il n'y a qu'une seule occurrence mais la méthode serait ensuite identique.

- La première étape est de repérer les pseudos-pipelines : ici nous avons la variable *empty* qui est même un pipeline.
- la seconde étape consiste à propager les constantes de ces pseudos-pipelines. Ici, comme nous avons un pipeline nous pouvons propager aussi bien la valeur *tt* que la valeur *ff*. Nous avons représenté sur la figure 8.6.
- Cette seconde étape a modifié l'équation de A qui est maintenant sur la période étudiée :

$$A = \begin{cases} \{t,i \mid N+2i-1 \leq t \leq 2N+i-1; 1 \leq i \leq N\} : C.(t, i \rightarrow t-1, i) \\ \{t,i \mid i \leq t \leq N+2i-2; 1 \leq i \leq N\} : A.(t, i \rightarrow t-1, i-1) \\ \{t,i \mid 0 \leq t \leq 2N; i = 0\} : a \end{cases}$$

Notre agrandissement est donc donné par la direction de $(t, i \rightarrow t-1, i-1)$, sur le domaine $\{t, i \mid i \leq t \leq N+2i-2; 1 \leq i \leq N\}$. Nous devons de plus prouver la formule $\{t, i \mid t = 3N-1; i = N\} : C \downarrow \textit{tt}$.

La preuve de A engendre la formule $\{t, i \mid t = N+2i-1; 1 \leq i \leq N\} : C \downarrow \textit{tt}$. Nous devons là encore agrandir le domaine de C, l'heuristique utilisée est la même que précédemment : nous propageons cette fois-ci les valeurs des pipelines *init* et *accum*. L'agrandissement devient alors trivial. Nous avons illustré dans la figure 8.7 les agrandissements de A et de C.

La construction de l'arbre va alors se poursuivre automatiquement et ne produire que des feuilles fermées.

Les deux domaines que nous avons agrandis ne sont pas actuellement agrandis automatiquement, les heuristiques présentées dans le chapitre 7 nous permettraient ici d'automatiser complètement la construction de l'arbre. Nous sommes dans le cas où nous avions des pseudos-pipelines et ou nous connaissions des valeurs que nous avons pu propager.

3 Conclusion

Notre outil de preuve nous a permis de prouver deux spécifications.

- La première concernait un arbitre matériel, et la construction de la preuve a réussi automatiquement. Cet exemple est représentatif des propriétés que nous serons amenées à prouver. Nous remarquons donc que notre outil est adapté pour de tels types de propriétés.
- Le second exemple concernait un produit matrice-vecteur. Cet exemple nous a permis d'utiliser d'autres aspects de notre outil de preuve. En l'état actuel de l'implémentation la preuve est effectuée de manière semi automatique. Cependant dès que nous aurons implémenté les heuristiques concernant les pseudos-pipelines, la preuve pourra être effectuée automatiquement. Cet exemple permet de mettre en avant le rôle joué par les pipelines et les pseudos-pipelines dans une preuve.

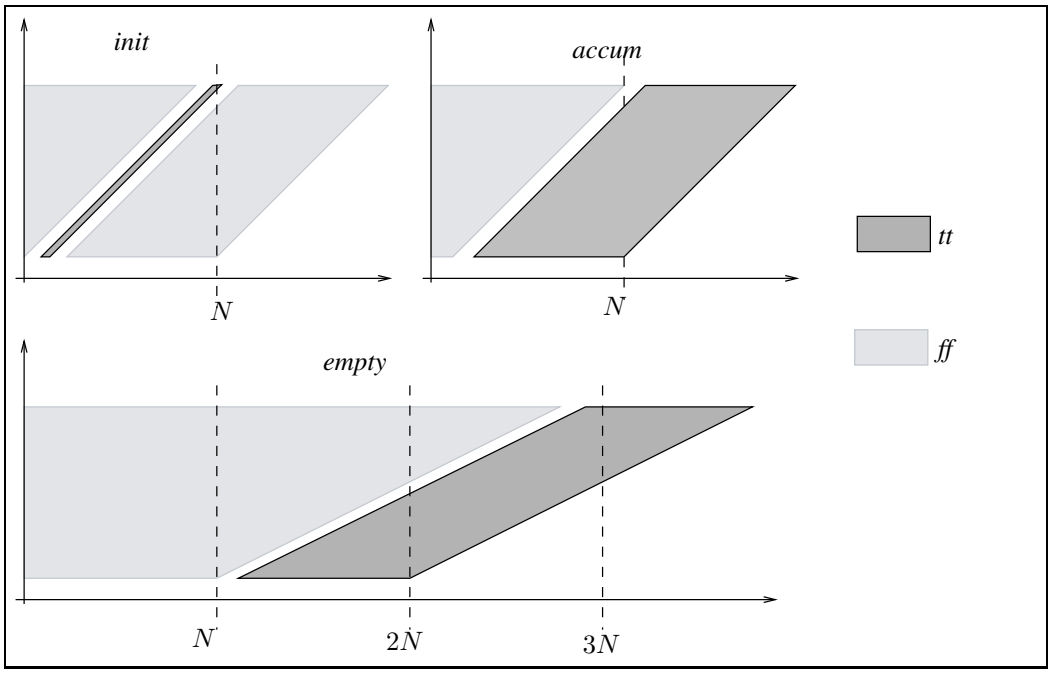


FIG. 8.6 – Répartitions des valeurs des signaux *init*, *empty* et *accum*.

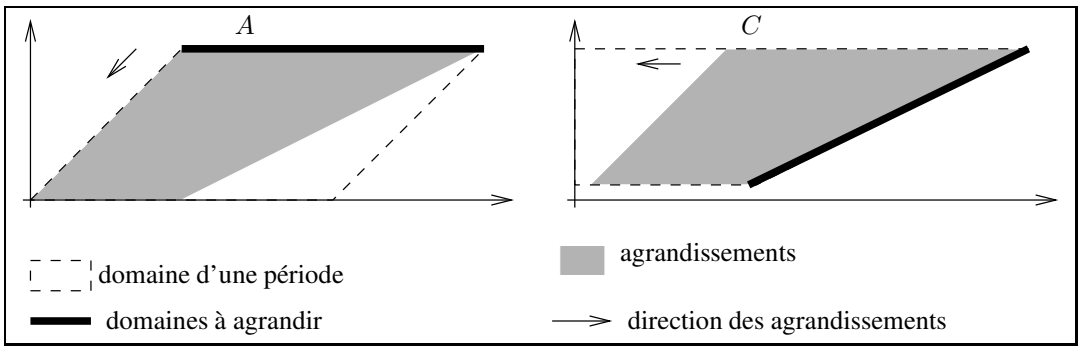


FIG. 8.7 – Agrandissements des domaines des variables A et C.

Chapitre 9

Perspectives : enrichissement de la « logique polyédrique »

Les applications traitées dans le chapitre précédent nous ont permis de voir comment utiliser notre outil de preuve. Cependant cet outil s'avère limité dès que l'on souhaite passer à des applications plus complexes, comme par exemple des applications dont une partie du comportement est spécifiée avec des hypothèses. Les propriétés que nous avons prouvées dans les chapitres précédents s'énonçaient à l'aide d'une formule polyédrique (*cf.* page 57), c'est-à-dire que le but était de prouver qu'une expression valait vrai (ou faux) sur tout un domaine. Dans le cas général, les propriétés peuvent être beaucoup plus complexes, il peut être nécessaire de quantifier les indices des domaines. Pour les exprimer, il faut donc enrichir notre notion de formule. Pour représenter des indices quantifiés, nous utiliserons des paramètres. Nous devons alors présenter de nouvelles techniques de preuve, et nous verrons que nos preuves sont paramétrées. Cependant, comme ce travail est en cours de formalisation, nous nous contenterons d'expliquer le principe de ces techniques sur un exemple. Ce chapitre peut donc être vu comme un chapitre de perspectives. L'exemple que nous avons choisi d'étudier est un arbitre matériel. Cet exemple a l'avantage de présenter différentes difficultés, tout d'abord dans la spécification des hypothèses faites sur son fonctionnement et sur les propriétés à prouver, puis dans la preuve en elle même.

Nous commencerons dans la section 1 par enrichir la notion de formule, et nous étudierons la notion de paramètre. Puis dans la section 2, nous décrirons notre exemple : un arbitre matériel. Dans la section 3, nous nous intéresserons à la spécification des propriétés et hypothèses. La section 4 permettra de présenter les différentes techniques mises en œuvre dans la preuve. La section 5 permettra de conclure.

1 Des propriétés plus riches : formules généralisées et paramètres

Dans cette section, nous allons tout d'abord généraliser la notion de formule. Jusqu'à présent, nous nous sommes limités à la preuve qu'un signal où une expression polyédrique valait vrai (ou faux) sur tout un domaine. Supposons maintenant que nous voulons prouver une implication entre deux formules. En l'état actuel de notre définition de la notion de formule, nous ne pouvons pas l'effectuer. Nous allons donc généraliser la notion de formule.

1.1 Les formules généralisées : définition et sémantique

Nous posons donc la définition suivante :

Définition 9.1 (formule généralisée) *Une* formule généralisée *se définit récursivement sur sa structure :*

– *toute formule f est une formule généralisée,*

– *Soient $\mathcal{F}_1$ et $\mathcal{F}_2$ deux formules généralisées alors*
  – *$\mathcal{F}_1 \vee \mathcal{F}_2$ est une formule généralisée,*
  – *$\mathcal{F}_1 \wedge \mathcal{F}_2$ est une formule généralisée,*
  – *$\mathcal{F}_1 \implies \mathcal{F}_2$ est une formule généralisée,*
  – *$\neg\mathcal{F}_1$ est une formule généralisée.*

Intéressons nous maintenant à la sémantique des formules généralisées.

Définition 9.2 (Validité) *Soit S un système, soit $\mathcal{F}$ une formule généralisée telle que toutes les formules la composant sont construites à partie des variables de S. Soit ρ un environnement sémantiquement correct de S. La formule généralisée $\mathcal{F}$ sera valide pour ρ (notée $\models^{\star}_{\rho} \mathcal{F}$) si et seulement si :*

– *Si $\mathcal{F} = f$ où f est une formule, alors $\models_\rho f$,*
– *Soit $\mathcal{F}_1, \mathcal{F}_2$ deux formules généralisées alors,*
  – *Si $\mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$, alors $\models^{\star}_{\rho} \mathcal{F}_1$ et $\models^{\star}_{\rho} \mathcal{F}_2$,*
  – *Si $\mathcal{F} = \mathcal{F}_1 \vee \mathcal{F}_2$, alors $\models^{\star}_{\rho} \mathcal{F}_1$ ou $\models^{\star}_{\rho} \mathcal{F}_2$,*
  – *Si $\mathcal{F} = \mathcal{F}_1 \Rightarrow \mathcal{F}_2$, alors non $\models^{\star}_{\rho} \mathcal{F}_1$ ou $\models^{\star}_{\rho} \mathcal{F}_2$,*
  – *Si $\mathcal{F} = \neg\mathcal{F}_1$, alors non $\models^{\star}_{\rho} \mathcal{F}_1$.*

Nous dirons que la formule généralisée $\mathcal{F}$ est valide (notée $\models^{\star} \mathcal{F}$) si et seulement si pour tout $\rho \propto S$, $\models^{\star}_{\rho} \mathcal{F}$.

Il est intéressant de remarquer que la négation d'une formule n'est pas une formule mais une formule généralisée. Cependant, si le domaine de la formule est réduit à un point, sa négation est alors une formule. Nous appelons de telles formules des formules locales.

Formules locales Plus formellement, nous appelons domaine local, un domaine réduit à un point pour toutes valeurs des paramètres. Ces domaines sont de la forme suivante :

$$\{i_1, \ldots, i_n \mid i_1 = f_1(P_{l_1}, \ldots, P_{l_n}), \ldots, i_n = f_n(P_{l_1}, \ldots, P_{l_n})\}$$

où $i_1, \ldots, i_n$ sont les indices du domaine, et $P_1, \ldots, P_m$ les paramètres du domaine et pour tout k, l_k appartient à $\{1, \ldots, m\}$, et $f_i(P_{l_1}, \ldots, P_{l_n})$ des combinaisons linéaires de paramètres.

Une formule dont le domaine est un domaine local est appelée formule locale.

Ces formules ont la particularité que leur négation est elle même une formule locale.

Proposition 9.3 *Soit S un système, soit e une expression polyédrique construite à partir de variables du système, soit $\mathcal{D}$ un domaine restreint à un point. La négation de la formule $\mathcal{D} : e \downarrow v$ est $\mathcal{D} : e \downarrow \bar{v}$.*

Démonstration : *Il faut montrer que pour tout environnement ρ sémantiquement correct pour S nous avons :*

$$S \models^{\star} \neg\mathcal{D} : e \downarrow v \Leftrightarrow S \models^{\star} \mathcal{D} : e \downarrow \bar{v}$$

Travaillons par équivalence

$$\begin{aligned} S \models^{\star} \neg\mathcal{D} : e \downarrow v &\Leftrightarrow \text{non } S \models^{\star} \mathcal{D} : e \downarrow v \\ &\Leftrightarrow \text{non } S \models \mathcal{D} : e \downarrow v \\ &\Leftrightarrow \text{non } \forall z \in \mathcal{D},\ val(\mathcal{E}(e)(\rho))[z] = v \\ &\Leftrightarrow \exists z \in \mathcal{D},\ val(\mathcal{E}(e)(\rho))[z] = \bar{v} \end{aligned}$$

Or, le domaine $\mathcal{D}$ est un domaine réduit à un point, il est de la forme $\{i_1, \ldots, i_n \mid i_1 = P_{l_1}, \ldots, i_n = P_{l_n}\}$. L'expression $\exists z \in \mathcal{D}$ correspond au seul point $i_1 = P_{l_1}, \ldots, i_n = P_{l_n}$ et l'expression $\forall z \in \mathcal{D}$ correspond au même point. Nous avons ainsi montré que

$$S \models^{\star} \neg\mathcal{D} : e \downarrow v \Leftrightarrow S \models^{\star} \mathcal{D} : e \downarrow \bar{v}$$

□

Nous devons maintenant compléter notre système de preuve (*cf.* page 115), pour tenir compte de ces formules généralisées. Mais avant cela nous redéfinissons la notion d'implication dans le cadre des formules généralisées.

Définition 9.4 (Implication) *Soit S un système, Γ un ensemble de formules généralisées, et $\mathcal{F}$ une formule généralisée sur ce système. On dit que Γ implique $\mathcal{F}$ ce qu'on note $\Gamma \models\!\!\!\!^{\star} \mathcal{F}$ si pour tout environnement sémantiquement correct ρ tel que toutes les formules de Γ sont valides, $\mathcal{F}$ est valide pour ρ.*

L'ensemble Γ est appelé *contexte généralisé*.

Définition 9.5 *Le contexte généralisée est l'ensemble de formules généralisées situées à gauche de $\models\!\!\!\!^{\star}$.*

Le contexte est utilisée pour spécifier les hypothèses faites sur le comportement du système étudié.

Nous ne donnerons ici qu'une version simplifiée de ce système de preuve permettant d'illustrer les règles de décomposition des formules. En particulier, nous ne faisons intervenir ni le système ni les listes de motifs, la formalisation n'étant pas achevée. Ce système simplifié est situé en annexe F section 2.

L'existence d'une preuve pour une formule généralisée $\mathcal{F}$ dans un système S sera notée sous le contexte Γ :

$$\Gamma; S \vdash\!\!\!\!^{\star} \mathcal{F}$$

S'il existe une preuve pour une formule généralisée $\mathcal{F}$ alors, cette formule sera valide :

$$\Gamma; S \vdash\!\!\!\!^{\star} \mathcal{F} \;\Rightarrow\; \Gamma \models\!\!\!\!^{\star} \mathcal{F}$$

Cette implication n'a bien sûr pas été démontrée, puisque nous n'avons pas montré que notre système de preuve était sûr.

1.2 Paramètres

Utilisation des paramètres pour spécifier L'enrichissement que nous venons de proposer n'est cependant pas suffisant. Supposons que nous voulions spécifier une propriété s'énonçant sémantiquement de la manière suivante :

$$\forall t \geq 0, val(\mathcal{E}(A)(\rho))[t] \implies \forall t' \geq t,\ val(\mathcal{E}(B)(\rho))[t']$$

où ρ un environnement sémantiquement correct d'un système S et A et B sont des signaux booléens de S.

D'après la définition de formule généralisée, nous remarquons que la propriété que nous souhaitons spécifier est composée de deux formules :

$$\mathcal{D}_1 : A \downarrow tt \tag{9.1}$$
$$\mathcal{D}_2 : B \downarrow tt \tag{9.2}$$

où $\mathcal{D}_1$ et $\mathcal{D}_2$ sont des domaines que nous allons déterminer. Notre formule généralisée est donc :

$$(\mathcal{D}_1 : A \downarrow tt) \Rightarrow (\mathcal{D}_2 : B \downarrow tt)$$

Le domaine $\mathcal{D}_1$ est très simple, il correspond simplement à $\{t \mid t \geq 0\}$. Déterminer le domaine $\mathcal{D}_2$ est plus difficile, l'indice t' est contraint par l'indice t utilisé dans la formule à gauche de l'implication. Or, la portée des indices est limité au domaine, nous ne pouvons donc pas exprimer le domaine $\mathcal{D}_2$ à

l'aide d'un polyèdre. La solution est simplement de trouver un indice dont la portée n'est pas limitée à un domaine. C'est le cas des paramètres. Nous allons donc ajouter un paramètre T au système étudié pour représenter l'indice t. Toutes les variables seront donc définies avec un paramètre supplémentaire. Pour la première formule, nous aurons comme contrainte $t = T$, et dans la deuxième formule, nous aurons la contrainte $t' \geq T$. La formule généralisée est donc la suivante :

$$(\{t \mid t = T\} : A \downarrow tt) \Rightarrow (\{t \mid t' \geq T\} : B \downarrow tt)$$

Intéressons nous maintenant à la preuve de cette propriété.

Preuves paramétrées Nous voulons donc fournir une preuve de :

$$S \models\!\!\!\models (\mathcal{D}_1 : A \downarrow tt) \Rightarrow (\mathcal{D}_2 : B \downarrow tt)$$

Nous avons ici une implication, donc en utilisant la règle de l'implication présenté dans le système de preuve présenté en annexe F, notre problème est transformée en :

$$\{\mathcal{D}_1 : A \downarrow tt\}; S \models\!\!\!\models \mathcal{D}_2 : B \downarrow tt$$

Un raisonnement simpliste pourrait nous permettre de dire que, si nous obtenons pour le signal A des feuilles fausses alors nous aurons montré notre formule généralisée (règle AX1, AX3). Ce raisonnement est incorrect. En effet, si nous n'obtenons des feuilles fausses que pour certaines valeurs du paramètre T, par exemple pour $T = 0$, alors nous n'aurons montré la formule généralisée que pour $T = 0$, il restera encore à la montrer pour toutes les valeurs de T strictement positif. Cette explication montre que les preuves que nous construisons sont paramétrées. Reprenons ce raisonnement dans un cadre plus général.

Toutes les preuves que nous construisons sont paramétrées par les paramètres du système étudié.

Les paramètres ne font pas partie du modèle polyédrique standard, ils ont été introduits dans [29] pour permettre de représenter des familles de systèmes de manière finie. À chaque valeur du vecteur de paramètres est associée un unique système non paramétré. Rien ne distingue dans la sémantique les indices des paramètres : un environnement est sémantiquement correct s'il est sémantiquement correct pour toutes les valeurs des paramètres. Nous pouvons cependant considérer un environnement comme une famille d'environnements. À chaque valeur du vecteur de paramètres est associé un environnement associé à un système non paramétré. Ainsi, nous pouvons considérer que l'expression « pour tout environnement sémantiquement correct de S » est un raccourci pour : « pour tout vecteur $(v_1, \ldots, v_n)$ de paramètres, pour tout environnement sémantiquement correct du système $S_{(v_1,\ldots,v_n)}$, où $S_{(v_1,\ldots,v_n)}$ est le système de la famille de systèmes S tel que les paramètres $P_1, \ldots, P_n$ de S valent respectivement $P_1 = v_1, \ldots, P_n = v_n$ ».

Nous rappelons la notion d'implication :

> *Soient S un système,* Γ *un ensemble de formules et* φ *une formule sur ce système, on dit que* Γ *implique* φ *noté* $\Gamma \models \varphi$ *si*
>
> $$\forall \rho \propto S,\ (\forall \psi \in \Gamma,\ \models_\rho \psi) \Rightarrow \models_\rho \varphi$$

L'implication est sous la portée de la quantification de l'environnement, elle est donc implicitement sous la portée de la quantification universelle des paramètres. Nous avons ainsi mis en évidence le fait que les preuves étaient paramétrées par les paramètres du système.

Dans cette section, nous avons donc enrichi notre environnement de spécification, et nous avons montré que les preuves étaient paramétrées. La section suivante sera consacrée à la description et à la spécification de notre exemple.

2 Arbitre : description et spécification

L'exemple que nous allons utiliser dans toute cette partie, est un arbitre matériel. Il a été décrit, spécifié et vérifié par Halbwachs *et al.* dans [44]. Nous allons ici en donner une description détaillée, puis nous le spécifierons en ALPHA.

Le problème est le suivant, P unités $U_1, U_2, \ldots, U_P$ se partagent une ressource en exclusion mutuelle. De plus, il y a une règle de priorité : si $i < j$ alors l'unité U_i est prioritaire sur l'unité U_j. Une unité peut demander la ressource, attendre qu'on lui fournisse, l'utiliser ou la rendre.

2.1 Description d'une unité

Une unité est un processus avec deux sorties booléennes *ask* et *use* et une entrée $grant$. Ces signaux sont définis de la manière suivante :

- la sortie *ask* est vraie tant que l'unité demande la ressource,
- la sortie *use* est vraie tant que l'unité utilise la ressource,
- l'entrée $grant$ est vraie quand l'arbitre fournit la ressource à l'unité.

Nous supposons de plus que chaque unité a le comportement suivant :

- elle n'utilise pas la ressource sans que l'arbitre la lui ait fournie,
- elle ne demande pas la ressource quand elle l'utilise,
- elle commence à utiliser la ressource dès qu'on la lui fournit,
- quand elle demande la ressource, elle continue à la demander jusqu'à ce qu'on la lui fournisse,
- au départ, elle n'utilise pas la ressource.

2.2 Description de l'arbitre

La solution que nous considérons ici est une solution distribuée. À chaque unité est associé un dispositif d'arbitrage. L'arbitrage se fait à l'aide d'un jeton, symbolisé par le signal tk, passant de dispositif en dispositif. Un jeton est émis à chaque fois que la ressource est libre et qu'une unité la demande. Le jeton va ensuite être transmis de dispositif d'arbitrage en dispositif d'arbitrage, selon l'ordre de priorité établi, jusqu'à ce qu'il soit consommé par l'un des dispositifs d'arbitrage associé à une unité attendant la ressource.

L'arbitre est implémenté sous forme matérielle (figure 9.1) : les valeurs des signaux booléens sont représentées par les valeurs des fils. Le fil "$requested$" représente la disjonction de tous les fils ask venant des unités, de même, le fil "$used$" représente la disjonction de tous les fils *use*. Nous appelons requête d'arbitrage le moment correspondant à un front montant de tk, c'est-à-dire le moment où tk change de valeur pour devenir vraie (figure 9.4).

2.3 Description du dispositif d'arbitrage

Le dispositif d'arbitrage (figure 9.2) reçoit un jeton représenté par le signal tk, calcule le signal $grant$ associé et selon la valeur d'un signal interne dépendant de *ask*, transmet ou non le jeton au dispositif suivant. De manière plus précise, voici le fonctionnement du dispositif. Pour chaque dispositif, il y a un signal d'entrée tk et un signal de sortie new_tk qui représentent la présence d'un jeton en entrée et en sortie du dispositif, et un signal interne q qui représente l'autorisation de fournir la ressource à l'unité. Dans chaque dispositif, la demande d'arbitrage est modélisée par un front montant du jeton. Cela correspond dans la figure 9.3 à la boîte EDGE. Le signal q évolue de la façon suivante : sa valeur change à chaque demande d'arbitrage (front montant) pour prendre la valeur de Ask, puis elle reste constante jusque la prochaine demande d'arbitrage. La ressource sera fournie à l'unité si le dispositif est autorisé à la fournir (q vrai) et si le jeton est présent (tk vrai). Sinon, et si le jeton est présent, il est transmis au dispositif suivant (new_tk vrai).

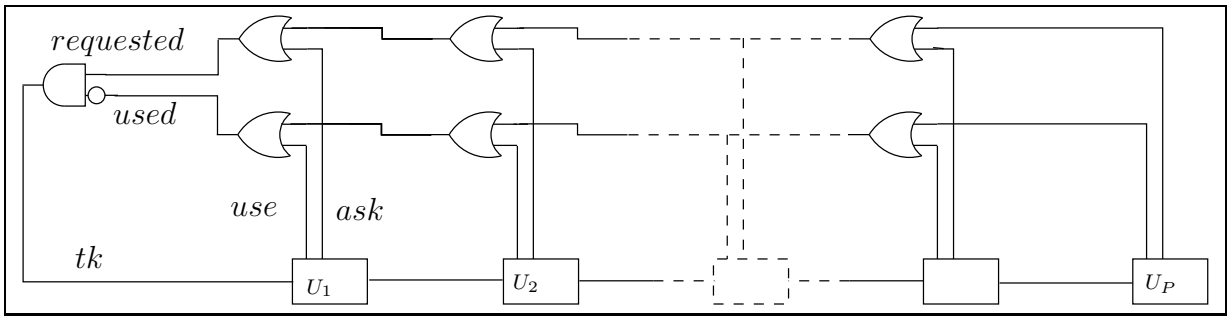


FIG. 9.1 – Arbitre

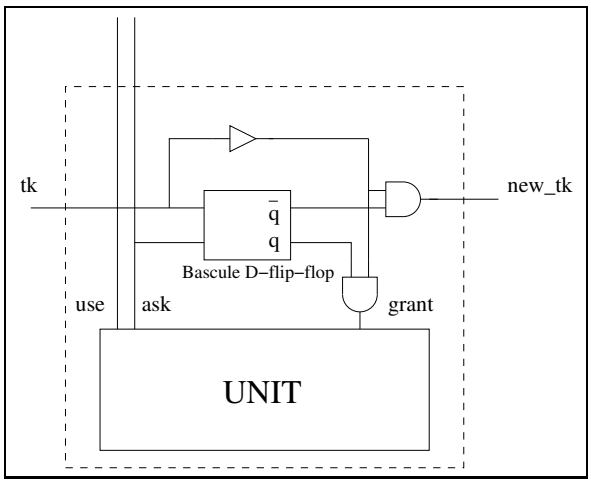


FIG. 9.2 – Une unité et son dispositif d'arbitrage

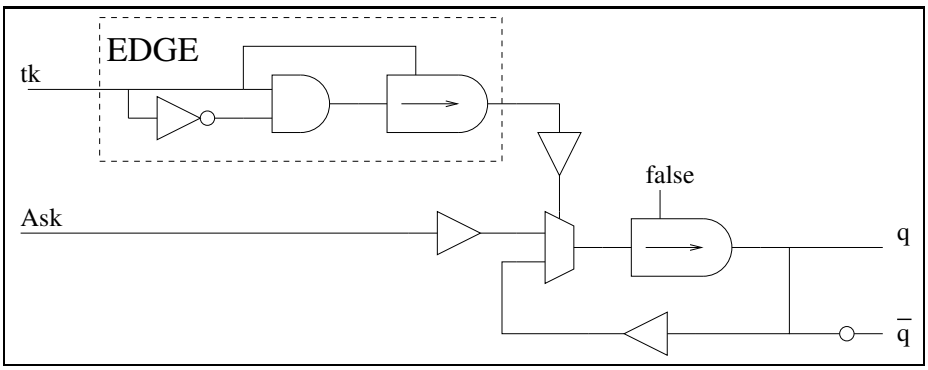


FIG. 9.3 – Bascule D-flip-flop

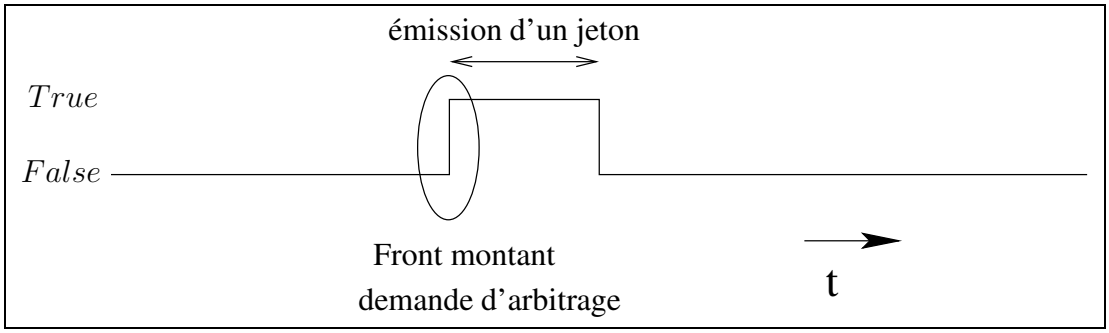


FIG. 9.4 – Valeurs d'un signal tk

2.4 Fonctionnement général de l'arbitre

Dans cet arbitre, la connaissance de l'existence d'une requête et de l'utilisation de la ressource est instantanée. Ainsi, dès qu'une unité demande la ressource, et si la ressource est libre, il y a instantanément une émission de jeton. Par contre, la transmission d'un jeton de dispositif en dispositif n'est pas instantanée. Ainsi, entre le moment où un jeton est émis et le moment où la ressource est utilisée, il s'écoule un certain laps de temps. Pendant cette période, et d'après les hypothèses faites sur le fonctionnement des unités, une unité demandant la ressource continue à la demander, et ainsi l'arbitre continue à émettre des jetons. Ces nouveaux jetons émis ne sont pas consommés par d'autres unités. De manière informelle, ces unités n'ont pas l'autorisation de les prendre car dans chaque unité, le signal q représentant l'autorisation de prendre le jeton ne change de valeur qu'au moment d'une demande d'arbitrage, c'est-à dire au moment de la première émission de jeton pendant cette période d'arbitrage. L'unité à qui l'on fournit la ressource va donc recevoir plusieurs jetons. D'après les hypothèses faites sur le fonctionnement des unités, elle est donc obligée d'utiliser la ressource pendant tout la durée où elle reçoit des jetons. Cela peut sembler au premier abord contraignant, mais ce laps de temps est négligeable par rapport au temps d'utilisation minimum de la ressource. Si ce n'était pas le cas, il faudrait choisir un autre type d'implémentation.

2.5 Les propriétés

Comme nous l'avons dit plus haut, cet arbitre doit respecter deux propriétés : l'exclusion mutuelle, et une priorité sur l'ordre d'attribution de la ressource. De manière informelle, cela signifie qu'une seule unité peut utiliser la ressource à un instant donné, et si deux unités à un instant donné demandent la ressource, celle de plus petit rang l'aura en premier. Cette dernière propriété est trop forte pour être vérifiée en ces termes dans le cas d'une solution distribuée. En effet, cette propriété nécessite une connaissance instantanée de l'ensemble des requêtes au niveau de chaque dispositif d'arbitrage. La propriété que cet arbitre va vérifier est la suivante : si la ressource est fournie à une unité à un instant donné, alors au moment de la requête d'arbitrage (au moment de l'émission d'un premier jeton), il n'y avait aucune unité de rang inférieur qui demandait la ressource. En fait, du point de vue de l'unité, cette propriété simplifiée n'est pas fondamentalement différente de la propriété idéale si le temps de propagation des informations d'arbitrage est très court.

2.6 Transcription du système dans le modèle polyédrique

La transcription de l'arbitre dans le modèle polyédrique ne pose pas de problème. Elle a été réalisée par Le Moënner au cours de son stage de DEA ([62]). Le circuit est décrit en affectant une variable polyédrique pour représenter un signal présent dans chaque unité. C'est-à-dire que la variable polyédrique *ask* va représenter l'ensemble des signaux *ask* de l'arbitre. Les signaux *ask*, *use*, $grant, q, tk$ seront représentés par des variables à deux dimensions et un paramètre : une dimension représentant le temps, l'autre le rang de l'unité, et le paramètre le nombre d'unités. Ainsi, l'instance *ask*$[t, i]$ représentera le signal *ask* de l'unité i à l'instant t. Pour la variable tk, l'interprétation est la suivante : l'instance $tk[t, i]$ représente la présence à l'instant t du jeton en sortie du dispositif associé à l'unité i, ou de manière équivalente sa présence en entrée du dispositif de l'unité $i + 1$. Deux variables sont rajoutées pour exprimer la disjonction des variables *use* et *ask*, ce sont les variables Or_ask et Or_use. La traduction du système dans le langage ALPHA est située dans la figure 9.5.

Dans la figure 9.6, nous avons représenté la répartition attendue des valeurs de la variable tk. Après un front montant, le jeton continue à être émis pendant N unités de temps (si l'unité N obtient la ressource).

La traduction des propriétés sur le fonctionnement de l'unité dans le modèle polyédrique est quant à elle plus difficile. Ce sera l'objet de la section suivante.

```
system arbitre:{P|P>1}
(ask,Use:         {t,i|t>=0;1<=i<=P} of boolean)
returns
        (grant:   {t,i|P>=i>=1;t>=0} of boolean);
var
        q:        {t,i|P>=i>=1;t>=0} of boolean;
        tk:       {t,i|P>=i>=0;t>=0} of boolean;
        Or_ask:   {t,i|P>=i>=0;t>=0} of boolean;
        Or_use:   {t,i|P>=i>=0;t>=0} of boolean;
        used:     {t|t>=0}           of boolean;
        requested:{t|t>=0}           of boolean;


let
q= case
  {t,i|t<=1}: false;
  {t,i|t>1}: (tk.(t,i -> t-1,i-1) and not tk.(t,i -> t-2,i-1)
            and ask.(t,i -> t-1,i))
       or ( not (tk.(t,i -> t-1,i-1) and not tk.(t,i -> t-2,i-1))
            and q.(t,i -> t-1,i));
  esac;

tk= case
  {t,i|t=0}: false;
  {t,i|t>0; i=0}: requested.(t,i -> t) and not used.(t,i -> t);
  {t,i|t>0;i>0}: not q.(t,i -> t,i) and tk.(t,i -> t-1,i-1);
  esac;

requested= Or_ask.(t -> t,P);

Or_ask= case
   {t,i|t=0}: false;
   {t,i|t>0; i=0}: false;
   {t,i|t>0;i>0}: Or_ask(t,i -> t,i-1) or ask.(t,i -> t,i);
  esac;

used= Or_use.(t -> t,P);

Or_use= case
   {t,i|t=0}: false;
   {t,i|t>0; i=0}: false;
   {t,i|t>0;i>0}: Or_use.(t,i -> t,i-1) or Use.(t,i -> t,i);
   esac;

grant= case
   {t,i|t<=1}: false;
   {t,i|t>1}: q.(t,i -> t,i) and tk.(t,i -> t-1,i-1);
   esac;

tel;
```

FIG. 9.5 – Traduction dans le langage ALPHA du système représentant l'arbitre.

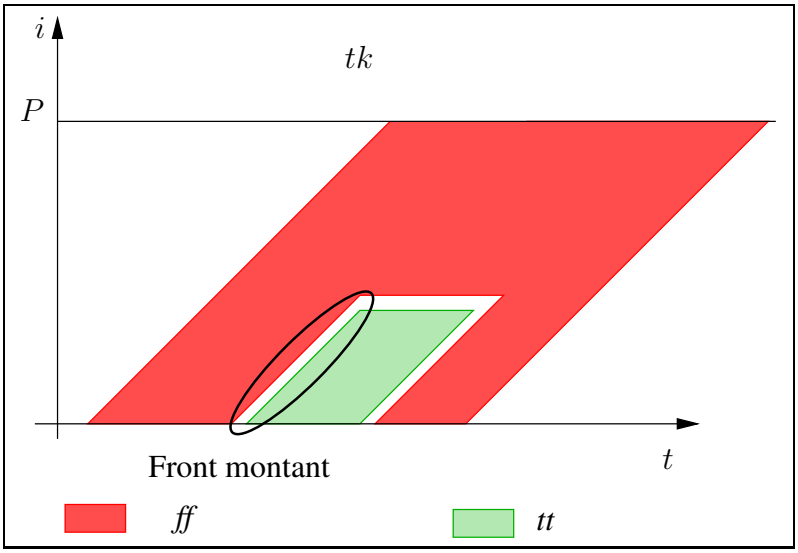


FIG. 9.6 – répartition des valeurs de tk, et front montant

3 Spécification des hypothèses et des propriétés.

Dans cette section, nous allons spécifier les hypothèses et les propriétés de l'arbitre. Nous utiliserons pour cela la notion de formule généralisée présentée en section 1.

3.1 Hypothèses sur le fonctionnement d'une unité

Dans son rapport de DEA, Le Moënner avait commencé à étudier le problème. Elle proposait de décrire ces propriétés sous formes d'équations récurrentes et d'assertions. Elle s'appuyait pour cela sur des travaux menés par Halbwachs *et al* dans [44]. Nous reprenons ici la spécification proposée par Le Moënner et nous l'adapterons à notre formalisme. L'idée est de représenter des propriétés temporelles à l'aide d'équations récurrentes. Dans ce mémoire, nous ne justifierons pas formellement la représentation proposée, ce travail est en cours de formalisation. Nous précisons simplement que la correction repose sur la sémantique du modèle polyédrique.

L'unité ne demande pas la ressource quand elle l'utilise. Cette hypothèse porte sur les variables *ask* et *use*. Elle s'exprime de la manière suivante :

$$\forall \rho \propto S_{arbitre},\ \models_\rho \{t, i \mid 0 \leq t; 1 \leq i \leq P\} : ask \wedge\ use \downarrow ff$$

Nous associons à cette hypothèse la variable $AskAndUse$ définie sur le domaine $\{t, i \mid 0 \leq t; 1 \leq i \leq P\}$ par l'équation suivante :

$$AskAndUse = ask \wedge\ use$$

L'hypothèse sera donc représentée par la formule

$$f_{AskAndUse} = \{t, i \mid 0 \leq t; 1 \leq i \leq P\} : AskAndUse \downarrow ff$$

L'unité commence à utiliser la ressource dès qu'elle lui est accordée. Cette hypothèse porte sur les signaux $grant$ et *use*. Elle s'exprime de la manière suivante :

$$\forall \rho \propto S_{arbitre},\ \models_\rho \{t, i \mid 0 \leq t; 1 \leq i \leq P\} : \neg grant \vee\ use \downarrow tt$$

Nous associons à cette hypothèse la variable $NotGrantOrUse$ définie sur le domaine $\{t, i \mid 0 \leq t; 1 \leq i \leq P\}$ par l'équation suivante :

$$NotGrantOrUse = \neg grant \vee \mathit{use}$$

L'hypothèse sera donc représentée par la formule

$$f_{NotGrantOrUse} = \{t, i \mid 0 \leq t; 1 \leq i \leq P\} : NotGrantOrUse \downarrow \mathit{tt}$$

Au départ, l'unité n'utilise pas la ressource. Cette hypothèse porte sur le signal *use*. Elle s'exprime de la manière suivante :

$$\forall \rho \propto S_{arbitre},\ \models_\rho \{t, i \mid t = 0; 1 \leq i \leq P\} :\ \mathit{use} \downarrow \mathit{ff}$$

Il n'est pas nécessaire de rajouter une variable. L'hypothèse se représente simplement par la formule :

$$f_{NotUse} = \{t, i \mid t = 0; 1 \leq i \leq P\} :\ \mathit{use} \downarrow \mathit{ff}$$

Les deux dernières hypothèses sont plus difficiles à spécifier. Ces propriétés sont les suivantes :
- l'unité n'utilise pas la ressource sans qu'on la lui ait fournie,
- quand elle la demande, elle continue à la demander jusqu'à ce qu'on lui fournisse.

Ces hypothèses font appel à une notion de « passé ». Leur traduction passe par l'utilisation de pseudo-pipelines. Elles sont spécifiées de la manière suivante :

L'unité demande la ressource jusqu'à ce qu'elle lui soit fournie. Plaçons-nous à un instant t, et supposons cette hypothèse correcte à cet instant, alors,
- soit l'unité ne demande pas la ressource et dans ce cas :
  - soit elle ne la demandait pas à l'instant précédent et l'hypothèse à l'instant précédent était vérifiée,
  - soit elle la demandait et donc la ressource vient de lui être fournie.
- sinon, elle demande la ressource.

Nous venons donc de donner une version récursive de l'hypothèse. Nous la traduire à l'aide d'une variable polyédrique $AskUntilGrant$ définie par l'équation suivante.

$$AskUntilGrant = \begin{cases} \{t, i \mid t = 0\} : & \mathit{tt} \\ \{t, i \mid t > 0\} : & \mathit{ask} \\ & \vee \quad (\mathit{ask}.(t, i \rightarrow t-1, i) \wedge grant \\ & \vee \quad (\neg \mathit{ask}.(t-1, i \rightarrow t, i) \wedge AskUntilGrant.(t, i \rightarrow t-1, i) \end{cases}$$

et la formule représentant l'hypothèse est ainsi

$$f_{AskUntilGrant} = \{t, i \mid t \geq 0; 1 \leq i \leq P\} : AskUntilGrant \downarrow \mathit{tt}$$

L'unité n'utilise pas la ressource sans qu'on lui ait fournie. Cette hypothèse s'exprime de la manière suivante.
- Soit elle n'utilise pas la ressource,
- sinon, la ressource a été fournie depuis l'instant où elle n'utilisait pas la ressource, c'est-à-dire :
  - soit la ressource vient de lui être fournie,
  - soit elle l'utilise et à l'instant précédent la ressource avait été fournie depuis l'instant où elle n'utilisait pas la ressource.

Cette traduction nécessite donc une variable polyédrique intermédiaire notée $OnceGrantSinceNotUse$. Cette variable est définie de la manière suivante :

$$OnceGrantSinceNotUse = \begin{cases} \{t,i \mid t=0\}: & grant \\ \{t,i \mid t>0\}: & grant \\ & \vee \quad (\ use \wedge OnceGrantSinceNotUse.(t, i \rightarrow t-1, i)) \end{cases}$$

Nous pouvons maintenant définir la variable associée à l'hypothèse :

$$NotUseIfNotGrant = \neg\ use \vee OnceGrantSinceNotUse$$

La formule représentant l'hypothèse sera

$$f_{NotUseIfNotGrant} = \{t,i \mid t \geq 0; 1 \leq i \leq P\} : NotUseIfNotGrant \downarrow tt$$

Définition du contexte Nous notons $\mathcal{F}_{contexte}$ la formule généralisée définie par la conjonction des formules associées aux hypothèses :

$$\mathcal{F}_{contexte} = f_{AskAndUse} \wedge f_{NotGrantOrUse} \wedge f_{NotUse} \wedge f_{NotUseIfNotGrant} \wedge f_{AskUntilGrant}$$

3.2 Spécification des propriétés

Il nous faut maintenant exprimer les propriétés que nous voulons prouver. La spécification des propriétés proposée par Le Moënner ne permettait d'exprimer les propriétés que pour une valeur fixée des paramètres. La solution que nous proposons ici est paramétrée : elle est donc valable quelle que soit la valeur des paramètres.

Exclusion mutuelle. La première propriété concerne l'exclusion mutuelle. Deux unités ne peuvent pas utiliser simultanément la ressource. Cette propriété peut s'exprimer sémantiquement de la manière suivante :

$$\forall \rho \propto S_{arbitre}, \forall P, \forall t, \forall i,\ 1 \leq i \leq P, val(\mathcal{E}(\ use)(\rho))[t,i] \Rightarrow \forall j, j \neq i, \neg val(\mathcal{E}(\ use)(\rho))[t,j] \quad (9.3)$$

Si nous voulons prouver cette propriété dans le modèle polyédrique, il faut transformer cette expression en une formule généralisée. Comme nous avons une implication, la formule généralisée sera une implication entre deux formules. Pour la première formule, nous pouvons donc prendre $\{t,i \mid 1 \leq i \leq P; t \geq 0\}$: $use \downarrow tt$, la seconde formule devrait quant à elle avoir la forme suivante $\mathcal{D}$: $use \downarrow ff$ où $\mathcal{D}$ est un domaine. Cependant nous ne pouvons pas exprimer le domaine $\mathcal{D}$. En effet, dans l'expression 9.3, la quantification de j est sous la portée de la quantification de i. La solution est d'utiliser un paramètre comme nous l'avons expliqué dans la section 1.2.

Dans l'expression 9.3, la deuxième partie de l'implication, $\forall j \neq i, \neg val(\mathcal{E}(\ use)(\rho))[t,j]$ est sous la portée des deux indices t et i (P est un paramètre). Nous allons donc rajouter au système $S_{arbitre}$ deux paramètres T_1 et I_1 qui seront liés aux indices t et i de la manière suivante $T_1 = t$ et $I_1 = i$ et que nous allons utiliser pour spécifier la propriété sur l'exclusion mutuelle :

$$\begin{aligned} \forall \rho \propto S_{arbitre}, \forall P, \forall T_1, \forall I_1, (\forall t,\ t = T_1, \forall i,\ i = I_1, val(\mathcal{E}(\ use)(\rho))[t,i])) \\ \Rightarrow (\forall j,\ j \neq I_1, \forall t,\ t = T_1,\ \neg val(\mathcal{E}(\ use)(\rho))[t,j]) \end{aligned}$$

Avec cette nouvelle expression, nous pouvons maintenant exprimer facilement nos deux formules

$$\begin{aligned} f_{mutex1} &= \{t,i \mid t = T_1, i = I_1\} : use \downarrow tt \\ f_{mutex2} &= \{t,i \mid t = T_1, i < I_1\} \cup \{t,i \mid t = T_1, i > I_1\} : use \downarrow ff \end{aligned}$$

La formule généralisée $\mathcal{F}_{mutex}$ associée à la propriété sur l'exclusion mutuelle se définit donc par :

$$\mathcal{F}_{mutex} = f_{mutex1} \Rightarrow f_{mutex2} \quad (9.4)$$

Priorité. La seconde propriété concerne la priorité : si la ressource vient d'être fournie à une unité i, alors lors de la précédente requête d'arbitrage aucune unité de rang inférieure à i ne demandait la ressource. Ceci peut s'exprimer sémantiquement de la manière suivante :

$$\begin{array}{c}\forall \rho \propto S_{arbitre}, \forall P, \forall i,\ 1 \leq i \leq P, \forall t,\ t - i > 0 \\ val(\mathcal{E}(grant)(\rho))[t, i] \wedge \neg val(\mathcal{E}(grant)(\rho))[t-1, i] \Rightarrow \\ \forall j,\ j < i,\ \neg val(\mathcal{E}(ask)(\rho))[t - i - 1, j]\end{array}$$

Nous retrouvons le même problème que pour l'exclusion mutuelle : l'indice j est sous la portée de l'indice i, et l'indice t à une portée des deux côtés de l'implication. Nous devons ajouter des paramètres pour pouvoir exprimer cette propriété sous forme de formule généralisée. Nos deux formules sont donc ici :

$$\begin{array}{rcl} f_{prior1} & = & \{t, i \mid t = T_2, i = I_2\} : grant \wedge grant.(t, i \rightarrow t - 1, i) \downarrow tt \\ f_{prior2} & = & \{t, i \mid t = T_2, i < I_2\} \cup \{t, i \mid t = T_2, i > I_2\} : ask \downarrow ff \end{array}$$

La formule généralisée $\mathcal{F}_{prior}$ associée à la propriété de priorité se définit donc par :

$$\mathcal{F}_{prior} = f_{prior1} \Rightarrow f_{prior2}$$

Nous devrons donc prouver que :

$$\mathcal{F}_{contexte}; S_{arbitre} \models^{\star} \mathcal{F}_{prior} \wedge \mathcal{F}_{mutex} \tag{9.5}$$

4 Preuve de l'arbitre

Le but de cette section est double :
- il est, bien sûr, de donner le schéma de preuve des propriétés,
- mais aussi, de présenter de nouvelles techniques de preuve que nous illustrerons à l'aide de l'arbitre.

Jusqu'ici, l'existence d'une preuve repose sur la construction d'un arbre de preuve dont toutes les feuilles sont fermées et les expressions sont des constantes. Le processus de construction de la preuve se déroule donc ainsi : nous commençons par construire un arbre de preuve et sur les feuilles pendantes, nous essayons d'appliquer des heuristiques d'agrandissement. Cependant, il y a de nombreux cas où ces heuristiques ne peuvent pas s'appliquer ou ne donnent aucun résultat concluant. L'utilisateur doit alors intervenir, soit pour donner un meilleur agrandissement, soit comme nous l'avons vu en conclusion du chapitre 6 pour fournir un invariant. Pour que la méthode de preuve que nous proposons soit intéressante, il faut que l'utilisateur intervienne le plus rarement possible. Nous allons donc ici proposer de nouvelles techniques pour permettre de reprendre les preuves sur les feuilles pendantes.

Nous voulons donc prouver les propriétés $\mathcal{F}_{prior}$ et $\mathcal{F}_{mutex}$. Ces deux propriétés portent sur des variables d'entrée. La construction de l'arbre ne donnera donc rien d'autre qu'une feuille pendante sur laquelle aucune règle ni agrandissement ne peut être appliquée. Rappelons que notre problème est de prouver le séquent (9.5). Ce séquent contient un contexte (*cf.* page 9.5). Jusqu'à présent, le système de preuve que nous avons développé ne tenait pas compte du contexte. Ainsi, nous ne tirons pas parti dans la preuve des formules $\mathcal{F}_{prior}$ et $\mathcal{F}_{mutex}$ des hypothèses faites sur le fonctionnement de l'arbitre. Nous devrons donc développer un nouveau système de règles de preuve. Nous nous contenterons simplement de compléter le système de règles de preuve sur les formules en ajoutant des règles pour le contexte. Ces règles sont similaires à celles que nous utilisons en partie droite du séquent. Nous les donnons en annexe F. Elles ne sont données qu'à titre informatif et nous ne prouverons pas ici leur sûreté (nous

rappelons que le but de cette section est simplement de donner l'intuition des techniques que nous souhaitons développer). La dérivation les formules du contexte, permet de construire ce que l'on appelle un arbre de dérivation. Nous préférons employer le terme arbre de dérivation à arbre de preuve lorsque l'on dérive les formules du contexte car le but n'est pas pour ses formules de construire une preuve mais de construire des formules qui sont des conséquences de formules du contexte. En construisant cet arbre, nous augmentons ainsi le nombre d'hypothèses. pour les formules du contexte. La construction des arbres de dérivation des formules du contexte permet de générer de nouvelles formules dans le contexte, de fournir des motifs et de simplifier le système. Les motifs ainsi obtenus peuvent ensuite être utilisés dans la construction des arbres de preuve des formules à droite du séquent.

Cependant, dans la preuve des propriétés de l'arbitre, utiliser notre outil de preuve ne permet pas d'effectuer directement la preuve de ces propriétés. Nous n'avons ici pas trouvé d'autre solution que de fournir un invariant pour construire les preuves. Nous commencerons par définir l'invariant, puis nous le prouverons. Une fois l'invariant prouvé, il pourra être ajouté aux formules du contexte pour prouver les deux propriétés.

4.1 Recherche de l'invariant

Le choix de l'invariant est effectué par l'utilisateur. Rappelons brièvement le fonctionnement de l'arbitre. Lors d'une demande d'arbitrage, c'est-à-dire lorsqu'une unité demande la ressource et que la ressource est libre, un jeton est émis. Ce jeton est transmis d'unité en unité jusqu'à arriver à une unité demandant la ressource. Tant que la ressource n'est pas transmise à l'unité la demandant, l'unité continue de la demander, et nous sommes toujours dans l'état de demande d'arbitrage ; des jetons continuent donc à être émis. Il faut N unités de temps pour qu'un jeton arrive à l'unité N, ainsi N jetons seront émis. Nous allons utiliser ce fait pour construire notre invariant. Notre invariant sera donc le suivant : si la ressource a été fournie à l'instant T à l'unité N, alors lors de la précédente requête d'arbitrage qui a débuté en $T - N$, N jetons ont été émis. Les jetons sont représentés par la variable tk.

Nous introduisons la variable $edge$ qui représente le fait que la ressource vient d'être allouée à une unité. Cette variable se définit de la manière suivante :

$$edge = grant \wedge \neg grant.(t, i \rightarrow t - 1, i)$$

L'invariant s'exprime donc sémantiquement de la manière suivante :

$$\begin{gathered}
\forall N, \forall T, val(\mathcal{E}(edge)(\rho)[T, N] \Rightarrow \\
\forall i,\ 0 \leq i < N, \forall t,\ T - N + i \leq t \leq T + i - 1, val(\mathcal{E}(tk)(\rho)[T, N] \\
\wedge \forall i,\ N \leq i, \forall t,\ T - N + i \leq t \leq T + i - 1, \neg val(\mathcal{E}(tk)(\rho)[T, N] \\
\wedge \forall i,\ 0 \leq i, \forall t,\ t = T - N + i - 1, \neg val(\mathcal{E}(tk)(\rho)[T, N] \\
\wedge \forall i,\ 0 \leq i, \forall t,\ t = T + i, \neg val(\mathcal{E}(tk)(\rho)[T, N]
\end{gathered}$$

Les indices T et N ont une portée des deux côtés de l'implication, on les représente donc à l'aide de deux paramètres T_0 et N_0 pour exprimer cette expression sous forme d'une formule généralisée. Nous définissons les deux domaines $\mathcal{D}_T$ et $\mathcal{D}_F$ représentant respectivement le domaine où tk est vrai, et le domaine où tk est faux par :

$$\begin{aligned}
\mathcal{D}_T = \quad & \{t, i \mid T_0 - N_0 + i \leq t \leq T_0 + i - 1, 0 \leq i < N_0\} \\
\mathcal{D}_F = \quad & \{t, i \mid T_0 - N_0 + i \leq t \leq T_0 + i - 1, N_0 \leq i\} \cup \{t, i \mid t = T_0 - N_0 + i - 1, 0 \leq i\} \\
\cup \quad & \{t, i \mid t = T_0 + i, 0 \leq i\}
\end{aligned}$$

Sur la figure 9.7, nous représentons en gris clair le domaine $\mathcal{D}_T$ et en gris foncé le domaine $\mathcal{D}_F$.

Les trois formules suivantes sont utilisées dans la définition de l'invariant :

$$\begin{aligned}
f_{inv_E} &= \{t, i \mid t = T_0; i = N_0\} : edge \downarrow tt \\
f_{inv_T} &= \mathcal{D}_T : tk \downarrow tt \\
f_{inv_F} &= \mathcal{D}_F : tk \downarrow ff
\end{aligned}$$

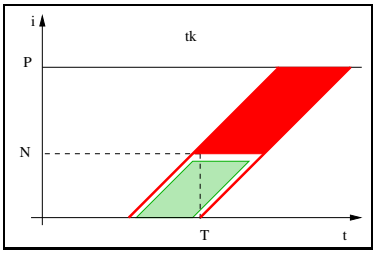


FIG. 9.7 – Répartition des valeurs de tk.

L'invariant est défini par la formule généralisée $\mathcal{F}_{inv}$:

$$\mathcal{F}_{inv} = f_{inv_E} \Rightarrow f_{inv_T} \wedge f_{inv_F}$$

Nous devrons donc prouver :

$$\mathcal{F}_{contexte}; S \mid\!\!\!\sim^{\star} \mathcal{F}_{inv} \tag{9.6}$$

Puis nous prouverons

$$\{\mathcal{F}_{contexte}, \mathcal{F}_{inv}\}; S \mid\!\!\!\sim^{\star} \mathcal{F}_{prior} \wedge \mathcal{F}_{mutex} \tag{9.7}$$

4.2 Mise en place de la preuve de l'invariant

Dans la section précédente, nous avons défini l'invariant, notre problème est donc maintenant de fournir une preuve de (9.6).

Pour expliquer la démarche de la construction de la construction de la preuve de l'invariant, nous allons effectuer deux constructions :

- La première construction s'appuie sur une utilisation directe du système de preuve. Cette construction ne permettra pas d'obtenir une preuve de l'invariant c'est-à-dire que nous allons obtenir un certain nombre de sous-buts que nous n'arriverons pas à résoudre. Cette construction nous permettra d'illustrer l'utilisation du système de preuve. De plus, elle sera réutilisée dans la seconde construction.
- La seconde preuve nous permettra de présenter différentes tactiques de preuve et nous mènera à la construction d'une preuve. La démarche de la construction de cette preuve sera expliquée dans la section suivante.

Pour simplifier les explications, nous avons préféré séparer les deux constructions, cependant la seconde construction de la preuve pourrait commencer la où s'arrête la première.

Première construction d'un arbre de preuve de l'invariant. Nous pouvons donc commencer la première construction qui consiste simplement à utiliser les règles du système de preuve. Cette construction est complètement automatique. Nous nous contentons ici d'expliquer cette construction étape par étape. Cette décomposition par étape permettra de montrer l'enchaînement du processus de preuve.

Pour prouver le séquent 9.6, nous pouvons ici commencer par appliquer :

- soit les règles de preuve sur la partie droite du séquent et construire un arbre de preuve,
- soit sur la partie gauche du séquent c'est-à-dire sur le contexte et construire des arbres de dérivation.

Si nous commençons par construire les arbres de dérivation, nous augmentons le nombre d'hypothèses et nous générons de nouveaux motifs. De plus, il est possible de simplifier le système avant de commencer la preuve. Nous avons cependant choisi de commencer par la construction des arbres de

preuve : la construction des arbres de dérivation et donc l'utilisation des formules du contexte n'intervient qu'à partir du moment où la construction de l'arbre de preuve est terminée et a échoué. Les hypothèses (formules du contexte) ne sont utilisées qu'au moment nécessaire.

Nous ne nous intéressons pour le moment qu'à la preuve de la formule $\mathcal{F}_{inv}$. Cette formule est une implication : nous utilisons la règle de preuve définie en annexe F pour le cas de l'implication. Nous ajoutons alors au contexte la formule f_{inv_E} et nous devons maintenant prouver la formule $f_{inv_T} \wedge f_{inv_F}$. Le problème est donc transformé en prouver :

$$\{\mathcal{F}_{contexte}, f_{inv_E}\}; S \stackrel{*}{\vdash} f_{inv_T} \wedge f_{inv_F} \tag{9.8}$$

Nous construisons les deux arbres de preuve et nous obtenons les deux arbres représentés dans la figure 9.8. Pour simplifier la compréhension de l'arbre, nous nous contentons de donner la structure générale de l'arbre en omettant certains nœuds intermédiaires et nous ne représentons que les formules associées aux nœuds. De plus, toujours dans l'optique d'une simplification de la compréhension de l'arbre, nous ne représentons pas la phase d'initialisation du système, nous supposons donc que T_0 est suffisamment grand. En effet, durant la phase d'initialisation du système, si $T_0 - N_0$ est négatif ou nul, nous obtenons des feuilles pendantes fausses. Nous traiterons ce cas dans le paragraphe suivant.

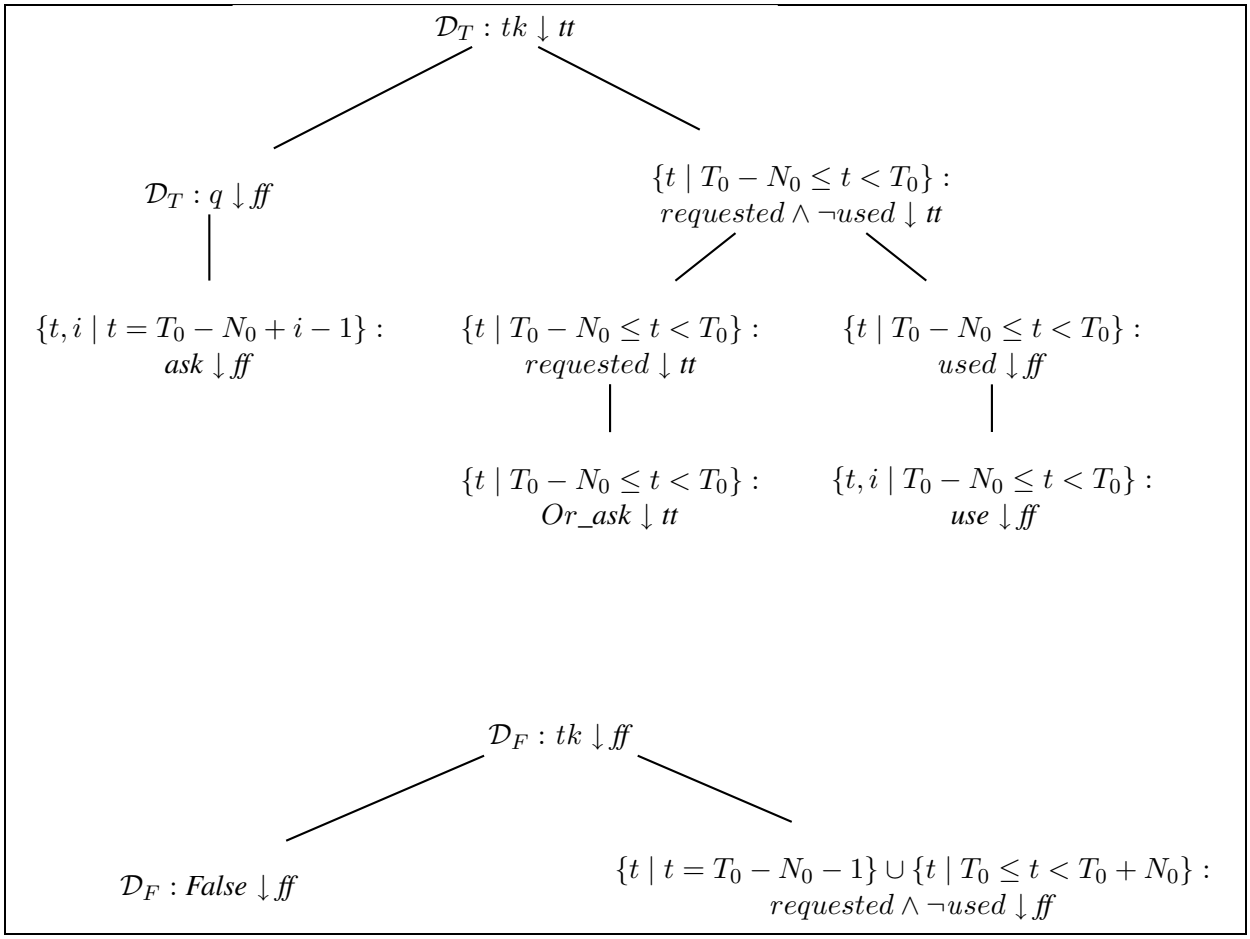


FIG. 9.8 – Structure des arbres de preuve des formules f_{inv_T} et f_{inv_F} sous le contexte $\mathcal{F}_{contexte}$ et f_{inv_E}.

La substitution par constante a permis de plus de simplifier la variable $grant$ qui est maintenant

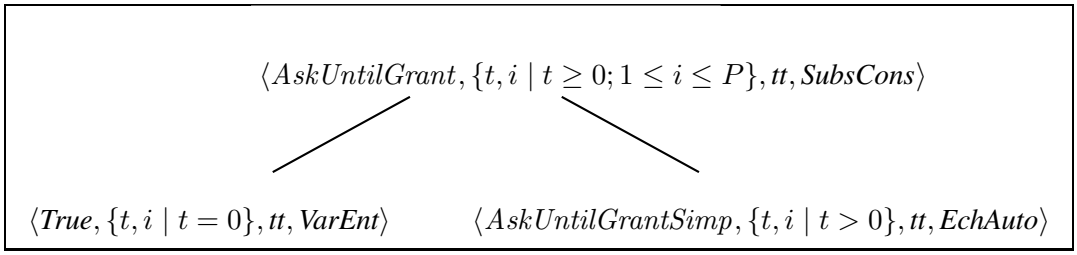
définie sur le domaine $\{t, i \mid T_0 - N_0 + i - 1 \leq t \leq T_0 + N_0\}$ par :

$$grant = \begin{cases} \{t, i \mid t = T_0; i = N_0\} : \textit{True} \\ \{t, i \mid T_0 - N_0 + i - 1 \leq t \leq T_0 + N_0\} \setminus \{t, i \mid t = T_0; i = N_0\} : \textit{False} \end{cases}$$

Nous n'obtenons qu'une seule feuille fermée dans l'arbre de la formule f_{inv_F}. Nous allons donc nous intéresser au contexte, et nous utiliserons les informations obtenues pour reprendre la construction des deux arbres de preuve.

Construction des arbres de dérivation du contexte. Pour chaque formule du contexte, nous construisons un arbre de dérivation. De plus lors de la substitution par constante, nous pouvons simplifier les expressions d'autres variables qui peuvent alors être définies par des constantes (comme c'est par exemple le cas pour la variable $grant$ lors de la construction des arbres des formules f_{inv_T} et f_{inv_F}). Dans ce cas, nous associons à ces variables une formule qui est ajoutée au contexte.

Pour les formules $f_{AskAndUse}$, $f_{NotGrantOrUse}$, f_{NotUse}, $f_{NotUseIfNotGrant}$ (*cf.* pages 167–169), les arbres de dérivation n'apportent pas de nouvelles formules dans le contexte. Ces variables sont définies par des motifs (*cf.* page 67), ainsi lors de la construction de l'arbre de dérivation, aucune règle ne peut être appliquée, seul un motif est ajouté à la liste de motifs. La dérivation de la formule $f_{AskUntilGrant}$ est plus intéressante. Nous l'avons représentée en figure 9.9.



où $AskUntilGrantSimp = \neg \textit{ask}.(t, i \rightarrow, t-1, i) \vee \textit{ask}.(t, i \rightarrow, t, i) \vee grant.(t, i \rightarrow, t-1, i)$

FIG. 9.9 – Arbre de dérivation de $f_{AskUntilGrant}$

Nous ajoutons une nouvelle formule au contexte. Elle est définie par la formule suivante :

$$\{t, i \mid t \geq 0; 1 \leq i \leq P\} : AskUntilGrantSimp \downarrow \textit{tt}$$

Cette variable est définie par :

$$AskUntilGrantSimp = \neg \textit{ask}.(t, i \rightarrow, t-1, i) \vee \textit{ask}.(t, i \rightarrow, t, i) \vee grant.(t, i \rightarrow, t-1, i) \quad (9.9)$$

Reste maintenant à construire l'arbre de dérivation de la formule f_{inv_E}.

Pour simplifier la présentation de cet arbre, nous commençons par effectuer le cas général où $T_0 - N_0 > 0$, puis nous verrons la phase d'initialisation où $T_0 - N_0 \leq 0$.

- $T_0 - N_0 > 0$. La structure de l'arbre est représentée en figure 9.10 lorsque la valeur des paramètres respecte la contrainte $T_0 - N_0 > 0$. La construction de cet arbre a permis de générer de nouvelles formules et des nouveaux motifs. En particulier, nous connaissons maintenant la valeur des signaux q, *use* et Or_ask en certains points de leur domaines. Ces signaux ont été substitués par leur valeur dans le système S. Ce système a donc été simplifié.
- $T_0 - N_0 \leq 0$. Si nous enlevons la contrainte $T_0 - N_0 > 0$, et en ne construisant que la branche correspondant à la formule $\{t, i \mid t = T_0; i = N_0\} : grant \downarrow \textit{tt}$, nous obtenons l'arbre représenté dans la figure 9.11. Su cet arbre nous n'avons plus aucune contrainte sur les paramètres T_0 et N_0. Nous obtenons donc des feuilles fermées fausses, c'est-à-dire des formules non valides. Nous rappelons que la formule f_{inv_E} fait partie du contexte (*cf.* (9.8)). Une interprétation hâtive de la

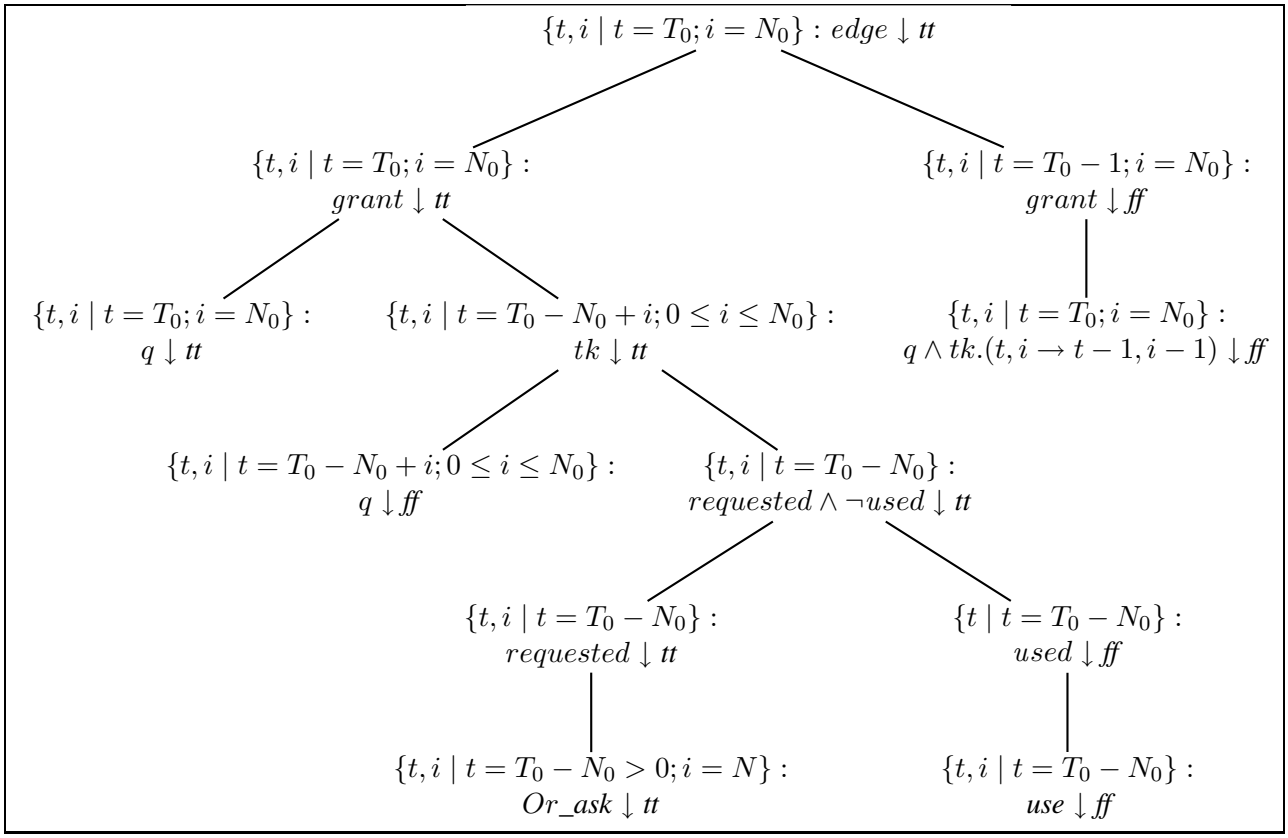
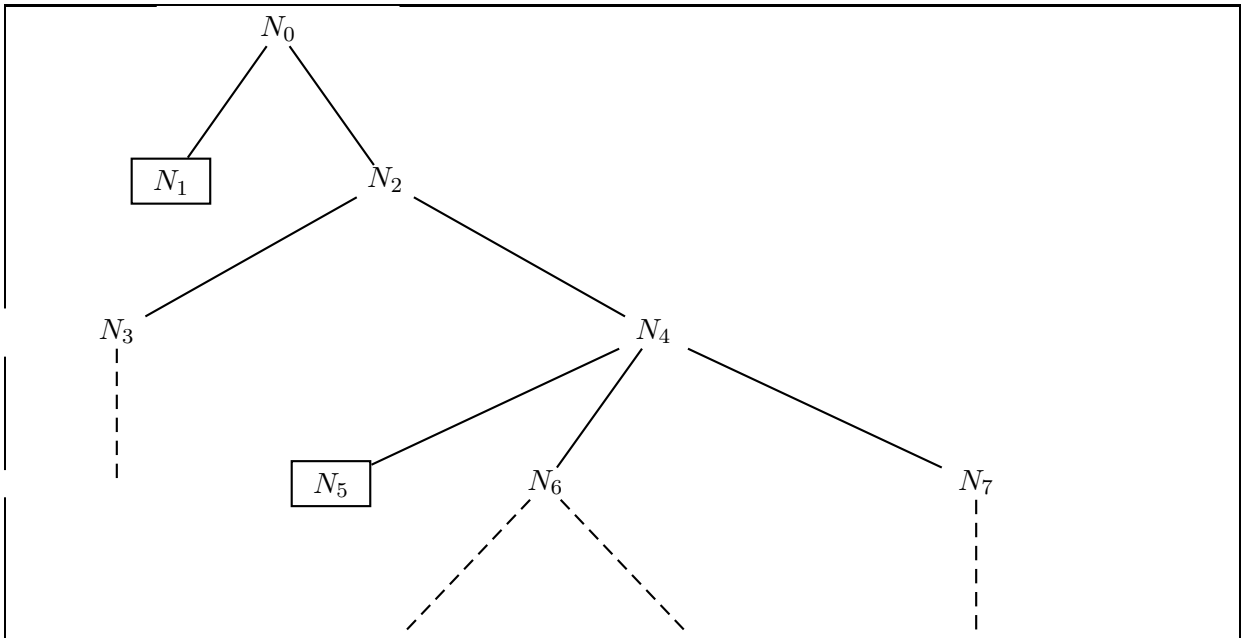


FIG. 9.10 – Arbre de dérivation de la formule f_{inv_E} pour $T_0 - N_0 > 0$. Les feuilles de l'arbre fournissent de nouvelles formules au contexte.



$N_0 = \{t, i \mid t = T_0; i = N_0\} : grant \downarrow tt$
$N_1 = \{t, i \mid t = T_0; T_0 = 1; i = N_0\} : False \downarrow tt$
$N_2 = \{t, i \mid t = T_0; i = N_0\} : q \wedge tk.(t, i \rightarrow t - 1, i - 1) \downarrow tt$
$N_3 = \{t, i \mid t = T_0; i = N_0\} : q \downarrow tt$
$N_4 = \{t, i \mid t - i = T_0 - N_0; 0 \leq i \leq N_0; t \geq 0\} : tk \downarrow tt$
$N_5 = \{t, i \mid t - i = T_0 - N_0; t = 0\} : False \downarrow tt$
$N_6 = \{t, i \mid t = T_0 - N_0; i = 0; T_0 - N_0 > 0\} : requested \wedge \neg used \downarrow tt$
$N_7 = \{t, i \mid t = T_0 - N_0; i = 0; T_0 - N_0 > 0\} : q \downarrow ff$

Les deux feuilles encadrées sont des feuilles fermées fausses, les pointillés indiquent que l'arbre n'est pas entièrement construit ici.

FIG. 9.11 – Arbre de dérivation de la formule $f = \{t, i \mid t = T_0; i = N_0\} : grant \downarrow tt$ sans contraintes supplémentaires sur les paramètres.

présence de feuilles fermées fausses peut nous mener à la conclusion que la formule f_{inv_E} est non valide et ainsi en appliquant l'axiome AX1 des règles de preuves des formules généralisées nous aurions montré $f_{inv_T} \wedge f_{inv_F}$. Cette interprétation est erronée, il suffit de se rappeler que les preuves sont paramétrées (*cf.* section 1.2). Les formules non valides correspondant aux feuilles N_1 et N_5 ne sont définies que pour certaines valeurs des paramètres : il faut que $T_0 - N_0$ soit inférieur à 0. Pour les autres valeurs des paramètres il n'y a pas de feuilles fermées fausses. Ainsi si $T_0 - N_0$ est strictement négatif, nous avons montré le séquent 9.6 et la preuve est terminée dans ce cas.

Nous pouvons maintenant reprendre la construction des arbres de preuve des formules f_{inv_T} et f_{inv_F} en se limitant au cas où $T_0 - N_0$ est positif. La structure générale des arbres est similaire aux arbres 9.8. Cependant, après la construction de l'arbre de dérivation de f_{inv_E}, quelques feuilles pendantes des arbres de preuve de f_{inv_T} et f_{inv_F} ont été transformées sur une partie de leur domaine en feuille fermée. Nous avons obtenu les feuilles fermées suivantes :

$$\begin{array}{c} \{t, i \mid t = T_0 - N_0\} : \mathit{Or\_ask} \downarrow \mathit{tt} \\ \{t, i \mid t = T_0 - N_0\} : \ \mathit{use} \downarrow \mathit{ff} \end{array}$$

Il reste cependant des feuilles pendantes dont les formules sont les suivantes :

$$\left.\begin{array}{c} \{t \mid t = T_0 - N_0 - 1\} : \mathit{requested} \wedge \neg \mathit{used} \downarrow \mathit{ff} \\ \{t, i \mid T_0 - N_0 < t \leq T_0 - 1; i = N\} : \mathit{Or\_ask} \downarrow \mathit{tt} \\ \{t, i \mid T_0 - N_0 < t \leq T_0 - 1\} : \ \mathit{use} \downarrow \mathit{ff} \\ \{t, i \mid T_0 - N_0 + i \leq t \leq T_0 - 1; 1 \leq i \leq N_0 - 1\} : \mathit{ask} \downarrow \mathit{ff} \\ \{t, i \mid t = T_0 - 1; i = N_0\} : \mathit{ask} \downarrow \mathit{tt} \end{array}\right\} \quad (9.10)$$

Les feuilles pendantes que nous n'avons pas réussi à prouver ont le sens suivant :

- La première signifie simplement que l'instant précédent le début de la requête d'arbitrage, il n'y avait pas de requête d'arbitrage.
- Les deux suivantes signifient que tant que la ressource n'a pas été attribuée à l'unité N_0, il continue à y avoir une requête d'arbitrage, c'est-à-dire que les unités voulant la ressource continuent à la demander, et aucune unité n'a la ressource.
- Les deux dernières permettent d'assurer que la ressource sera attribuée à l'unité N.

Certaines de ces feuilles pendantes correspondent à des formules sur des variables d'entrée et de sortie. Nous représentons ces feuilles dans la figure 9.12, c'est-à-dire que nous représentons la répartition des valeurs des variables d'entrée et de sortie que nous devons prouver.

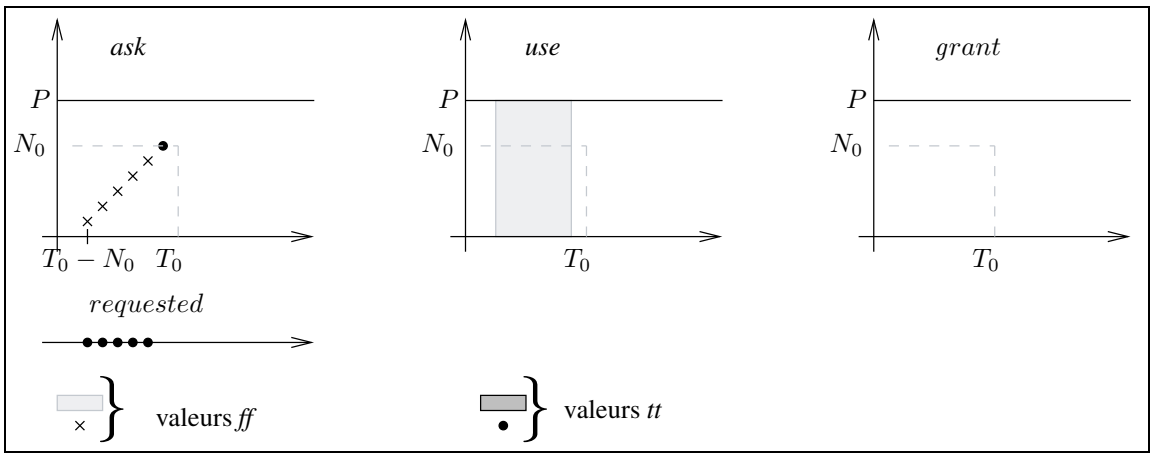


FIG. 9.12 – Répartition des valeurs à prouver

Nous avons transformé notre problème initial en la preuve de cinq formules portant sur des variables d'entrée. Le système de preuve que nous avons construit (*cf.* annexe F) ne nous permet pas de poursuivre la preuve. Nous allons donc maintenant étudier la deuxième construction de la preuve de l'invariant. Il ne faut cependant pas considérer cette première construction comme un échec, car il serait possible de commencer la deuxième construction à partir de ce qui a déjà été prouvé, cela compliquerait cependant les explications.

Schéma de construction de la seconde preuve. La seconde construction que nous allons présenter maintenant est plus complexe et fait appel à de nombreuses tactiques. Nous allons donc tout d'abord présenter le schéma de construction de la preuve et nous allons indiquer ce qui a été prouvé à chaque étape.

1. La tactique que nous utilisons est un raisonnement pas récurrence. L'idée est de prouver l'invariant en effectuant une récurrence sur les paramètres. Nous présentons simplement la mise en place du processus de récurrence.
2. La seconde étape consiste à construire l'arbre de preuve de la formule $\mathcal{F}_{inv}$, cette étape reprend exactement la première construction de la preuve, nous ne la présenterons donc pas. Nous obtenons des résultats similaires à la première construction exceptés que nous avons une formule supplémentaire dans le contexte que nous avons obtenu lors de la mise en place de l'étape de récurrence. Nous allons donc utiliser divers tactiques pour continuer à enrichir le contexte à partir de cette formule.
3. Enrichissement du contexte :
   (a) La formule introduite lors de l'étape de récurrence est une implication qui est paramétrée. Nous présentons alors une technique pour générer des formules à partir de cette implication. Nous obtenons ainsi une formule portant sur la variable $edge$.
   (b) La tactique suivante est tournée vers l'étude des pseudos-pipelines. Nous montrons comment à partir d'un pseudo-pipeline générer des formules en ajoutant simplement des paramètres.
   (c) Puis nous présentons une nouvelle utilisation de l'opérateur d'agrandissement, en agrandissant des domaines de formules portant sur des variables d'entrées.
4. Les tactiques que nous venons d'énumérer ne permettent que d'enrichir le contexte. En enrichissant le contexte, nous arrivons à prouver partiellement les formules à droite du séquent. Cependant cette preuve n'est que partielle, les formules ne sont valides que sur des sous-domaine de leur domaine. Pour effectuer la preuve complète des formules de droite, nous utilisons un raisonnement par l'absurde

Pour résumer et de manière très approximative, la preuve est donc faite en trois parties,
– tout d'abord la mise en place de la récurrence, et la construction des arbres de preuve
– puis nous appliquons un certain nombre de tactiques pour enrichir le contexte,
– enfin nous nous tournons vers la preuve des formules de droite. Comme une preuve par construction de l'arbre de preuves donne des feuilles pendantes et que nous ne pouvons plus dériver le contexte, nous utilisons un raisonnement par l'absurde.

La première tactique que nous allons présenter est un raisonnement par récurrence sur les paramètres du système.

4.3 Raisonnement par récurrence

Pour prouver des propriétés sur un système, la solution peut être une preuve par récurrence. C'est ce que nous utilisons ici pour prouver l'invariant. Bien entendu ce processus n'est pas complètement automatisable, et l'utilisateur sera amené à donner l'indice de récurrence. Cependant, des heuristiques

pourraient être développées pour proposer des indices de récurrence. La récurrence ne peut se faire que sur des paramètres (sur les indices, nous avons la substitution par constante).

Nous rappelons que l'invariant est défini par la formule généralisée $\mathcal{F}_{inv}$:

$$\{t, i \mid t = T_0; i = N_0\} : edge \downarrow tt \Rightarrow \mathcal{D}_T : tk \downarrow tt \wedge \mathcal{D}_F : tk \downarrow ff$$

où $\mathcal{D}_T$ et $\mathcal{D}_F$ sont définis par :

$$\begin{array}{lll} \mathcal{D}_T = & & \{t, i \mid T_0 - N_0 + i \leq t \leq T_0 + i - 1, 0 \leq i < N_0\} \\ \mathcal{D}_F = & & \{t, i \mid T_0 - N_0 + i \leq t \leq T_0 + i - 1, N_0 \leq i\} \cup \{t, i \mid t = T_0 - N_0 + i - 1, 0 \leq i\} \\ & \cup & \{t, i \mid t = T_0 + i, 0 \leq i\} \end{array}$$

Nous souhaitons donc prouver par récurrence la formule $\mathcal{F}_{inv}$ sur un paramètre qui reste encore à déterminer. Cette formule est paramétrée par T_0 et N_0 ainsi que par les paramètres présents initialement dans le système. Pour faire apparaître le paramétrage, nous noterons maintenant la formule $\mathcal{F}_{inv,T_0,N_0}$. Nous voulons donc prouver que pour tous T_0, N_0 $\models^{\star} \mathcal{F}_{inv,T_0,N_0}$. Nous pouvons donc effectuer la récurrence sur T_0 ou sur N_0 ou sur une expression combinant T_0 et N_0. Nous effectuons cette récurrence sur $T_0 - N_0$: l'instant $T_0 - N_0$ correspond à l'émission du premier jeton, si la ressource est fournie à l'unité N_0 à l'instant T_0.

Lors d'un raisonnement par récurrence, la première étape est de prouver le cas de base, c'est-à-dire dans le cas qui nous intéresse ici, nous restreindre au cas où $T_0 - N_0 = 0$. Dans notre technique de récurrence, le cas de base peut être directement résolu lors de l'étape de récurrence. En effet, pour effectuer l'étape nous devons supposer l'hypothèse vérifiée jusqu'au rang $T_0 - N_0 - 1$ et la prouver au rang $T_0 - N_0$, autrement dit, nous devons prouver :

$$\forall T_0, N_0,\ (\forall T_1, N_1,\ 0 \leq T_1 - N_1 < T_0 - N_0,\ \mathcal{F}_{inv,T_1,N_1}) \models^{\star} \mathcal{F}_{inv,T_0,N_0} \tag{9.11}$$

Nous avons pris ici quelques libertés de notations : tout d'abord nous avons considéré que T_0 et N_0 étaient des paramètres de la formule $\mathcal{F}_{inv,T_0,N_0}$, ce qui nous a permis d'écrire la formule $\mathcal{F}_{inv,T_1,N_1}$, puis nous avons quantifié des paramètres. La première notation est utilisée pour simplifier l'écriture, elle ne pose pas de problème particulier. L'utilisation des quantificateurs est en revanche plus délicate. Notre but est de l'intégrer à court terme à notre langage de spécification. La quantification externe de T_0, N_0 ne pose pas de problème particulier d'interprétation car nous avons vu que les preuves étaient paramétrées. Cependant la quantification de T_1 et N_1 est plus gênante tant qu'une sémantique des quantificateurs n'est pas fournie[1]. Nous allons expliquer cette notation.

Comme nous avons une quantification universelle à la gauche du signe $\models^{\star}$ dans (9.11) et que sa portée est limitée à l'expression à gauche de ce signe, nos formules ne sont pas spécifiées dans le modèle polyédrique. La solution serait de remplacer les paramètres T_1 et N_1 par des indices. Ce n'est cependant pas possible car T_1 et N_1 sont utilisés dans les formules composant la formule généralisée $\mathcal{F}_{inv,T_1,N_1}$, et nous rappelons que pour écrire la formule généralisée $\mathcal{F}_{inv,T_0,N_0}$, nous avons dû introduire les paramètres T_0 et N_0 (*cf.* section 4.1). Nous considérons donc T_1, N_1 comme des paramètres normaux que nous ajoutons au système. La formule généralisée $\mathcal{F}_{inv,T_1,N_1}$ est paramétrée par T_1 et N_1. Elle représente donc un ensemble de formules, chaque formule de cet ensemble étant associée à une valeur des paramètres T_1 et N_1. Pour chaque valeur des paramètres T_1 et N_1, la formule $\mathcal{F}_{inv,T_1,N_1}$ est sémantiquement valide. Nous pouvons donc considérer que l'expression $\forall T_1, N_1,\ 0 \leq T_1 - N_1 < T_0 - N_0,\ \mathcal{F}_{inv,T_1,N_1}$ représente une conjonction de formules associées à chaque valeur des paramètres T_1 et N_1. Nous utilisons alors une règle de preuve similaire à celle de la conjonction sur le contexte.

Ainsi, si nous prouvons (9.11), nous aurons aussi prouvé le cas où $T_0 - N_0 = 0$. En effet, si $T_0 - N_0 = 0$ l'expression $(\forall T_1, N_1,\ 0 \leq T_1 - N_1 < T_0 - N_0, (\mathcal{F}_{inv,T_1,N_1})$ n'est pas définie et nous prouvons dans ce cas :

$$\forall T_0, N_0, T_0 - N_0 = 0 \Rightarrow \models^{\star} \mathcal{F}_{inv,T_0,N_0}$$

[1] Ce travail est en cours de réalisation.

Nous devons donc prouver :

$$\mathcal{F}_{contexte}, \mathcal{F}_{inv,T_1,N_1}; S \models^{\star} \mathcal{F}_{inv,T_0,N_0}$$

Seul le contexte a été modifié par rapport à 9.6. La construction de l'arbre de preuve de $\mathcal{F}_{inv,T_0,N_0}$ est donc identique. Nous commençons par appliquer la règle de l'implication sur la formule $\mathcal{F}_{inv,T_0,N_0}$, puis nous dérivons les formules du contexte. Nous obtenons donc les formules (9.10) à prouver. Nous allons donc maintenant nous intéresser à la formule $\mathcal{F}_{inv,T_1,N_1}$. Cette formule est une implication. La section suivante sera consacrée à la manipulation des implications dans le contexte.

4.4 Cas d'une implication dans le contexte

Dans cette section nous allons nous intéresser aux implications appartenant au contexte. Nous allons ici présenter plusieurs tactiques, tout d'abord une tactique spécifique aux implications (nous n'en verrons qu'un aspect), puis une technique permettant d'« intersecter des formules » pour vérifier si elles ne sont pas contradictoires, et enfin une technique permettant de transformer des paramètres en indices. La principale difficulté que nous avons ici vient des quantificateurs T_1 et N_1 dont la portée est limitée au contexte.

La formule $\mathcal{F}_{inv,T_1,N_1}$ est une implication qui appartient au contexte. Nous pouvons donc supposer cette formule valide, sinon les formules à droite du séquent seraient automatiquement prouvées. Ainsi pour tout point T_1, N_1, soit les formules $f_{inv_{T,T_1,N_1}}$ et $f_{inv_{F,T_1,N_1}}$ sont valides soit $f_{inv_{E,T_1,N_1}}$ ne l'est pas. Notre but est donc de déterminer pour quelles valeurs de T_1 et N_1, $f_{inv_{E,T_1,N_1}}$ est invalide par rapport au reste du contexte. Pour cela, nous allons déterminer pour quelles valeurs de T_1 et N_1 les formules $f_{inv_{T,T_1,N_1}}$ et $f_{inv_{F,T_1,N_1}}$ ne sont pas valides.

Des arbres de preuve ont déjà été construits pour les formules $f_{inv_{T,T_1,N_1}}$ et de $f_{inv_{F,T_1,N_1}}$ dans la section 4.2. Puisque la construction d'un arbre de dérivation est identique à celle d'un arbre de preuve, nous reprenons ces arbres, et nous ajoutons les formules construites à partir des variables simplifiées lors de la substitution par constante (page 174, paragraphe "Construction des arbres de dérivation du contexte"). Nous ajoutons donc des formules pour la variable $grant$. Puisque nous n'avons pas reconstruit ces arbres, ces arbres ne tiennent pas compte des formules du contexte. Nous pouvons donc soit reprendre leur construction pour faire intervenir les formules du contexte, soit utiliser une autre technique que nous allons étudier ici. L'idée est de vérifier que les formules obtenues dans les arbres de dérivation sont compatibles entre elles, c'est-à-dire qu'elles n'impliquent pas qu'un même signal vaille deux valeurs distinctes en un même point.

L'arbre de dérivation de $f_{inv_{E,T_0,N_0}}$, nous donne les formules suivantes :

$$\left.\begin{array}{ll}
- & \{t,i \mid t = T_0 - N_0 + i; 1 \leq i < N_0\} \cup \{t,i \mid t = T_0 - N_0 + i; B - 0 < i \leq P\} \\
& \cup \{t,i \mid t = T_0 - 1; i = N_0\} : grant \downarrow \mathit{ff} \\
- & \{t,i \mid t = T_0; i = N_0\} : grant \downarrow \mathit{tt} \\
- & \{t,i \mid t = T_0 - N_0\} : requested \downarrow \mathit{tt} \\
- & \{t,i \mid t = T_0 - N_0\} : \mathit{use} \downarrow \mathit{ff} \\
- & \{t,i \mid t = T_0 - N_0 + i; 0 \leq i < N_0\} : tk \downarrow \mathit{tt} \\
- & \{t,i \mid t = T_0 - N_0 + i; N_0 \leq i \leq P\} : tk \downarrow \mathit{ff}
\end{array}\right\} \quad (9.12)$$

Les arbres de dérivation de $f_{inv_{T,T_1,N_1}}$ et de $f_{inv_{F,T_1,N_1}}$ donnent les formules suivantes :

- $\{t,i \mid T_1 - N_1 + i \leq t \leq T_1 + i - 1, 0 \leq i < N_1\} : tk \downarrow tt$
- $\{t,i \mid T_1 - N_1 + i \leq t \leq T_1 + i - 1, N_1 \leq i\} \cup \{t,i \mid t + 1 = T_1 - N_1 + i - 1, 0 \leq i\} : tk \downarrow ff$
- $\{t,i \mid T_1 + 1 \leq t \leq T_1 + N_1; i = N_1\} : grant \downarrow tt$
- $\{t,i \mid T_1 - N_1 + i \leq t \leq T_1 + i - 1, 0 \leq i < N_1\} : grant \downarrow ff$
- $\{t,i \mid T_1 + 1 \leq t \leq T_1 + N_1; i = N_1\} :$ *use* $\downarrow tt$
- $\{t,i \mid T_1 + 1 \leq t \leq T_1 + N_1\} : used \downarrow tt$
- $\{t \mid T_1 - N_1 + 1 \leq t \leq T_1\} : used \downarrow ff$
- $\{t \mid T_1 - N_1 + 1 \leq t \leq T_1; 0 \leq i\} :$ *use* $\downarrow ff$
- $\{t \mid T_1 - N_1 + 1 \leq t \leq T_1\} : requested \downarrow tt$
- $\{t \mid t = T_1 - N_1\} : requested \downarrow ff$

Nous allons vérifier si ces formules sont compatibles. En effet, un signal ne peut valoir qu'une valeur à la fois. Considérons le signal *use*. Le domaine $\{t,i \mid t = T_0 - N_0\}$ où *use* vaut faux, et le domaine $\{t,i \mid T_1 + 1 \leq t \leq T_1 + N_1; i = N_1 + 1\}$ ne doivent pas s'intersecter sinon nous obtenons une contradiction. Nous calculons donc l'intersection des deux domaines et nous obtenons le domaine suivant :

$$\{t,i \mid T_1 + 1 \leq t \leq T_1 + N_1; i = N_1 + 1; t = T_0 - N_0\}$$

Ainsi si nous avons $T_1 + 1 \leq T_0 - N_0 \leq T_1 + N_1$ le domaine sera non vide et nous avons une contradiction.

Nous obtenons d'autres contraintes en intersectant les formules concernant les signaux tk et $grant$. En effectuant l'union de toutes les contraintes, nous obtenons les contraintes suivantes :

$$T_1 - N_1 < T_0 - N_0 \leq T_1 + N_1$$

Ce domaine est représenté sur la figure 9.13.

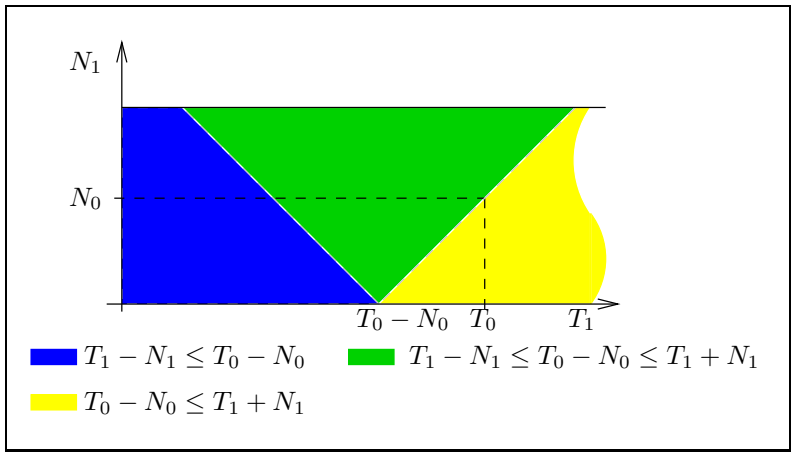


FIG. 9.13 – Contraintes sur les indices N_1, N_0, T_1 et T_0.

Nous avons donc montré que si $T_0 - N_0 \leq T_1 + N_1$ les formules $f_{inv_{T,T_1,N_1}}$ et $f_{inv_{F,T_1,N_1}}$ sont non valides, on en conclut donc que $f_{inv_{E,T_1,N_1}}$ est non valide. Nous sommes ici dans un cas intéressant car le domaine $\{t,i \mid t = T_1, i = N_1\}$ de cette formule est réduit à un point. Nous avons ainsi une formule locale et il suffit d'appliquer la proposition 9.3 correspondant à la négation d'une formule locale. Ainsi, si $edge$ en ce point ne vaut pas vrai, c'est que $edge$ vaut faux. Nous avons donc montré :

$$\{t,i \mid t = T_1, i = N_1\} : edge \downarrow ff$$

Pour résumer, nous sommes partis d'une implication de la forme $A \implies B$ où A et B sont des formules généralisées. En construisant l'arbre de preuve de B, nous avons obtenu des contradictions, et nous avons réussi à en déduire que A n'était pas valide. Un autre aspect de cette méthode est de montrer la validité de A, pour en déduire B. Cette technique peut être utilisée systématiquement sur toutes les implications du contexte. Une fois les formules obtenues, l'implication peut être supprimée.

Il est intéressant de noter qu'ici nous travaillons sur une implication paramétrée par T_1 et N_1, nous n'avons obtenu des contradictions que pour certaines valeurs de T_1 et de N_1, nous ne la supprimerons donc que pour ces valeurs de T_1 et de N_1.

Nous avons donc montré que pour tout T_0, N_0 tels que $T_1 - N_1 < T_0 - N_0 \leq T_1 + N_1$, $\{t, i \mid t = T_1; i = N_1\} : edge \downarrow \mathit{ff}$ était valide si $\{t, i \mid t = T_0; i = N_0\} : edge \downarrow \mathit{tt}$ était valide. Ainsi pour $T_1 - N_1 < T_0 - N_0 \leq T_1 + N_1$, la formule généralisée $\mathcal{F}_{inv,T_1,N_1}$ est remplacée par la formule polyédrique simple $\{t, i \mid t = T_1; i = N_1\} : edge \downarrow \mathit{ff}$. Cette dernière formule représente une conjonction de formules associées à chaque valeur des paramètres T_1 et N_1. Nous rappelons que nous avons du utiliser des paramètres pour représenter T_1 et N_1 car ils étaient utilisées dans la formule généralisée $\mathcal{F}_{inv,T_1,N_1}$, or la formule $\{t, i \mid t = T_1; i = N_1\} : edge \downarrow \mathit{ff}$ est une formule polyédrique simple, l'utilisation de paramètres pour représenter T_1 et N_1 n'est plus justifiée. Nous allons donc transformer, pour cette formule, les paramètres T_1 et N_1 en indices.

Transformation de paramètres en indices Reprenons notre formule en faisant apparaître les paramètres T_1 et N_1 dedans :

$$\{t, i, T_1, N_1 \mid t = T_1; i = N_1; T_1 - N_1 < T_0 - N_0 \leq T_1 + N_1\} : edge.(t, i, T_1, N_1 \rightarrow t, i, T_1, N_1) \downarrow \mathit{ff} \tag{9.13}$$

Si nous considérons maintenant les paramètres T_1 et N_1 comme des indices, nous ne devons pas pour autant modifier les dimensions du domaine de $edge$. C'est-à-dire que le nombre d'indices du domaine de $edge$ reste égale à 2. Si nous reprenons directement la formule (9.13), les dimensions du domaine de $edge$ changent et valent 4 ce qui n'est pas possible. Ici, l'indice t est égale à l'indice T_1, et de même i est égale à N_1, on peut donc supprimer T_1 et N_1.

Notre formule se réécrit de la manière suivante $\{t, i \mid t - i < T_0 - N_0 \leq t + i\} : edge \downarrow \mathit{ff}$.

Nous supprimons ainsi les paramètres T_1 et N_1 et nous ajoutons au contexte

$$\{t, i \mid t - i < T_0 - N_0 \leq t + i\} : edge \downarrow \mathit{ff} \wedge \{t, i \mid t = T_0; i = N_0\} : edge \downarrow \mathit{tt}$$

Cette technique de suppression ne peut être effectuée que sur des formules polyédriques locales qui ne sont pas des formules étendues. Lorsque nous disons que nous avons supprimé les paramètres T_1 et N_1 ce n'est pas tout à fait exact, nous les avons supprimés quand $T_1 - N_1 < T_0 - N_0 \leq T_1 + N_1$, c'est-à-dire que nous avons simplement restreint leur domaine de définition à $T_1 - N_1 \geq T_0 - N_0 \vee T_0 - N_0 > T_1 + N_1$, et nous avons ajouté une formule qui représente le cas où $T_1 - N_1 < T_0 - N_0 \leq T_1 + N_1$. Dans le reste de la preuve, nous n'utiliserons plus la formule généralisée $\mathcal{F}_{inv,T_1,N_1}$.

Nous avons donc augmenté le nombre de formules dans le contexte.

Les différentes techniques présentées ici s'implémentent facilement. L'utilisation de la technique de suppressions des paramètres peut facilement être automatisée car son utilisation s'appuie sur des critères syntaxiques (il faut une formule locale non généralisée). De même, l'utilisation des techniques d'intersection de formules peut facilement être automatisée, dès que nous obtenons deux formules postant sur le même signal et dont la valeur de validité est distincte, il suffit d'appliquer cette technique.

Quant à l'implication, nous avons ici présenté qu'un seul aspect de cette technique, c'est-à-dire que nous avons réussi à en déduire la non validité de la formule à gauche de l'implication. De manière similaire si nous avions tout d'abord montré la validité de la formule de gauche, alors, nous aurions pu en déduire la validité des formules à droite de l'implication. Cette tactique peut donc être utilisée systématiquement, et son utilisation peut donc être automatisée.

Nous allons maintenant nous intéresser aux pseudo-pipelines.

4.5 Utilisation des pseudos-pipelines

La dérivation de la formule f_{inv_E}, nous a permis de générer entre autres la formule $\{t, i \mid t = T_0 - N_0; i = N_0\} : Or\_ask \downarrow tt$ qui a donc été ajoutée au contexte. La variable Or_ask est un pseudo-pipeline. Nous sommes dans le cas où la valeur absorbante est propagée. D'après le corollaire 7.12 page 139, comme la formule $\{t, i \mid t = T_0 - N_0; i = N_0\} : Or\_ask \downarrow tt$ est valide et que nous avons une formule locale, nous savons qu'il existe

- soit j inférieur à N_0 tel que *ask* vaut vrai,
- soit nous avons $Or\_ask[t, 0]$ qui vaut vrai.

La seconde hypothèse est exclue par définition de Or_ask, ainsi nous savons qu'il existe j tel que *ask* vaut vrai. Nous représentons cet indice par un paramètre et nous expliquons intuitivement notre démarche dans le paragraphe suivant.

Générer des paramètres : *Jusqu'à présent, nous avons ajouté des paramètres pour spécifier des propriétés. Nous avons ensuite vu que les preuves étaient paramétrées et que les formules fausses pouvaient être définies seulement pour certaines valeurs des paramètres. Nous allons cette fois ajouter des paramètres pour effectuer des agrandissements de domaine et représenter les bornes de ces agrandissement. Afin d'automatiser cet ajout de paramètre, nous nous limiterons ici à des domaines locaux. En effet, si nous n'avons pas de domaines locaux, alors nous pouvons être amenés à ajouter une infinité de paramètres : les bornes d'agrandissement peuvent être différentes d'un point à l'autre du domaine et donc nécessitées des paramètres différents (cf. page 139). Supposons donc que nous avons la formule locale suivante :*

$$\{z \mid z = z_0\} : X \downarrow v$$

où X est définie par

$$X = \begin{cases} \mathcal{D}_1 : X.d \barwedge e \\ \mathcal{D}_2 : f \end{cases}$$

et v est la valeur absorbante pour l'opération booléenne $\barwedge$. D'après le corollaire 7.12 qui décrit la propagation de la valeur absorbante dans le sens des dépendances, nous avons :

$$\begin{aligned} val(\rho(X))[z_0] = v \Rightarrow \quad & (\exists z \in \mathcal{D}_{X,d,z_0},\ val(\mathcal{E}(e)(\rho))[z] = v \\ & \quad \wedge\ val(\rho(X))[z] = \bar{v}) \vee \\ & \exists z \in \mathcal{D}_{X,d,z_0} \cap \mathcal{D}_2, val(\mathcal{E}(f)(\rho))[z] = v \end{aligned}$$

où $\mathcal{D}_{X,d,z_0}$ est définie par :

$$\mathcal{D}_{X,d,z_0} = \{z \mid z \in \mathcal{D}_1 \wedge \exists j \in \mathbb{N},\ z = d^j(z_0)\}$$

Nous allons ajouter des paramètres pour représenter le vecteur z quantifié existentiellement. Soit z_1 le vecteur de paramètres représentant z. Ces paramètres sont définis sur le domaine $\mathcal{D}_{X,d,z_0}$.

$$\begin{aligned} val(\rho(X))[z_0] = v \Rightarrow \quad & (\forall z \in \{z \mid z = z_1\},\ val(\mathcal{E}(e)(\rho))[z] = v \\ & \quad \wedge \forall z \in \mathcal{D}_{X,d^{-1},z_1} \cap \mathcal{D}_{X,d,z_0},\ val(\rho(X))[z] = v \\ & \quad \wedge \forall z \in \{z \mid z = z_1\},\ val(\rho(X))[z] = \bar{v}) \vee \\ & (\forall z \in \{z \mid z = z_1\} \cap \mathcal{D}_2,\ val(\mathcal{E}(f)(\rho))[z] = v \\ & \quad \wedge \forall z \in \mathcal{D}_{X,d^{-1},z_1} \cap \mathcal{D}_{X,d,z_0},\ val(\rho(X))[z] = v) \end{aligned}$$

Nous allons donc maintenant chercher pour quelles valeurs de z_1 la preuve est correcte et pour quelle valeur de z_1 la preuve est fausse. C'est-à-dire que dans le cas où nous obtenons

une feuille fausse, nous aurons effectué une preuve par l'absurde, et dans les autres cas, nous aurons prouvé la formule initiale.
Nous devons donc prouver l'une des deux formules généralisées suivantes :

$$\{z \mid z = z_1\} : e \downarrow v \wedge \mathcal{D}_{X,d^{-1},z_1} \cap \mathcal{D}_{X,d,z_0} : X \downarrow v$$
$$\{z \mid z = z_1\} \cap \mathcal{D}_2 : f \downarrow v \wedge \mathcal{D}_{X,d^{-1},z_1} \cap \mathcal{D}_{X,d,z_0} : X \downarrow v$$

En pratique, la deuxième formule généralisée sera très souvent fausse, f étant généralement la constante correspondant à $\bar{v}$. La formule f correspond à l'initialisation du pseudo-pipeline, si f est définie par la constante v, le problème est trivialement vérifié, il suffit d'utiliser la proposition 7.7 page 135 dans le cas de la propagation de la valeur absorbante dans le sens inverse des dépendances. Ainsi, l'expression e n'a aucune utilité dans la définition de X. En revanche si f est la constante $\bar{v}$, la valeur de la variable X ne dépend que de l'expression e. Nous sommes dans ce cas pour la variable Or_ask qui correspond à la disjonction des instances de ask, la valeur de Or_ask ne doit dépendre que de la valeur de ask, elle est donc initialisée par False.
L'automatisation de cette technique ne pose pas de problème particulier. Il suffit simplement de savoir détecter les pseudos-pipelines et nous avons vu que les pseudos-pipelines étaient définis sur des critères syntaxiques. Nous l'appliquons dès que nous avons une formule locale sur un pseudo-pipeline et dont la valeur de validité est absorbante par rapport au pseudo-pipeline.

Nous avons donc montré comment ajouter automatiquement des paramètres dans le cas des pseudos-pipelines. La variable Or_ask étant un pseudo-pipeline, nous ajoutons un paramètre N_2 et la formule

$$\{t, i \mid t = T_0 - N_0; i = N_2\} : ask \downarrow tt$$

Une répartition des valeurs des variables d'entrée et de la variable de sortie que nous avons réussi à déduire du contexte est représentée dans la figure 9.14. Cependant, nous n'avons toujours pas avancé

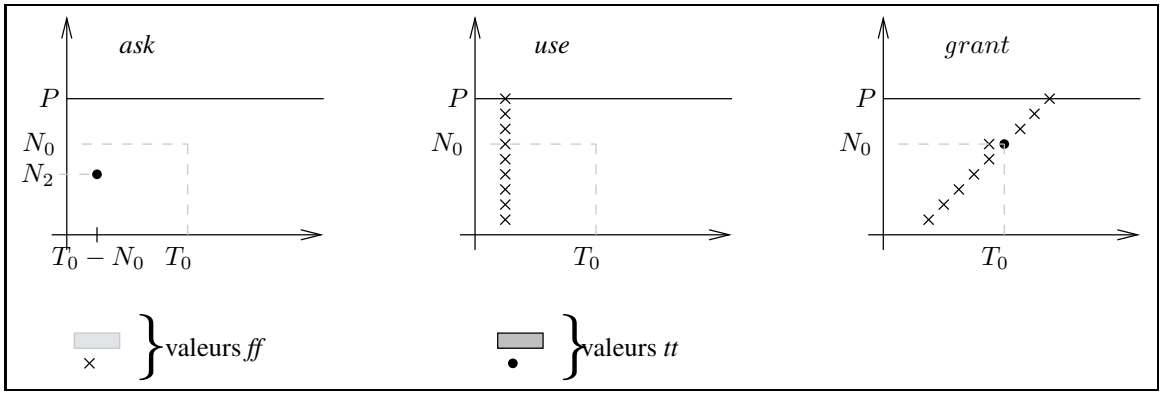


FIG. 9.14 – Répartition possible des valeurs des variables d'entrées et de sorties déduites du contexte.

dans la preuve de nos propriétés, nous avons simplement enrichi notre contexte en générant la formule $\{t, i \mid t = T_0 - N_0; i = N_2\} : ask \downarrow tt$. Cet enrichissement n'est pas suffisant pour prouver les formules (9.10). Dans la prochaine section, nous continuons à enrichir notre contexte, en utilisant cette fois-ci des techniques d'agrandissement de domaine.

4.6 Agrandissements de domaines d'expressions

Dans cette section, nous allons étudier de nouvelles applications de l'opérateur d'agrandissement de domaine pour étendre les domaines des formules du contexte et ainsi augmenter l'information contenue dans le contexte.

Agrandissement du domaine de $grant$: Les substitutions par constante lors de la construction des arbres ont simplifié d'autres variables comme par exemple la variable $NotGrantOrUse$:

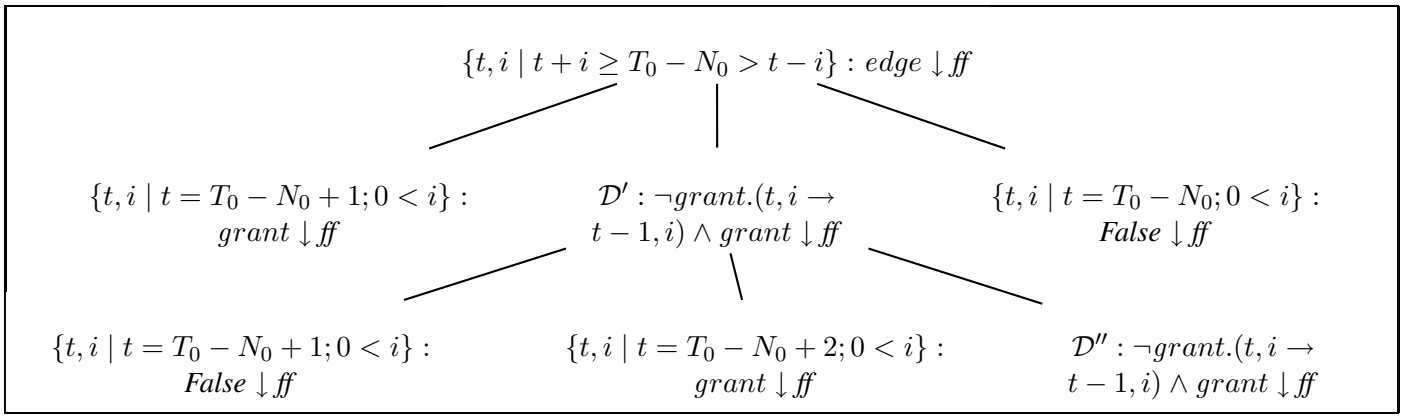
$$NotGrantOrUse = \begin{cases} \{t,i \mid t = T_0 - N_0; 0 < i \leq N_0\} : \neg grant \\ \{t,i \mid t \geq 0; 0 < i \leq P\} \setminus \{t,i \mid t = T_0 - N_0; 0 < i \leq N_0\} : \neg grant \vee \textit{ use} \end{cases}$$

Ainsi, si nous construisons l'arbre de dérivation de la formule $f_{NotGrantOrUse}$, nous obtenons la formule

$$\{t,i \mid t = T_0 - N_0; 0 < i\} : grant \downarrow \textit{ff}$$

La variable $grant$ est donc substituée par faux, en particulier dans la variable $edge$.

Or, sur le domaine $\{t,i \mid t + i \geq T_0 - N_0 > t - i\}$ la variable $edge$ vaut faux. Si nous construisons son arbre de dérivation, nous obtenons l'arbre présenté en figure 9.15.



$\mathcal{D}' = \{t,i \mid t + i \geq T_0 - N_0 > t - i\} \setminus \{t,i \mid t = T_0 - N_0 + 1; 0 < i\}$
$\mathcal{D}'' = \mathcal{D}' \setminus \{t,i \mid t = T_0 - N_0 + 2; 0 < i\}$

FIG. 9.15 – Arbre de dérivation de la formule $\{t,i \mid t + i \geq T_0 - N_0 > t - i\} : edge \downarrow \textit{ff}$, il y a deux nœuds semblables pour la variable $grant$.

Dans cet arbre, nous trouvons deux nœuds semblables concernant la variable $grant$. ce sont les nœuds $\{t,i \mid t = T_0 - N_0 + 1; 0 < i\} : grant \downarrow \textit{ff}$ et $\{t,i \mid t = T_0 - N_0 + 2; 0 < i\} : grant \downarrow \textit{ff}$. Il faut agrandir le domaine de $grant$. Cette étape est délicate car, nous n'avons pas ici un pseudo-pipeline. Nous devons donc utiliser une autre technique pour effectuer cet agrandissement. Cette technique sera expliquée en détail dans le paragraphe suivant concernant le domaine de *ask*. Nous nous contentons ici de remarquer que si $grant.(t, i \rightarrow t-1, i)[T_0 - N_0 + 1, I]$, où I est positif, vaut faux, alors $\neg grant.(t, i \rightarrow t - 1, i)[T_0 - N_0 + 1]$ vaut vrai et ainsi comme $edge[T_0 - N_0 + 1, I]$ vaut faux, $grant[T_0 - N_0 + 2, N_0]$ vaut faux. Autrement dit si $grant[T_0 - N_0 + 1, I]$ vaut faux, alors $grant[T_0 - N_0 + 2, I]$ vaut faux. En répétant le processus, nous pourrions montrer que $grant[T_0 - N_0 + 3, I]$ vaut faux. L'agrandissement de domaine se fait donc dans la direction $(t, i \rightarrow t + 1, i)$, et nous obtenons la formule : $\{t,i \mid t + i \geq T_0 - N_0; t - i \leq T_0 - N_0\} : grant \downarrow \textit{ff}$.

Nous représentons sur la figure 9.16 le domaine où nous connaissons les valeurs de $grant$.

La variable $grant$ est donc substituée par sa valeur dans toutes les expressions du système. La formule $AskUntilGrantSimp$ (9.9) est ainsi modifiée sur le domaine $\mathcal{D} = \{t,i \mid T_0 - N_0 \leq t \leq T_0 - N_0 + i\}$.

$$AskUntilGrantSimp = \begin{cases} \mathcal{D} \setminus \{t,i \mid T_0 \leq t \leq T_0 + 1; i = N_2\} : \neg \textit{ask}.(t, i \rightarrow t - 1, i) \vee \textit{ask} \\ \{t,i \mid t = T_0 + 1; i = N_2\} : \textit{ask}.(t, i \rightarrow t, i) \\ \{t,i \mid t = T_0; i = N_2\} : \textit{True} \end{cases}$$

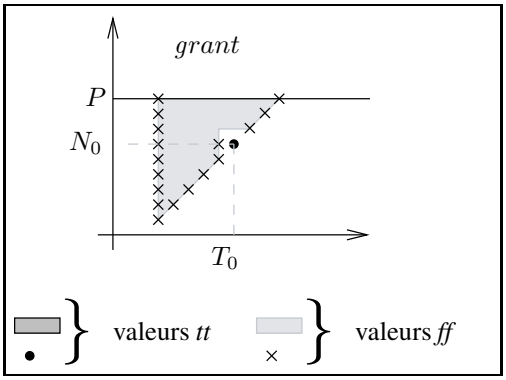


FIG. 9.16 – Domaines où les valeurs de $grant$ sont connues

Nous allons utiliser cette nouvelle définition pour étudier les valeurs de la variable *ask* sur le domaine $\mathcal{D}$.

Domaine de *ask* : Décrivons le processus de construction de l'arbre de dérivation (*cf.* figure 9.17) de la formule $f_{AskUntilGrantSimp}$ en nous restreignant au domaine $\mathcal{D}$:

- Tout d'abord la formule suivante est générée :

$$\{t, i \mid t = T_0 + 1; i = N_2\} : \mathit{ask} \downarrow \mathit{tt}$$

- La variable ask peut alors être substituée par la valeur *tt* en particulier dans la définition de la variable $AskUntilGrantSimp$.
- Ainsi, la formule suivante est générée :

$$\{t, i \mid t = T_0 + 2; i = N_2\} : \mathit{ask} \downarrow \mathit{tt}$$

- puis la construction de l'arbre s'arrête car deux nœuds semblables ont été rencontrés. Ce sont les nœuds $\{t, i \mid t = T_0 + 1; i = N_2\} : \mathit{ask} \downarrow \mathit{tt}$ et $\{t, i \mid t = T_0 + 2; i = N_2\} : \mathit{ask} \downarrow \mathit{tt}$

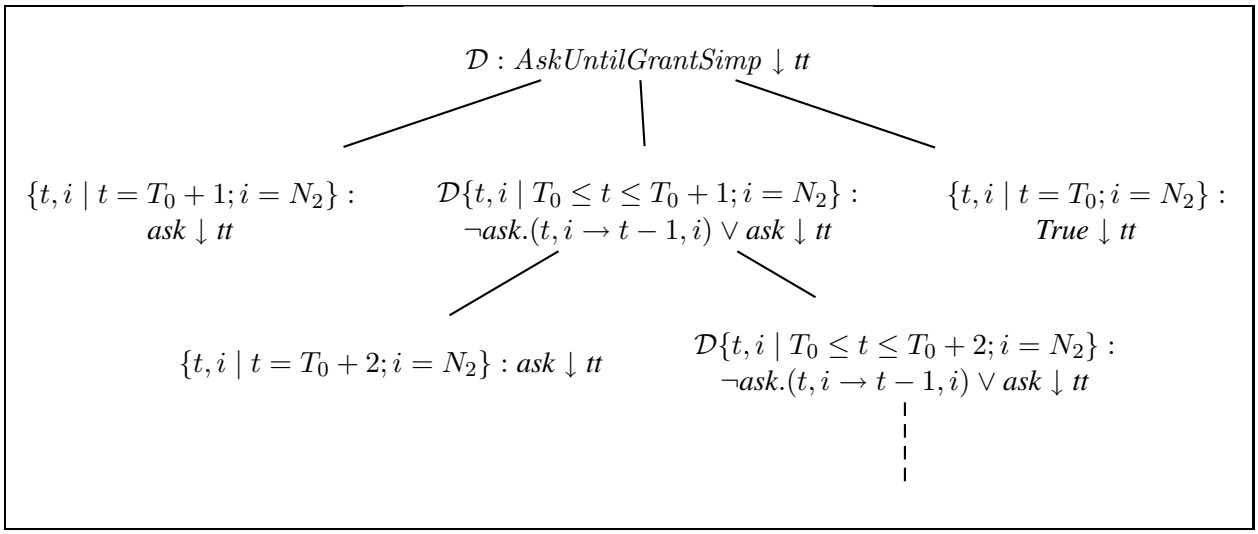


FIG. 9.17 – Arbre de dérivation de la formule $f_{AskUntilGrantSimp}$, il y a deux nœuds semblables concernant la variable *ask*.

On souhaiterait donc en déduire que sur le domaine $\mathcal{D} \cap \{t, i \mid t \leq T_0; i = N_2\}$, la variable ask vaut vrai. Il suffirait pour cela d'agrandir le domaine $\{t, i \mid t = t_0; i = N_0\}$ dans la direction de $(t, i \rightarrow t-1, i)$. Le problème est de trouver automatiquement la dépendance $(t, i \rightarrow t-1, i)$.

La difficulté réside dans le fait que nous avons une variable d'entrée : nous ne pouvons donc pas agrandir son domaine dans le sens de ses auto-dépendances. Nous devons donc calculer une dépendance. Puisque nous retrouvons deux nœuds semblables contenant la formule *ask*, nous allons rechercher quel est l'enchaînement d'opérations qui a permis de passer d'une formule à l'autre. Cet enchaînement nous permettra de calculer la dépendance servant à l'agrandissement.

Ici, l'enchaînement est simple :

- nous substituons ask par la valeur *ff* dans la définition de la variable $AskUntilGrantSimp$,
- puis, par utilisation de la règle de la négation et de la dépendance, nous obtenons de nouveau une formule concernant *ask*.

Reste maintenant à calculer la dépendance. Nous utilisons deux règles qui font intervenir des dépendances :

- la substitution par constante, pour laquelle nous calculons la préimage d'un domaine. Dans ce cas nous prenons l'inverse de la dépendance, et comme nous avons ici l'identité, son inverse est l'identité.
- puis la règle de la dépendance, où nous calculons cette fois ce l'image d'un domaine. Nous prenons donc ici la dépendance $(t, i \rightarrow t-1, i)$.

Nous composons toutes les dépendances et nous obtenons la dépendance $(t, i \rightarrow t-1, i)$.

Nous venons donc de présenter le principe de cet agrandissement. Le principe est donc :

- Recherche de l'enchaînement des règles,
- combinaison des dépendances.

Cependant, l'automatisation pose quelques problèmes au niveau de la recherche de l'enchaînement des règles. La difficulté vient de la substitution par constante, nous pouvons substituer une variable par une constante dans différentes variables, pour toutes ces variables modifiées nous devons vérifier si elles mènent à une feuille semblable et l'exploration de l'enchaînement des règles peut être combinatoire. Dans l'exemple que nous venons d'étudier, la substitution a lieu dans les variables tk et Or_ask. Cependant ces variables n'interviennent jamais dans l'arbre de dérivation de $AskUntilGrantSimp$, elles ne peuvent donc pas mener vers une feuille semblable de cet arbre ainsi l'exploration de l'arbre de preuve est immédiate.

Nous obtenons donc la formule $\{t, i \mid T_0 - N_0 \leq t \leq T_0 + i; i = N_2\} : ask \downarrow tt$. Si $N_2 \leq N_0$, alors nous montrons la formule $\{t, i \mid t = T_0 - N_0 + N_2 - 1; i = N_2\} : ask \downarrow tt$ et ainsi nous montrons que la formule $\{t, i \mid t = T - 0 - N_0 + N_2 - 1; i \leq N_0\} : ask \downarrow ff$ est invalide. Nous en déduisons donc que $N_2 \geq N_0$.

Au passage, nous pouvons remarquer que la formule $\{t, i \mid t = T_0 - N_0; i \leq N_0\} : ask \downarrow ff$ et nous avons donc prouvé la formule f_{prior} qui représente la propriété de priorité.

La variable *ask* est substituée par la constante *tt*, ce qui modifie la variable Or_Ask. Il suffit de propager la valeur pour obtenir $\{t \mid T_0 - N_0 \leq t \leq T_0 - N_0 - 1\} : requested \downarrow tt$.

Nous représentons sur la figure 9.18 le domaine où nous connaissons les valeurs de *ask*.

La variable *ask* est substitué par la constante *tt* dans toutes les variables, en particulier, dans la variable Or_ask. Cette variable est un pseudo-pipeline, et la valeur *tt* est absorbante pour Or_ask, elle est donc propagée dans le sens des dépendances. Ainsi nous obtenons que Or_ask vaut vrai sur le domaine $\{t, i \mid T_0 - N_0 \leq t \leq T_0 - 1; i = N\}$, ce qui correspond à avoir prouver la formule suivante sur la variable $requested$:

$$\{t \mid T_0 - N_0 < t \leq T_0 - 1\} : requested \downarrow tt$$

Nous avons maintenant dérivé toutes les formules du contexte et nous n'avons plus de tactiques pour continuer la dérivation. Nous reprenons donc maintenant la construction des arbres de preuve.

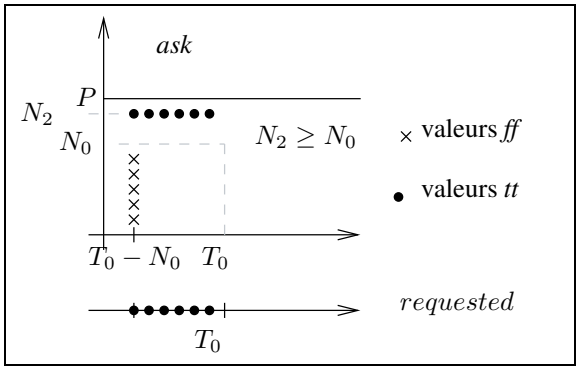


FIG. 9.18 – Domaines où les valeurs de *ask* sont connues

Nous devions prouver les formules (9.10).

En reprenant la construction de la formule concernant la variable Or_ask, nous obtenons une feuille fermée valide.

Reste encore à prouver les formules :

$$\{t, i \mid t = T_0 - N_0 - 1; i = 0\} : requested \wedge \neg used \downarrow \textit{ff}$$
$$\{t, i \mid T_0 - N_0 < t \leq T_0 - 1\} : \textit{use} \downarrow \textit{ff}$$
$$\{t, i \mid T_0 - N_0 + i \leq t \leq T_0 - 1; 1 \leq i \leq N_0 - 1\} : \textit{ask} \downarrow \textit{ff}$$
$$\{t, i \mid t = T_0 - 1; i = N_0\} : \textit{ask} \downarrow \textit{tt}$$

La reprise de la construction des arbres de preuve échoue, nous allons donc utiliser un raisonnement par l'absurde.

4.7 Preuves des propriétés : démonstration par l'absurde

Nous prouverons toutes les formules restantes par l'absurde. L'ordre des preuves n'a ici aucune importance. Nous commençons donc par la preuve de la formule :

$$\{t, i \mid T_0 - N_0 < t \leq T_0 - 1\} : \textit{use} \downarrow \textit{ff}$$

Preuve de la formule $\{t, i \mid T_0 - N_0 < t \leq T_0 - 1\} : \textit{use} \downarrow \textit{ff}$ **:** Pour effectuer des preuves par l'absurde nous devons avoir des formules locales. En effet, le but est de supposer qu'une formule est invalide et d'obtenir une contradiction pour prouver qu'elle est en fait valide. Si nous supposons la formule invalide, c'est que nous supposons sa négation valide, or la négation d'une formule n'est une formule que si nous avons une formule locale. Nous ne pouvons donc utiliser le raisonnement par l'absurde que sur des formules locales. La formule $\{t, i \mid T_0 - N_0 < t \leq T_0 - 1\} : \textit{use} \downarrow \textit{ff}$ n'est pas une formule locale, nous allons donc la transformer en formule locale. Pour cela, nous ajoutons des paramètres T_3, N_3 tels que $T_0 - N_0 < T_3 \leq T_0 - 1$ et $N_3 \leq P$. Notre formule est transformée en $\{t, i \mid t = T_3; i = N_3\} : \textit{use} \downarrow \textit{ff}$ et nous la supposons non valide, c'est-à-dire que nous supposons que la formule $\{t, i \mid t = T_3; i = N_3\} : \textit{use} \downarrow \textit{tt}$ est valide et nous utilisons cette hypothèse dans la formule $f_{NotUseIfNotGrant}$ du contexte.

En effectuant une substitution par constante de *use* dans la variable $NotUseIfNotGrant$, nous obtenons les définitions suivantes :

$$NotUseIfNotGrant = \{t, i \mid t = T_3; i = N_3\} : OnceGrantSinceNotUse$$

La construction de l'arbre de $f_{NotUseIfNotGrant}$ permet de générer la formule $\{t, i \mid t = T_3; i = N_3\} : OnceGrantSinceNotUse \downarrow \textit{tt}$ et $OnceGrantSinceNotUse$ est substitué par sa valeur.

De plus nous avons montré que la variable $grant$ valait faux sur le domaine $\{t, i \mid T_0 \leq t \leq T_0 - N_0 + i\} \setminus \{t, i \mid t = T_0; i = N_0\}$ représenté sur la figure 9.16. Cette variable a donc été substituée par la constante *False* en particulier dans la variable $OnceGrantSinceNotUse$ qui se trouve maintenant définie sur le domaine $\mathcal{D} = \{t, i \mid T_0 \leq t \leq T_0 - N_0 + i\} \setminus \{t, i \mid t = T_0; i = N_0\}$ par :

$$OnceGrantSinceNotUse = \begin{cases} \mathcal{D} \setminus \{t, i \mid t = T_3; i = N_3\} : & (\ \textit{use} \wedge \\ & OnceGrantSinceNotUse.(t, i \rightarrow t - 1, i)) \\ \{t, i \mid t = T_3; i = N_3\} : & \textit{True} \end{cases}$$

La variable $OnceGrantSinceNotUse$ ainsi redéfinie est un pseudo-pipeline. Au point (T_3, N_3), cette variable vaut vrai. Nous pouvons appliquer l'opérateur d'agrandissement sur le domaine réduit à un point contenant (T_3, N_3). Soit $\mathcal{D}'$ le domaine correspondant à l'agrandissement. Il est défini par :

$$\{t, i \mid T_0 \leq t \leq T_3; i = N_3\}$$

Il suffit ensuite de construire l'arbre de dérivation $OnceGrantSinceNotUse$ sur ce domaine. Nous obtenons alors une feuille décrivant la formule $\mathcal{D}' :$ *use* $\downarrow$ *tt*. Reste à vérifier si cette formule est compatible avec les formules que nous avons sur *use*. Dans le contexte, nous avons la formule $\{t, i \mid t = T_0\} :$ *use* $\downarrow$ *ff*. Quelle que soit la valeur de T_3, les deux formules sont incompatibles. Nous avons donc obtenu une contradiction, et nous avons ainsi montré la formule $\{t \mid T_0 - N_0 \leq t \leq T_0 - N_0 - 1\} : used \downarrow$ *ff*.

Nous pouvons maintenant supprimer les paramètres T_3 et N_3. Il suffit de supprimer toutes les occurrences de T_3 et de N_3 dans le système et dans les arbres de preuve.

Preuve de la formule $\{t, i \mid t = T_0 - N_0 - 1; i = 0\} : requested \wedge \neg used \downarrow$ *ff* **:** Nous essayons donc de montrer que

$$\{t, i \mid t = T_0 - N_0 - 1\} : requested \wedge \neg used \downarrow \textit{tt}$$

Cette expression correspond à la définition de $tk[T_0 - N_0, 0]$, comme tk est un pseudo-pipeline, et que la valeur *tt* est la valeur neutre, la valeur *tt* se propage dans le sens des dépendances. Nous allons donc supposer que la variable tk est vraie en un point t, i tel que $t - i = T_0 - N_0 - 1$. Nous devons donc ajouter deux paramètres T_3 et N_3 telles que $T_3 - N_3 = T_0 - N_0 - 1$ et nous essayons de prouver la formule correspondante. En construisant son arbre de dérivation, nous générons les formules :

$$\{t, i \mid t = T_3; i = N_3\} : q \downarrow \textit{tt}$$
$$\{t, i \mid t - i = T_3 - N_3; t < T_3\} : tk \downarrow \textit{tt}$$

Ces deux formules sont ajoutées. En substituant q par la valeur *tt*, nous obtenons une contradiction pour toute valeur de T_3. Reste alors à vérifier la formule suivante :

$$\{t, i \mid t - i = T_3 - P; t < T_3\} : tk \downarrow \textit{tt}$$

Cette formule mène aussi à une contradiction.

Ainsi pour toute valeur de T_3 et de N_3 nous obtenons une contradiction. Nous avons ainsi montré par l'absurde la formule suivante :

$$\{t, i \mid t = T_0 - N_0 - 1; i = 0\} : tk \downarrow \textit{ff}$$

La preuve des deux dernières formules se fait là encore par l'absurde. et nous obtenons immédiatement des contradictions. Nous avons ainsi montré l'invariant.

Nous pouvons donc montrer maintenant les propriétés d'exclusion mutuelle et de priorité.

4.8 Preuve de la priorité

Nous avons donc prouvé notre invariant. Le problème est maintenant de prouver les propriétés d'exclusion mutuelle et celle de priorité. Nous devons donc prouver :

$$\mathcal{F}_{contexte}, \mathcal{F}_{inv}; S_{arbitre} \models^{\star} \mathcal{F}_{prior} \wedge \mathcal{F}_{mutex}$$

Nous ne présenterons que la preuve de la propriété de priorité, celle d'exclusion mutuelle étant similaire. Nous allons donc fournir une preuve pour :

$$\mathcal{F}_{contexte}, \mathcal{F}_{inv}; S_{arbitre} \models^{\star} \mathcal{F}_{prior}$$

Nous rappelons que $\mathcal{F}_{prior}$ est définie par :

$$\mathcal{F}_{prior} = f_{prior1} \Rightarrow f_{prior2}$$

où f_{prior1} et f_{prior2} sont définies par

$$\begin{aligned} f_{prior1} &= \{t, i \mid t = T_2, i = I_2\} : edge \downarrow tt \\ f_{prior2} &= \{t, i \mid t = T_2, i < I_2\} \cup \{t, i \mid t = T_2, i > I_2\} : ask \downarrow ff \end{aligned}$$

La propriété $\mathcal{F}_{prior}$ est une implication, nous utilisons donc la règle de l'implication, nous faisons donc passer la formule f_{prior1} dans le contexte.

Nous allons maintenant travailler sur le contexte. La technique que nous proposons ressemble à une technique d'unification.

Notre contexte contient entre autres les deux formules $\mathcal{F}_{inv}$ et f_{prior1}. La formule $\mathcal{F}_{inv}$ est une implication, la formule à droite de l'implication est $f_{inv_E} = \{t, i \mid t = T_0; i = N_0\} : edge \downarrow tt$. Cette formule est donc semblable à f_{prior1}. L'unique différence vient des paramètres : l'une des formules est paramétrée par T_2 et N_2 et l'autre par T_0, N_0. La portée des paramètres T_0 et N_0 est limitée à la gauche du signe $\models^{\star}$, contrairement aux paramètres T_2 et N_2. En effet, la formule $\mathcal{F}_{inv}$ est quantifiée universellement par T_0 et N_0, la portée des quantificateurs se limite donc à cette formule, et si nous revenons au niveau sémantique, notre problème est de prouver :

$$\mathcal{F}_{contexte}, \mathcal{F}_{inv}; S_{arbitre} \models\!\!\!\models^{\star} \mathcal{F}_{prior} \wedge \mathcal{F}_{mutex}$$

Cette expression signifie simplement que si les formules $\mathcal{F}_{contexte}$ ou $\mathcal{F}_{inv}$ sont valides alors $\mathcal{F}_{prior} \wedge \mathcal{F}_{mutex}$ est valide. Nous avons donc une implication. Les paramètres T_0 et N_0 sont quantifiés universellement à gauche de l'implication, on ne peut donc pas les sortir de l'implication et les mettre en tête. Leur portée est donc limitée à la formule $\mathcal{F}_{inv}$. La formule $\mathcal{F}_{inv}$ représente un ensemble de formules associées à chaque valeur des paramètres T_0 et N_0, on peut donc la voir comme une conjonction de chacune des formules de l'ensemble. Nous allons utiliser la formule $\mathcal{F}_{inv}$ pour une unique valeur des paramètres T_0 et N_0 : pour $T_0 = T_2$ et $N_0 = N_2$. Les formules f_{inv_E} et f_{prior1} sont « unifiées », elles représentent une seule et même formule. Nous générons ainsi les formules $f_{inv_{T,T_2,N_2}}$ et $f_{inv_{F,T_2,N_2}}$. Si nous reprenons les arbres de dérivation, nous prouvons alors la formule f_{prior2}.

Dans cette technique, nous sommes confrontés à plusieurs difficultés, tout d'abord au niveau quantification et portée des quantifications. Pour cela, nous allons définir une sémantique des quantificateurs, puis définir des règles de preuve liées à ses quantificateurs. La difficulté réside principalement dans le fait que nous représentons les quantificateurs par des paramètres. Dans le modèle polyédrique, l'ordre des paramètres n'a aucune importance, ce qui n'est bien sûr pas le cas pour les quantificateurs. De plus dans le modèle polyédrique, la portée des paramètres s'étend à tout le système, ce qui n'est là encore pas le cas des quantificateurs, certains sont limités au contexte, d'autre à la partie droite, et enfin certains

ont une portée sur tous les séquents, il nous faudra donc intégrer cette information à notre sémantique. Actuellement, nous avons formalisé les quantificateurs se trouvant en partie droite, nous allons devoir généraliser cette formalisation au contexte et prendre en compte la notion de portée.

La seconde difficulté consiste à repérer que les formules f_{prior1} et f_{inv_E} peuvent être unifiées et trouver la valeur des paramètres. Nous utiliserons ici un critère syntaxique pour repérer que les expressions des deux formules sont identiques ainsi que la valeur des formules, puis nous effectuons des calculs sur les domaines et des projections sur les indices T_0, N_0, T_2 et N_2 pour connaître la relation entre ces paramètres.

5 Conclusion

Dans ce chapitre nous avons tout d'abord esquissé un cadre plus large de spécification puis nous avons présenté différentes tactiques de preuves :

- principe de récurrence,
- recherche de pseudos-pipelines,
- preuve par l'absurde,
- intersection de formules,
- nouvelles techniques d'agrandissement,
- ajout de paramètres pour représenter des quantifications,
- unification polyédrique

Ces tactiques sont en cours de formalisation et nous avons ici simplement présenté les idées générales.

La correction de ces méthodes reposera principalement sur la correction de notre représentation des quantificateurs.

Une fois ces techniques implémentées nous nous intéresserons à fournir des tactiques plus complexes combinant ces techniques. Nous ne pensons pas pouvoir fournir un algorithme alliant toutes ces techniques. Cependant certaines techniques peuvent être utilisées complètement automatiquement et être insérées à notre algorithme de preuve. Ces techniques sont la recherche de pseudos-pipelines, les techniques d'agrandissement, l'unification, et l'ajout de paramètres. Cette dernière technique devra cependant être utilisée avec précaution afin de ne pas ajouter trop de paramètres. Les techniques de récurrence sont quant à elles plus difficiles à utiliser. Pour la récurrence la principale difficulté est la recherche du ou des paramètres de récurrence. La preuve par l'absurde pourrait, peut-être, être utilisée en dernier recours.

Dans les travaux que nous avons présentés dans ce travail, nous avons plusieurs fois précisé qu'il n'était pas utile de reconstruire l'arbre de preuve et qu'il suffisait ensuite de vérifier que les formules étaient compatibles entre elles. Cette remarque devrait permettre d'améliorer la rapidité de construction des preuves en évitant à notre outil de construire plusieurs fois la même preuve. Elle pose cependant des difficultés d'implémentation. Cela demande en particulier de faire une première construction des arbres de preuve de manière complètement indépendante, sans tenir compte de ce qui a été prouvé. En particulier pour chaque preuve, il faudrait travailler sur le système initial qui n'a pas été modifié. Dans un premier temps, nous ne tiendrons donc pas contre de cette remarque et nous reconstruirons les arbres de preuve.

Les différentes techniques que nous avons présentées ici peuvent donner l'impression que le processus de preuve est extrêmement compliqué. Ce processus n'est en réalité pas si difficile. Nous avons au cours de ce chapitre présenté différentes techniques et nous avons du pour cela rentrer dans des détails techniques. Ces techniques étaient présentées en même temps que le processus de preuve. Le principe de preuve n'est en lui même pas très compliqué : tout d'abord nous construisons les arbres de preuves des formules à droite du séquent puis les arbres de dérivation, sur les formules obtenues lors de la dérivation, nous appliquons les différentes tactiques vues dans ce chapitre jusqu'à ce qu'on ne puisse plus obtenir

de nouvelles formules. Puis nous reprenons la construction des arbres de preuve. Si cette construction échoue, nous utilisons alors un raisonnement par l'absurde.

Nous présentons ici un algorithme envisageable alliant les techniques pouvant être utilisées automatiquement.

- *Répéter*
  - *construire les arbres de preuve,*
  - *propager les valeurs des pseudos-pipeline*

  jusqu'à ce qu'il n'y ait plus de modifications
- *Répéter*
  - *construire les arbres de dérivation,*
  - *substituer les variables d'entrées par leurs valeurs,*
  - *ajouter des paramètres, propager les valeurs des pseudos-pipelines et agrandir les domaines,*
  - *simplifier les domaines des paramètres et ajouter les hypothèses obtenues après simplification des domaines des paramètres, intersecter ces nouvelles formules avec les formules déjà présentes dans le contexte.*

  jusqu'à ce qu'il n'y ait plus de modifications

Cet algorithme consiste donc en une alternance de constructions d'arbres de preuve et de transformations et d'enrichissement sur le contexte. Nous serons cependant confrontés à l'efficacité d'un tel algorithme. En effectuant systématiquement certaines opérations aussi bien sur le contexte que sur les formules à droite du séquent, notre outil risque d'être moins efficace.

Il pourrait donc être plus judicieux de développer un outil de preuve semi automatique, où l'utilisateur pourrait choisir d'utiliser certaines règles de preuve sur certaines formules, ou alors pourrait programmer des tactiques de preuve comme c'est effectué dans les prouveurs de théorèmes. La création de schémas de preuve spécifiques pourrait aussi être développée. Ces schémas de preuve pourraient être utilisés en fonction de la structure du système étudié et des propriétés à prouver. Nous proposons donc de développer un outil proche des prouveurs de théorèmes s'appuyant sur notre « logique polyédrique », notre système de preuve et l'algorithme de construction des preuves. Les différentes techniques que nous avons vues dans ce chapitre seraient utilisées pour créer des schémas de preuve.

Troisième partie

Conclusion

Conclusion

Les travaux réalisés au cours de cette thèse étaient orientés vers la vérification formelle de propriétés de sûreté dans le cadre de la conception de systèmes enfouis.

Le but de ce travail était de prouver formellement des propriétés de contrôle sur des systèmes réguliers paramétrés dans le modèle polyédrique. Nous avons montré que ce modèle était adapté pour la vérification formelle de telles propriétés.

Contributions

Nous avons donc développé au cours de ce travail une « logique polyédrique » puis des algorithmes pour automatiser la construction des preuves.

Dans le chapitre 4, nous avons présenté des techniques de spécification et un système de règles de preuve qui nous ont permis d'introduire les règles de preuve spécifiques à notre modèle : la substitution par constante et la recherche d'auto-dépendance.

- *La substitution par constante :* Cette règle s'appuie sur un raisonnement par récurrence sur l'indice représentant le temps (ce n'est donc pas du temps réel), elle permet de substituer simultanément un ensemble d'instances par leur valeur.
- *Recherche de motifs – Recherche d'auto-dépendance* La recherche de motif permet de substituer une expression polyédrique par une variable, elle est utilisée dans la recherche d'auto-dépendance qui permet de transformer une variable non récursive en une variable récursive afin de pouvoir utiliser ensuite le principe de substitution par constante.

Ce chapitre était donc consacré à l'élaboration d'une « logique polyédrique » pour spécifier nos propriétés, et d'un système de règles de preuve associées à notre logique. Une fois ce cadre fixé, nous nous sommes intéressés à l'obtention d'un algorithme de construction des preuves de nos propriétés.

Dans l'optique de construire cet algorithme de preuve automatique, nous avons étudié dans le chapitre 5 l'itération alternée des règles de preuve de substitution par constante et recherche de motifs. Nous avons caractérisé les cas où l'itération était finie, et les cas où elle était infinie ce qui nous a permis de déterminer les cas où nous pouvons itérer ces deux règles de preuve pour construire notre preuve. Ayant déterminé les cas où l'itération était finie et ceux où elle était infinie, nous avons donc pu étudier le problème d'automatisation.

Le chapitre 6 a ainsi été consacré à l'implémentation du processus de preuve. Pour cela, nous avons tout d'abord défini une structure de preuve (un arbre de preuve), puis nous avons enrichi le système de preuve, ce qui nous a permis de construire un algorithme de preuve. Pour assurer la terminaison de cet algorithme, nous avons dû arrêter la construction des preuves lorsque nous retrouvions deux sous-buts semblables à prouver, alors que des règles de preuve pouvaient encore être appliquées sur ces sous-buts. En effet, lorsque nous obtenons deux sous-buts semblables la construction de la preuve est infinie, pour schématiser le problème, la preuve est trop lente. Notre but a donc été de fournir un opérateur pour accélérer la construction de la preuve.

Le chapitre 7 était tourné vers la création d'un tel opérateur d'accélération permettant de continuer la construction des preuves. Cet opérateur est en fait un opérateur d'agrandissement de domaine. L'opé-

rateur d'agrandissement que nous avons développé s'est montré particulièrement adapté pour l'agrandissement des domaines de certains signaux. Ces signaux sont appelés pseudos-pipelines, ils jouent un rôle prépondérant dans les heuristiques d'agrandissement que nous avons ensuite développées. En effet, lorsque ces signaux sont présents, nous pouvons facilement agrandir les domaines. Dans tous les systèmes que nous avons étudiés, ces signaux étaient présents. Nos heuristiques d'accélération de preuve donnent donc en pratique de bons résultats.

Nous avons exposé dans le chapitre 8 les applications de ces méthodes. Le premier exemple consistait à étudier un signal de contrôle sur un filtre adaptatif. Cet exemple est très simple et la preuve est donc construite automatiquement. Le but était de prouver qu'un registre ne pouvait plus être atteint après une phase d'initialisation. C'est typiquement le genre de signaux qui sont ajoutés au cours de la synthèse et sur lequel nous serons amenés à travailler. Le second exemple est un exemple plus complexe. Il nous a permis d'utiliser plus exhaustivement notre outil de preuve. Le but était de prouver que dans un produit de matrices et de vecteurs, le résultat de sortie ne dépendait que des coefficients en entrée. La preuve est actuellement semi-automatique, mais sera complètement automatisée dès que l'implémentation sera terminée.

Le chapitre 9 est un chapitre prospectif. Nous avons présenté les développements actuels et à venir de notre système de preuve. Nous avons tout d'abord présenté un nouveau cadre de spécification des propriétés beaucoup plus riche, puis nous avons présenté des tactiques de preuve permettant de prouver des propriétés sur un arbitre matériel.

Comparaison

Jusqu'à présent, les travaux menés dans le cadre de la vérification formelle de systèmes décrits dans le modèle polyédrique étaient orientés soit vers la preuve d'équivalence [28, 8, 93] soit vers la preuve de propriétés fonctionnelles [22, 21].

A notre connaissance, aucun travail de vérification formelle dans le cadre du modèle polyédrique et n'utilisant que le modèle polyédrique n'était orienté vers la vérification de propriétés de sûreté (les propriétés fonctionnelles peuvent être des propriétés de sûreté, mais les travaux menés sur le sujet utilisent des prouveurs de théorème).

Les techniques de vérification mise en œuvre dans ces différents travaux sont assez différentes.

Dans les travaux de Cachera *et al.* [22, 21], la vérification se fait en partie à l'aide d'un prouveur de théorèmes Coq [96] et PVS [92]. Un certain nombre de stratégies ont été développées et sont appliquées automatiquement sur les propriétés. Pour certaines propriétés très simples, une partie du travail est effectuée directement dans le modèle polyédrique. Les techniques utilisées dans ce modèle sont des techniques de substitutions des variables par leur définition. Ces techniques ne se sont cependant pas montrées satisfaisantes pour prouver les propriétés de l'arbitre matériel.

Dans les travaux de Barthou *et al.* [8], la preuve d'équivalence se fait en associant à chaque paire de systèmes un automate à états. Prouver l'équivalence revient à résoudre un problème d'atteignabilité dans l'automate. La construction de la preuve se fait sans utiliser les caractéristiques du modèle polyédrique, elle ne prend en particulier pas en compte la sémantique des opérateurs. Nous n'avons pas ce problème puisque nos règles de preuve reposent principalement sur la sémantique de ce modèle. De plus ces travaux sont restreints à la preuve de l'équivalence.

Les travaux de Shashidar *et al.* [93] sont les seuls à être effectués directement dans le modèle polyédrique. Il est donc intéressant de les comparer aux nôtres. Leur but est d'effectuer là encore des preuves d'équivalence entre deux systèmes dont l'un a été obtenu par transformation de l'autre en utilisant des techniques de synthèse. Les auteurs commencent par se restreindre aux systèmes ordonnancés. Nous effectuons la même restriction. Puis, ils restreignent les transformations possibles entre deux systèmes. Nous n'avons pas ce problème puisque nous ne prouvons pas l'équivalence mais simplement des propriétés de sûreté. Notre système peut donc être obtenu par n'importe quelle transformation de syn-

thèse. Pour vérifier l'équivalence, ils construisent des chaînes de registres telles que chaque registre de la chaîne reçoit la valeur du registre précédent dans la chaîne. Cette technique correspond simplement à la substitution d'une variable par sa définition. Elle correspond donc à notre règle de l'équation. Notre cadre de preuve est donc beaucoup plus riche.

Dans les différents travaux effectués, la substitution d'une variable par sa définition est fondamentale dans le processus de preuve. Notre règle de substitution par constante n'en est qu'une variante. Nous utilisons de plus largement la substitution par variable aussi bien dans la recherche d'auto-dépendance que dans la règle de l'équation. Nous avons en revanche introduit une nouvelle approche : nous fournissons un système de règles de preuve, et nous ne posons d'autres restrictions que celles d'avoir un système ordonnancé dont les dépendances sont uniformes. Cette dernière restriction n'en est pas vraiment une puisqu'il est possible de transformer automatiquement un système d'équations récurrentes affines en équations récurrentes uniformes [83]. Ainsi notre système de preuve peut potentiellement s'appliquer sur un système d'équations récurrentes affines.

Si maintenant nous comparons nos travaux avec ceux effectués dans un cadre plus général (celui de la vérification formelle de systèmes paramétrés), nous sommes confrontés à des problèmes similaires comme l'utilisation de techniques d'accélération. Nous effectuons nous aussi des restrictions, c'est-à-dire que nous ne travaillons que sur des systèmes qui peuvent s'exprimer dans le modèle polyédrique. En revanche, nous ne faisons aucune abstraction sur les systèmes étudiés, le modèle polyédrique fournit naturellement une représentation compacte, ainsi nous ne manipulons pas d'ensemble infini de calculs. L'abstraction fournie par le modèle polyédrique a de plus l'avantage de ne pas être une approximation contrairement aux abstractions utilisées dans les différents travaux de vérification formelle de systèmes paramétrés qui utilisent des sur-approximations. Ainsi quand nous prouvons qu'une propriété est fausse, nous savons qu'elle est fausse pour le système étudié et que cela n'est pas lié à l'abstraction utilisée.

Complexité et Efficacité

La complexité générale de l'algorithme que nous avons implémenté est exponentielle en nombre de sous-expressions du système. Cependant, notre implémentation est naïve. Si nous implémentons l'algorithme proposé dans le chapitre 6, la complexité de la construction de l'arbre devient quadratique. L'efficacité de notre algorithme pourra donc être facilement améliorée. La complexité des différentes règles de preuve est aussi exponentielle soit en nombre de contraintes soit en nombre de dimensions selon les opérations effectuées.

Nos règles de preuve ont donc toutes été implémentées en Mathematica. Nous avons utilisé une machine SUN Ultra SPARC de 502Mhz pour tester notre outil de preuve. Les temps de calcul que nous obtenons actuellement ne sont absolument pas significatifs, ils sont liés à l'implémentation récursive de notre algorithme. Ils ne pourront qu'être améliorés. Les preuves sur le filtre adaptatif ont été effectuées en moins d'une seconde.

En ce qui concerne le produit matrice vecteur, les résultats sont nettement moins performants actuellement. Pour la preuve de la périodicité, seize formules sont prouvées. Notre outil de preuve effectue les calculs en environ 72 heures. Il est cependant important de noter, que le calcul termine avec une solution correcte, et qu'il n'y a pas de dépassement de mémoire. De plus, ce temps de calcul est lié à notre implémentation exponentielle et sera donc réduit avec une nouvelle implémentation. Pour la preuve sur une période, le temps de calcul est d'une demi-heure.

En ce qui concerne l'arbitre, nous ne pouvons pour le moment pas évaluer le temps des calculs. Nous ne proposons pour le moment qu'une méthode semi automatique. De plus, de nombreuses tactiques ne sont pas encore implémentées et doivent donc être effectuées par l'utilisateur.

L'algorithme implémenté nécessite encore un important travail d'optimisation. Nous avons indiqué ici plusieurs pistes possibles.

Perspectives

Les perspectives de ce travail sont nombreuses. Nous en présentons quelques-unes.

Implémentation et bibliothèque polyédrique L'implémentation des différentes règles de preuve que nous avons présentées dans ce document suppose que les calculs effectués par la bibliothèque polyédrique se font dans $\mathbb{Z}^n$. Cette supposition est en théorie fausse, les calculs sont effectués dans $\mathbb{Q}^n$. Cependant, cette approximation ne nous a pas en pratique conduit à des erreurs. En effet, le problème posé par cette approximation n'intervient que dans le cas de l'axiome AX2 qui concerne les domaines vides. Un domaine peut être vide dans $\mathbb{Z}^n$ sans être vide dans $\mathbb{Q}^n$. Pour lever cette approximation, il suffit pour cela d'utiliser d'autres outils comme la bibliothèque Omega [53], PIP [35],ou les polynômes d'Ehrhardt [98]. Il faudra donc étudier comment prendre en compte ces outils dans notre système et quand les utiliser, leur utilisation pouvant être couteuse. De plus cette approximation peut avoir un rôle plus important quand nous ajoutons des paramètres et que nous souhaitons savoir pour quelles valeurs de ces paramètres, les formules sont valides. Il sera judicieux dans ce cas de prendre en compte la forme des domaines selon la valeur des paramètres pour déterminer précisémment pour quelles valeurs des paramètres les domaines sont vides.

Efficacité et Implémentation L'implémentation que nous avons effectuée doit être partiellement refaite pour améliorer l'efficacité. Il en est de même pour l'implémentation de la règle de recherche d'auto-dépendance qui correspond au cas général et ne tient pour le moment pas compte des différentes heuristiques que nous avons développées dans le chapitre 5. Une fois cette implémentation refaite, nous devrions obtenir des résultats meilleurs pour la vérification du produit matrice-vecteur. Parallèlement, nous devons maintenant terminer l'implémentation des différentes tactiques présentées dans le chapitre 9.

Quantificateurs et Paramètres Cette perspective a été esquissée dans le chapitre 9. L'idée est de représenter les indices quantifiés existentiellement ou universellement à l'aide de paramètres. Ce travail est actuellement en cours de formalisation et d'implémentation. Ce travail soulève de nombreuses difficultés, tout d'abord sur la définition de la sémantique : comme tous les indices quantifiés sont représentés par des paramètres, comment prendre en compte la portée des quantificateurs, leur ordre (dans le modèle polyédrique l'ordre des indices n'a aucune importance). La seconde difficulté se situe au niveau de l'efficacité car nous rajoutons des paramètres à nos systèmes et donc nous augmentons la complexité de notre algorithme qui dépend entre autre de la dimension des domaines. Pour éviter ce problème, nous proposons de limiter l'introduction des paramètres au moment où ils sont nécessaires dans la preuve et de les supprimer le plus vite possible. Le problème est de déterminer à partir de quand ils ne sont plus nécessaires.

Combinaison de motifs Lors de la construction de l'arbre de preuve, et lorsque nous obtenons des feuilles pendantes pour lesquelles la recherche d'auto-dépendance échoue, nous ajoutons un motif. Ces motifs peuvent être combinés entre eux pour obtenir de nouvelles informations sur les variables du système. Regardons pour cela l'exemple suivant :

$$\langle W_1, A.(t, i \to t-1, i) \vee B.(t, i \to t-1, i), \{t, i \mid t = i\}, tt\rangle$$
$$\langle W_2, A.(t, i \to t-1, i) \vee \neg B.(t, i \to t-1, i), \{t, i \mid t = i\}, tt\rangle$$

L'idée est de définir un système d'équations booléennes et de le résoudre. En combinant les deux motifs, nous obtenons facilement que pour tout environnement ρ,

$$\models_\rho \{t, i \mid t = i\} : W_1 \downarrow tt \wedge \models_\rho \{t, i \mid t = i\} : W_2 \downarrow tt \iff \models_\rho \{t, i \mid t + 1 = i\} : A \downarrow tt$$

Cet exemple est un cas très simple. Supposons maintenant que nos deux motifs soient définis de la manière suivante :

$$\langle W_1, A.(t, i \rightarrow t-2, i) \vee B.(t, i \rightarrow t-2, i), \{t, i \mid t-1 = i\}, tt \rangle$$
$$\langle W_2, A.(t, i \rightarrow t-1, i) \vee \neg B.(t, i \rightarrow t-1, i), \{t, i \mid t = i\}, tt \rangle$$

Le problème est aussitôt plus difficile, il faut tout d'abord se ramener au même domaine, puis il faut tenir compte des dépendances. Ce problème est en fait équivalent au précédent, le premier motif a été simplement translaté.

Nous pensons donc qu'il serait utile de développer des outils pour rechercher l'information concernant les variables du système contenue dans ces listes de motifs. Ainsi en utilisant cette méthode dans la preuve des propriétés de l'arbitre, nous aurions pu éviter certains raisonnements par l'absurde.

Ce travail ne présente pas de réelles difficultés théoriques, il pourra être étudié à court terme.

Preuve d'équivalence Dans le chapitre 5, nous avons vu que la recherche d'auto-dépendance pouvait être utilisée pour effectuer des preuves d'équivalence. La méthode que nous proposons n'est actuellement pas efficace. Si l'on souhaite effectuer des preuves d'équivalence, il faudra développer des outils spécifiques et de nouvelles tactiques pour améliorer l'efficacité de notre outil de preuve dans le cadre de la preuve d'équivalence. Ce projet ne fait pour le moment pas partie de nos priorités. Il nécessitera un travail important et sera effectué à plus long terme.

Localisation des erreurs Nous avons montré dans le chapitre 6 que dans certains cas nous obtenions des contre-exemples, il serait intéressant d'utiliser ces contres-exemples pour aider l'utilisateur à trouver l'erreur du système. Il faudra pour cela remonter la construction de l'arbre de preuve et étudier les opérations effectuées. En fonction, des opérations utilisées on peut retrouver soit des erreurs d'initialisation, soit des erreurs de dépendances. Ce travail pourra être effectué à moyen terme, il ne présente pas de difficultés particulières.

Treillis de polyèdres Nous revenons ici sur l'échec de l'itération de la recherche d'auto-dépendance et de la substitution par constante lorsque le motif et l'expression définissant la variable qui est substituée sont définis par deux opérations distinctes. Nous supposons que nous voulons montrer la formule suivante $\mathcal{D} : W \downarrow v$, où W est définie par le motif $D_W : X.d_1 \between_1 X.d_2$ et où $\between_1$ est un opérateur booléen tel que la valeur v est absorbante, et X une variable polyédrique définie par la $\between_2$-expression suivante($\between_2 \neq \between_1$), (*cf.* section 1.2, chapitre 5) :

$$X \quad = \quad \mathcal{D}_1 : X.\delta_1 \between_2 X.\delta_2$$

La solution que nous envisageons ici est la suivante : générer de nouvelles formules en utilisant des $\mathbb{Z}$-polyèdres. Intuitivement, l'idée est de rechercher une répartition minimale des valeurs de X de sorte que W vale v sur tout son domaine. Cette répartition permet d'obtenir un domaine $\mathcal{D}'$, et il reste alors à vérifier la formule $\mathcal{D}' : X \downarrow tt$. Pour obtenir le domaine $\mathcal{D}'$, nous pourrons utiliser les résultats existants sur les Systèmes d'Additions de Vecteurs avec États (SAVE [87]) (un automate dont les arcs sont des vecteurs de $\mathbb{Z}^n$), en particulier des résultats sur l'atteignabilité. En effet, comme nous utilisons des dépendances uniformes, ces dépendances peuvent être vues comme des vecteurs de $\mathbb{Z}^n$. Les SAVE pourront aussi être utilisés pour effectuer des agrandissements de domaines, dans le cas où ces domaines sont des $\mathbb{Z}$-polyèdres.

Propriétés de contrôles sur des signaux entiers À plus long terme, des travaux pourront être menés sur la preuve de propriétés de contrôle sur des signaux entiers. Une première approche serait d'utiliser les polyèdres d'une manière plus traditionnelle pour modéliser les valeurs des instances comme dans les travaux menés par Henzinger *et al.* autour de HYTECH [48], ceux de Asarin *et al.* autour de d/dt [6], ceux de Silva *et al.* autour de *CheckMate* [94] ou enfin ceux de Alur *et al.* autour de *CHARON* [3]. Les polyèdres permettent ici de représenter des ensembles de valeurs entières dans des automates hybrides et fournissent ainsi une abstraction d'une classe restreinte de systèmes paramétrés infinis. L'idée serait donc de rajouter une dimension non contrainte aux domaines des variables pour représenter la valeur de chaque instance, puis de calculer des contraintes sur cette dimension qui représenteraient la propriété à prouver.

Propriétés de vivacité Enfin, il pourrait être intéressant d'étudier des propriétés de vivacité. Ce problème est beaucoup plus difficile, il est lié à notre représentation des quantificateurs. Nous ne proposons aucune piste actuellement.

Annexe A

Démonstrations

1 Système de règles de preuve

Proposition 4.22 *Soit S un système, soient X et Y deux variables de ce système telles que :*

$$X = f$$
$$Y = e$$

où f et e sont deux expressions polyédriques, on a alors l'équivalence sémantique suivante :

$$f \cong_{\mathcal{D}_X} f[e/Y]$$

Démonstration : *Soit ρ un environnement sémantiquement correct : ρ est bien défini, et ρ est un plus petit point fixe de la sémantique de$S$. En appliquant récursivement la définition de la sémantique des expressions, on est amené à prouver*

$$\rho(Y) = \rho(e)$$

Comme ρ est un plus petit point fixe de la sémantique de, on a

$$\rho = \mathcal{Q}(Y = e)(\rho)$$

par définition de la sémantique des équations on obtient alors

$$\begin{aligned} \rho(Y) \quad = \quad & (dom(\rho(Y)) \cup dom(\mathcal{E}(e)(\rho)), \\ & Supval(val(\rho(Y)), val(\mathcal{E}(e)(\rho)))) \end{aligned}$$

Nous allons décomposer la preuve en deux parties : tout d'abord nous prouverons l'égalité des domaines, puis nous regarderons les valeurs de Y et de e sous l'environnement ρ.

Domaine *Nous voulons montrer que*

$$dom(\rho(Y)) = dom(\mathcal{E}(e)(\rho))$$

D'après la définition de la sémantique des équations on sait que

$$dom(\rho(Y)) = dom(\rho(Y)) \cup dom(\mathcal{E}(e)(\rho))$$

- *Soit $dom(\rho(Y)) = dom(\mathcal{E}(e)(\rho))$, et l'égalité est prouvée*
- *Soit $dom(\mathcal{E}(e)(\rho)) \subset dom(\rho(Y))$, à partir de l'environnement ρ on peut construire un environnement ρ' qui est un point fixe plus petit que ρ tel que pour tout point z appartenant à $dom(\rho(Y)) \setminus dom(\mathcal{E}(e)(\rho))$, $val(\rho(Y))[z] = \perp$, et pour tout z appartenant à $dom(\mathcal{E}(e)(\rho))$, $val(\rho(Y))[z] = Supval(val(\rho(Y)), val(\mathcal{E}(e)(\rho)))$. Comme ρ est un plus petit point fixe, $\rho = \rho'$, or ρ' n'est pas bien défini. On a donc l'égalité sur les domaines.*

Valeur *On veut maintenant montrer que*

$$val(\rho(Y)) = val(\mathcal{E}(e)(\rho))$$

Comme ρ est bien défini, on sait que pour toute variable id et en tout point z de leur domaine

$$val(\mathcal{E}(id)(\rho) \neq \bot$$

Ainsi, en appliquant récursivement la sémantique des expressions sur e, on en déduit que

$$val(\mathcal{E}(e)(\rho)) \neq \bot$$

Comme $val(\mathcal{E}(e)(\rho))$ est défini, $val(\rho(Y))$ est différent de $\bot$.

- *Soit $val(\rho(Y)) = val(\mathcal{E}(e)(\rho))$,*
- *soit $val(\rho(Y)) \neq val(\mathcal{E}(e)(\rho))$, alors on en déduit que $val(\rho(Y)) = \top$. Cependant, si $val(\rho(Y)) = \top$, alors ρ n'est plus un petit point fixe, car on peut construire un environnement ρ' sémantiquement correct tel que $val(\rho'(Y)) \neq val(\mathcal{E}(e)(\rho'))$. On a donc l'égalité sur les valeurs.*

On a ainsi montré l'équivalence suivante :

$$f(Y) \cong f(e)$$

□

2 Itérations

Le but de ce théorème est de caractériser les cas où l'itération substitution par constante recherche d'auto-dépendance est finie lorsque la variable pour laquelle la recherche d'auto-dépendance est effectuée contient des négations.

2.1 Le cas de la négation

Théorème 5.4 *Soit W et X deux variables polyédriques. Supposons que nous voulons prouver que pour tout environnement ρ, $\rho \models W : v$ où v est la valeur absorbante de $\Cap_1$. Alors, si W et X sont définies par les équations suivantes, où $l_1, l_2, l \in \mathbb{N}^*$ et $\forall i \in \{1, \ldots, l\}, m_i \in \mathbb{N}^*$:*

$$\begin{aligned} W &= (\bigCap_{j=1}^{l_1}{}_1 X.d_j) \Cap_1 (\bigCap_{j=1}^{l_2}{}_1 \neg X.d'_j) \\ X &= \bigCap_{i=1}^{l}{}_2 \bigCap_{j=1}^{m_i}{}_1 X.d^i_j \end{aligned}$$

Nous avons,

- *itérer alternativement la recherche d'auto-dépendance et la substitution par constante est fini si*
  - *$l_1 = l_2 = 1$,*
  - *ou si $l_1 = 1 \wedge l_2 > 1 \wedge \forall i,\ m_i = 1$,*
  - *ou enfin si $l_1 > 1 \wedge l_2 = 1 \wedge l = 1$*
- *itérer alternativement la recherche d'auto-dépendance et la substitution par constante est infini si $l_1 > 1$ et $l_2 > 1$.*

Nous ne démontrerons ici que le cas où $l_1 = l_2 = 1$, les autres cas se montrent de manière similaire.

Démonstration : *Nous supposons la variable W définie par l'équation suivante :*

$$W = X.d_1 \barwedge_1 \neg X.d_2$$

Après substitution de X par sa définition dans W, nous obtenons :

$$W = (\bigbarwedge_{i=1}^{l}{}_2 \bigbarwedge_{j=1}^{m_i}{}_1 X.d_j^i \circ d_1) \barwedge_1 \neg(\bigbarwedge_{i'=1}^{l}{}_2 \bigbarwedge_{j'=1}^{m_{i'}}{}_1 X.d_{j'}^{i'} \circ d_2)$$

Nous distribuons la négation pour obtenir :

$$W = (\bigbarwedge_{i=1}^{l}{}_2 \bigbarwedge_{j=1}^{m_i}{}_1 X.d_j^i \circ d_1) \barwedge_1 (\bigbarwedge_{i'=1}^{l}{}_1 \bigbarwedge_{j'=1}^{m_{i'}}{}_2 \neg X.d_{j'}^{i'} \circ d_2)$$

On distribue l'opérateur $\barwedge_1$ dans le premier terme.

$$W = \bigbarwedge_{i=1}^{l}{}_2((\bigbarwedge_{j=1}^{m_i}{}_1 X.d_j^i.d_1) \barwedge_1 (\bigbarwedge_{i'=1}^{l}{}_1 \bigbarwedge_{j'=1}^{m_{i'}}{}_2 \neg X.d_{j'}^{i'}.d_2))$$

Nous notons W_i la $\barwedge_1$-expression telle que $W = \barwedge_{2\,i=1}^{l} W_i$. On sépare l'expression indexée par i' en deux selon si $i = i'$ ou non. On obtient alors :

$$W_i = (\bigbarwedge_{j=1}^{m_i}{}_1 X.d_j^i.d_1) \barwedge_1 (\bigbarwedge_{j'=1}^{m_i}{}_2 \neg X.d_{j'}^i \circ d_2) \barwedge_1 (\bigbarwedge_{(i'=1;i\neq i')}^{l}{}_1 \bigbarwedge_{j'=1}^{m_{i'}}{}_2 \neg X.d_{j'}^{i'} \circ d_2)$$

Nous notons W_i^1 et W_i^2 les deux expressions suivantes :

$$\begin{aligned} W_i^1 &= (\bigbarwedge_{j=1}^{m_i}{}_1 X.d_j^i \circ d_1) \barwedge_1 (\bigbarwedge_{j'=1}^{m_i}{}_2 \neg X.d_{j'}^i \circ d_2) \\ W_i^2 &= \bigbarwedge_{(i'=1;i\neq i')}^{l}{}_1 \bigbarwedge_{j'=1}^{m_{i'}}{}_2 \neg X.d_{j'}^{i'} \circ d_2 \end{aligned}$$

Ainsi, $W_i = W_i^1 \barwedge_1 W_i^2$. Nous travaillons maintenant sur W_i^1. Après distribution de $\barwedge_1$ dans le second terme de W_i^1, nous obtenons :

$$W_i^1 = \bigbarwedge_{j'=1}^{m_i}{}_2(\neg X.d_{j'}^i \circ d_2 \barwedge_1 (\bigbarwedge_{j=1}^{m_i}{}_1 X.d_j^i \circ d_1))$$

Nous séparons maintenant en deux l'expression indexée par j selon si $j = j'$ ou non. On obtient alors :

$$W_i^1 = \bigbarwedge_{j'=1}^{m_i}{}_2(\neg X.d_{j'}^i \circ d_2 \barwedge_1 X.d_{j'}^i \circ d_1 \barwedge_1 (\bigbarwedge_{(j=1;j\neq j')}^{m_i}{}_1 X.d_j^i \circ d_1))$$

On applique alors la recherche d'auto-dépendance pour obtenir :

$$W_i^1 = \bigbarwedge_{j'=1}^{m_i}{}_2 W.d_{j'}^i \barwedge_1 (\bigbarwedge_{(j=1;j\neq j')}^{m_i}{}_1 X.d_j^i \circ d_1))$$

En appliquant la substitution par valeur dans W_i^1, et comme v est absorbante pour $\barwedge_1$ nous obtenons, pour tout i :

$$W_i^1 = \bigbarwedge_{j'=1}^{m_i}{}_2 v \barwedge_1 (\bigbarwedge_{(j=1;j\neq j')}^{m_i}{}_1 X.d_j^i \circ d_1) = \bigbarwedge_{j'=1}^{m_i}{}_2 v = v$$

Nous substituons maintenant W_i^1 par sa valeur dans W_i :

$$W_i = v \bowtie_1 W_i^2 = v$$

Il reste maintenant à substituer tous les W_i^1 par leur valeur dans W :

$$W = (\bowtie_{i=1}^{l}{}_2\, v) = v$$

Nous obtenons ainsi une constante et l'itération s'arrête. □

2.2 Cas des branches multiples

Cette proposition est utilisée pour déterminer les cas où l'itération de la recherche d'auto-dépendance et de la substitution pa constante est finie lorsque la variable pour laquelle la recherche d'auto-dépendance est effectuée est définie par plusieurs branche. Nous avons pour cela plonger le problème dans l'ensemble des rationnels et projeter les dépendances et les domaines sur un vecteur $\vec{\omega}$ orthogonal à l'hyperplan séparant les domaines des deux branches de la variable étudiée.

Proposition 5.6

$$\begin{aligned} & \alpha_2 - \alpha_1 > 0 \wedge \beta_1 \leq 0 \wedge \beta_2 \geq 0 \\ \Longleftrightarrow \quad & \bigcap_{i=1}^{q} d_1^{-1}(\delta_1^{-1})^i(P_1) \cap d_2^{-1}(\delta_2^{-1})^i(P_2) \neq \emptyset, \forall q \in \mathbb{N} \end{aligned}$$

Démonstration : *Soit A_i l'intersection $d_1^{-1} \circ (\delta_1^{-1})^i(P_1) \cap d_2^{-1} \circ (\delta_2^{-1})^i(P_2)$.*

$$\forall q \in \mathbb{N}, \bigcap_{i=1}^{q} A_i \neq \emptyset \tag{5.1}$$

est équivalent à

$$\forall q \in \mathbb{N}, \exists x, \forall 0 \leq j, j' \leq q,\ x \in A_j \cap A_{j'}$$

Soit x un point tel que

$$\forall 0 \leq j, j' \leq q, x \in A_j \cap A_{j'}$$

Ainsi, il existe $y_{1,j}, y'_{1,j'}$ dans P_1 tel que

$$x = d_1^{-1} \circ (\delta_1^{-1})^j(y_{1,j}) = d_1^{-1} \circ (\delta_1^{-1})^{j'}(y'_{1,j'})$$

et $y_{2,j}, y'_{2,j'}$ dans P_2, tel que

$$x = d_2^{-1} \circ (\delta_2^{-1})^j(y_{2,j}) = d_2^{-1} \circ (\delta_2^{-1})^{j'}(y'_{2,j'})$$

Notons x_ω la projection de x sur le vecteur $\vec{\omega}$. Puisque $y_{1,j}, y'_{1,j'}$ sont dans P_1 et $y_{2,j}, y'_{2,j'}$ dans P_2, il existe $\gamma_{1,j}, \gamma'_{1,j'} > 0$ et $\gamma_{2,j}, \gamma'_{2,j'} < 0$, où $\gamma_{1,j}$ (respectivement $\gamma'_{1,j'},\ \gamma_{2,j},\ \gamma'_{2,j'}$) est la projection sur le vecteur $\vec{\omega}$ de $y_{1,j}$ (respectivement $y_{1,j'},\ y_{2,j},\ y'_{2,j'}$) tels que

$$\begin{aligned} x_\omega &= (\gamma_{1,j} - \alpha_1 - j\beta_1)\vec{\omega} = (\gamma_{2,j} - \alpha_2 - j\beta_2)\vec{\omega} \\ &= (\gamma'_{1,j'} - \alpha_1 - j'\beta_1)\vec{\omega} = (\gamma'_{2,j'} - \alpha_2 - j'\beta_2)\vec{\omega} \end{aligned} \tag{5.2}$$

En utilisant l'équation (5.2), l'expression (5.1) peut ainsi se réécrire de la manière suivante :

$$\forall q \in \mathbb{N},\ \forall 0 \leq j, j' \leq q\ \exists \gamma_{1,j}, \gamma'_{1,j'} > 0\ \gamma_{2,j}, \gamma'_{2,j'} < 0$$

$$\begin{aligned}
\gamma_{1,j} - \alpha_1 - j\beta_1 &= \gamma_{2,j} - \alpha_2 - j\beta_2 && (5.3)\\
\gamma_{1,j} - \alpha_1 - j\beta_1 &= \gamma'_{2,j'} - \alpha_2 - j'\beta_2 && (5.4)\\
\gamma'_{1,j'} - \alpha_1 - j'\beta_1 &= \gamma_{2,j} - \alpha_2 - j\beta_2 && (5.5)
\end{aligned}$$

Puisque l'égalité(5.3) est vraie pour tout $0 \leq j \leq q$, elle est vraie pour $j = 0$. Il s'ensuit que $\gamma_{2,0} - \gamma_{1,0} = \alpha_2 - \alpha_1$. Ainsi, nous obtenons $\exists \gamma_{1,0} < 0, \gamma_{2,0} > 0, \gamma_{2,0} - \gamma_{1,0} = \alpha_2 - \alpha_1$ ce qui est équivalent à $\alpha_2 > \alpha_1$.

Puisque $\gamma'_{2,j'} > 0$ et $\gamma_{1,j} < 0$, nous pouvons transformer l'égalité (5.4) dans l'inégalité suivante :

$$\alpha_2 - \alpha_1 + j'\beta_2 - j\beta_1 > 0$$

Si $j' = 0$ et $j = q$, nous avons

$$\alpha_2 - \alpha_1 > q\beta_1$$

Ainsi $\frac{\alpha_2 - \alpha_1}{q} \geq \beta_1$. Puisque l'inégalité doit être vraie pour tout $q \in \mathbb{N}$, cela implique que $\beta_1 \leq 0$. De manière similaire, en utilisant l'équation (5.5), nous obtenons $\beta_2 \geq 0$. Inversement, si $\alpha_2 - \alpha_1 > 0 \wedge \beta_2 - \beta_1 \geq 0 \wedge \beta_1 \leq 0 \wedge \beta_2 \geq 0$, il existe $\gamma_{1,j}, \gamma'_{1,j'} < 0$, $\gamma_{2,j}, \gamma'_{2,j'} > 0$, tels que les égalités (5.3), (5.4), et (5.5) soient vraies. ❑

2.3 Motifs définis par plusieurs variables

Le but est encore d'étudier les cas où l'itération de la recherche d'auto-dépendance et de la substitution par constante sont finies, en supposant cette fois-ci la variable W, variable pour laquelle la recherche d'auto-dépendance est effectuée, définie à l'aide de plusieurs variables.

Nous supposons ici W définie par :

$$W = X.d_1 \between_1 Y.d_2$$

Théorème 5.9 *Nous allons ici donner quelques cas où la recherche d'auto-dépendance réussit :*

- *Si les variables X et Y sont définies par les équations suivantes :*

$$\begin{aligned}
X &= X.\delta_1 \between_1 \dots\\
Y &= Y.\delta_2 \between_1 \dots
\end{aligned}$$

  alors, la recherche d'auto-dépendance réussira si et seulement si les dépendances δ_1 et δ_2 sont parallèles.
- *Si les variables X et Y sont définies par les équations suivantes :*

$$\begin{aligned}
X &= Y.\delta_1 \between_1 \dots\\
Y &= X.\delta_2 \between_1 \dots
\end{aligned}$$

  alors, la recherche d'auto-dépendance réussira.
- *Si maintenant la variable X est définie par l'équation suivante :*

$$X = Y.\delta_2 \between_1 X.\delta_1 \dots$$

  alors la recherche d'auto-dépendance réussira si et seulement si

$$\delta_2 = d_2 \circ \delta_1 \circ d_1^{-1}$$

Démonstration :

– *Nous commençons par traiter le deuxième cas, à savoir X et Y définies par les équations suivantes :*

$$\begin{aligned} X &= Y.\delta_1 \between_1 \dots \\ Y &= X.\delta_2 \between_1 \dots \end{aligned}$$

Nous substituons X et Y par leurs définitions dans la variable W, et nous effectuons la recherche d'auto-dépendance. Nous ne pouvons écrire qu'un seul système qui correspond au système 5.4 du lemme 5.2. Ainsi nous avons une solution à ce système si et seulement si

$$\delta_1/2 + d_1 = \delta_2/2 + d_2$$

Si nous n'avons pas l'égalité, il suffit alors d'effectuer une étape de substitution supplémentaire et nous obtenons alors le système suivant :

$$\begin{aligned} \delta_1 \circ \delta_2 \circ d_1 &= d_1 \circ f \\ \delta_1 \circ \delta_2 \circ d_2 &= d_2 \circ f \end{aligned}$$

Nous reconnaissons le système 5.1 du lemme 5.2 et nous trouvons une solution.

– *Si maintenant nous regardons le premier cas, à savoir X et Y définies par les équations suivantes :*

$$\begin{aligned} X &= X.\delta_1 \between_1 \dots \\ Y &= Y.\delta_2 \between_1 \dots \end{aligned}$$

L'idée est de substituer X et Y un certain nombre de fois pour obtenir le système 5.1 du lemme 5.2. Il suffit de trouver deux entiers u et v tels que $\delta_1^u = \delta_2^v$. On substitue alors X u fois par sa définition et Y v fois par sa définition. On obtient alors le système suivant :

$$\begin{aligned} \delta_1^u \circ d_1 &= d_1 \circ f \\ \delta_2^v \circ d_2 &= d_2 \circ f \end{aligned}$$

et nous avons une solution.

– *Nous traitons maintenant le dernier cas, il suffit simplement de substituer X par sa définition et nous obtenons lors de la recherche d'auto-dépendance le système*

$$\begin{aligned} \delta_1 \circ d_1 &= d_1 \circ f \\ \delta_2 \circ d_2 &= d_2 \circ f \end{aligned}$$

En le résolvant nous trouvons qu'il y a une solution si et seulement si

$$\delta_2 = d_2 \circ \delta_1 \circ d_1^{-1}$$

□

Annexe B

Quelques rappels d'algèbre

Définition B.1 (Groupe) *Un groupe G est un ensemble muni d'une opération (loi de composition interne), notée ici $+$, qui à tout couple (x, y) de G associe un élément, noté $x + y$, de G pour laquelle les axiomes suivants sont vérifiés :*

- *la loi $+$ est associative : $(x + y) + z$ = $x + (y + z)$ pour tout x, y et z de G,*
- *la loi $+$ possède un élément neutre n : $x + n = n + x = x$*
- *pour tout x de G tout élément x de G possède un symétrique x' pour la loi $+$: $x+x' = x'+x = n$*

Un groupe est dit *abélien* ou *commutatif* si sa loi de composition est commutative : $x + y = y + x$ pour tout x et tout y de G.

Définition B.2 (Anneau) *On appelle* anneau *un groupe abélien A muni d'une seconde loi $*$, appelée multiplication, associative et distributive sur la loi de groupe (addition), Pour tout triplet (a, b, c) d'éléments de A :*

- *associativité : $(a * b) * c = a * (b * c)$*
- *distributivité $a * (b + c) = a * b + a * c$ et $(a + b) * c = a * c + b * c$.*

Un anneau est dit *commutatif* si la multiplication l'est. L'élément neutre du groupe est appelé *zéro*. Una nneau est dit *unitaire* si la multiplication admet un élément neutre, on appelle cet élément l'élément *unité*.

Définition B.3 (Corps) *Un anneau dans lequel tout élément non nul admet un symétrique pour la multiplication est un* corps.

Définition B.4 (Espace Vectoriel) *Soit $\mathbb{K}$ un corps commutatif d'élément unité noté 1. On nomme* espace vectoriel *sur $\mathbb{K}$, un ensemble E muni d'une loi de composition interne $(+)$ conférant à E la structure de groupe abélien, et d'une seconde loi dite externe, application de $E \times \mathbb{K}$ dans E notée ici par un simple point $\cdot$ faisant intervenir les éléments de $\mathbb{K}$, appelés scalaires. Cette loi externe est souvent appelée multiplication scalaire et doit vérifier les axiomes suivants, x et y désignant des éléments (vecteurs) de E, a et b désignant des scalaires*

- $a \cdot (x + y) = a \cdot x + a \cdot y$
- $(a + b) \cdot x = a \cdot x + b \cdot x$
- $a \cdot (b \cdot x) = ab \cdot x$
- $1 \cdot x = x$

Si $\mathbb{K}$ est un anneau commutatif unitaire, alors E sera un *module*.

Définition B.5 (Algèbre) *Considérons un corps $\mathbb{K}$. Une* algèbre *sur $\mathbb{K}$ est un espace vectoriel sur $\mathbb{K}$ muni, en outre, d'une seconde loi interne (multiplication) distributive par rapport à la loi de groupe*

(addition de la structure d'espace vectoriel). Si cette multiplication est associative (resp. commutative) l'algèbre est dite associative (resp. commutative).

Annexe C

Résolution d'équations sous forme matricielle dans $\mathbb{Z}$

Dans cette partie, nous allons voir comment résoudre une équation de la forme $Ax = b$, où A est une matrice de $\mathbb{Z}^{n\times m}$. Nous allons d'abord déterminer si le système admet une solution, et le cas échéant l'exhiber. Puis, si le système admet une solution, nous les trouverons toutes, en calculant une base du noyau de A. Le principe est d'échelonner la matrice A, c'est-à-dire de la transformer en matrice triangulaire inférieure. Les résultats que nous exposons ici sont démontrés dans [71].

Définition C.1 (Équivalence à droite) *Soient A et A' deux matrices à coefficients entiers. Elles sont dites* équivalentes à droite *si et seulement si il existe une matrice inversible R, vérifiant $A' = AR$. Cette propriété définit bien sûr une relation d'équivalence.*

Les matrices A et A' ont ainsi la même image. Nous allons exhiber une classe de matrices dont l'image est facile à calculer : c'est le cas des matrices échelonnées.

Théorème C.2 *Il existe un algorithme qui, appliqué à une matrice A de dimension $\mathbb{Z}^{n\times m}$, fournit une transformation à droite réversible qui échelonne A, donnant une nouvelle matrice A'. Plus précisément, cet algorithme, appliqué à A fournit un couple (A', R) où A' est une matrice $n \times m$ échelonnée, et R une matrice $m \times m$ de déterminant 1 dont l'inverse est à coefficients entiers. Ce couple est tel que $A' = AR$.*

Exhiber une solution à l'équation revient à savoir écrire explicitement le vecteur b comme combinaison linéaire des colonnes de A. En considérant la matrice obtenue en bordant à droite la matrice A par le vecteur b, nous étudierons le problème suivant :
Étant donné une matrice C, à n lignes et $m + 1$ colonnes, déterminer si la dernière colonne C_{m+1} est combinaison linéaire des colonnes précédentes $C_1, C_2, \ldots, C_m$, et si oui expliciter une telle combinaison linéaire.

Théorème C.3 *Soit C une matrice $n \times (m + 1)$ à coefficients entiers. Désignons par C' et R les deux matrices fournies par l'algorithme d'échelonnement en colonnes (R est la matrice carrée inversible et C' la matrice échelonnée telles que $C' = CR$). Si R égale $(r_{i\ j})$, les deux propriétés sont alors équivalentes :*

- *la dernière colonne C_{m+1} est combinaison linéaire des colonnes $C_1, C_2, \ldots, C_m$,*
- *la colonne C'_{m+1} est nulle et $r_{m\ m}$ est inversible.*

La dernière ligne de R est alors $(0, 0, \ldots, 0, 0, 1)$ et on a la relation :

$$C_{m+1} = -(r_{1\ m+1}C_1 + r_{2\ m+1}C_2 + \cdots + r_{m\ m+1}C_m)$$

Il ne reste plus qu'à calculer le noyau de la matrice A.

Proposition C.4 *Soit A une matrice $n \times m$ à coefficients entiers, et R et A' les deux matrices fournies par l'algorithme d'échelonnement en colonnes. Si $A'_{k+1}, A'_{k+2}, \ldots, A'_m$ sont les colonnes nulles de A', alors les dernières colonnes $R_{k+1}, \ldots, R_m$ forment une base du noyau de A,*

Tous ces résultats permettent de résoudre le système étudié. Pour résumer, nous énonçons la proposition suivante :

Proposition C.5 *Désignons par $(A \mid b)$ la matrice constituée de la matrice A bordée à droite par le vecteur b. Alors $(A \mid b)$ est l'application linéaire de $\mathbb{Z}^m \times \mathbb{Z}$ dans $\mathbb{Z}^n$, qui à (x, x_{m+1}) associe $Ax + bx_{m+1}$.*

Soit $(A' \mid b')$ la matrice échelonnée obtenue par l'algorithme d'échelonnement à partir de la matrice $(A \mid b)$, et R la matrice de passage vérifiant $(A \mid b)R = (A' \mid b')$.

- *Le système $Ax = b$ admet une solution si et seulement si la dernière ligne de R est $(0, 0, \ldots, 0, 0, 1)$ et si le vecteur b' est nul. Ces deux conditions fournissent alors la solution particulière $-\bar{R}_{m+1}$, où $-\bar{R}_{m+1}$ est le vecteur formé par les m premières composantes de R_{m+1}.*
- *Dans ce cas, A' est une matrice échelonnée équivalente à gauche à A. Soit $k \in [1, m]$ tel que les colonnes nulles de A soient exactement celles d'indice supérieur à k ; alors les vecteurs $\bar{R}_{k+1}, \bar{R}_{k+2}, \ldots, \bar{R}_{m+1}$ forment une base du noyau de A.*

Le problème que nous voulons résoudre est le suivant : nous cherchons la matrice X de $\mathbb{Z}^{m \times p}$ telle que $AX = B$ où A est une matrice de $\mathbb{Z}^{n \times m}$, et B une matrice de $\mathbb{Z}^{n \times p}$.

Proposition C.6 *La matrice X solution d'un tel système s'écrit sous la forme $X = \xi + \Gamma.\beta$ où ξ est une solution particulière, les colonnes de Γ forment une base du noyau de A, et β est une matrice quelconque de $\mathbb{Z}^{k \times p}$ où k est la dimension du noyau de A.*

Démonstration : *Soient $x_1, \ldots, x_p$ les lignes de X et $b_1, \ldots, b_p$ les colonnes de B. Pour tout i, nous savons résoudre le système $A.x_i = b_i$, et nous obtenons*

$$x_i = \xi_i + \sum_j \beta_{i,j}\gamma_j$$

où ξ_i est la solution particulière et γ_j un vecteur du noyau de A. Notons k la dimension du noyau de A et Γ la matrice de $\mathbb{Z}^{m \times k}$ dont les colonnes sont les vecteurs γ_j. Si β_i est le vecteur de dimension k, alors on peut écrire x_i de la manière suivante :

$$x_i = \xi_i + \Gamma.\beta_i$$

On note ξ la matrice de $\mathbb{Z}^{n \times m}$ dont les lignes sont les vecteurs ξ_i, et β la matrice dont les colonnes sont les vecteurs β_i. Alors la matrice X vaut :

$$X = \xi + \Gamma.\beta$$

□

Le deuxième problème que nous rencontrons est comment calculer l'intersection de deux ensembles $S_1 = \{\xi_1 + \Gamma_1.\beta_1 \mid \beta_1 \in \mathbb{Z}^{k \times p}\}$ et $S_2 = \{\xi_2 + \Gamma_2.\beta_2 \mid \beta_2 \in \mathbb{Z}^{k \times p}\}$. On cherche donc à calculer β_1 et β_2 tels que

$$\xi_1 + \Gamma_1.\beta_1 = \xi_2 + \Gamma_2.\beta_2$$

En écrivant l'équation différemment, on cherche à résoudre l'équation :

$$\Gamma_1.\beta_1 - \Gamma_2.\beta_2 = \xi_2 - \xi_1$$

Si on pose $(\Gamma_1 \mid -\Gamma_2)$ la matrice Γ_1 bordée à droite par la matrice $-\Gamma_2$, et si on pose $\left(\frac{\beta_1}{\beta_2}\right)$ la matrice β_1 bordée en dessous par la matrice β_2, on veut résoudre :

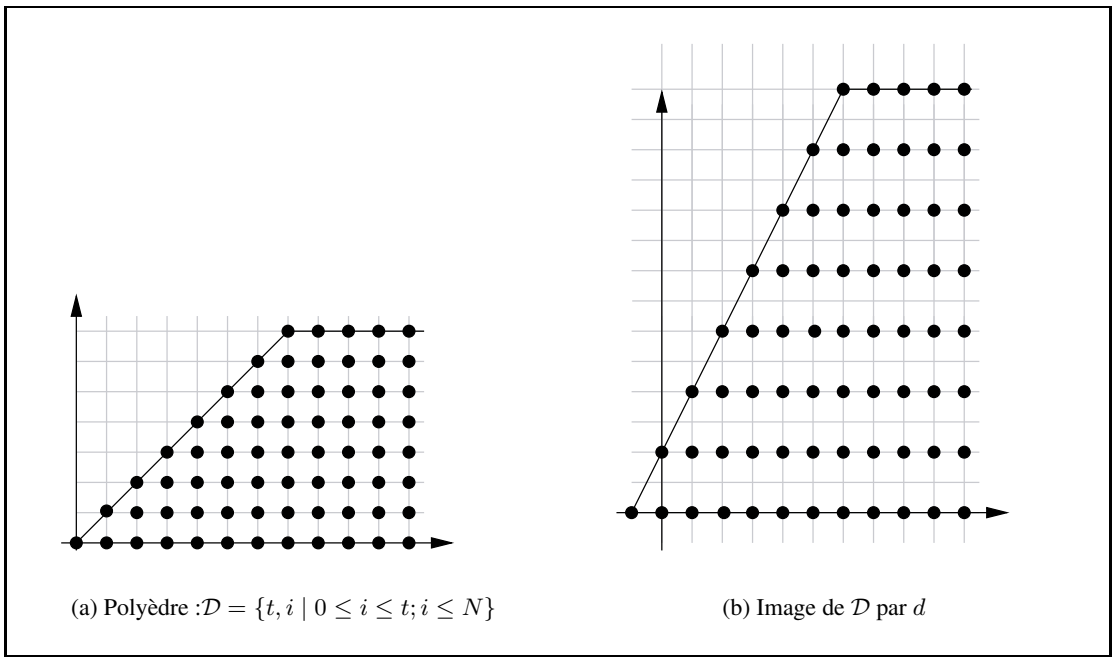
$$(\Gamma_1 \mid -\Gamma_2).\left(\frac{\beta_1}{\beta_2}\right) = \xi_2 - \xi_1$$

Pour résoudre un système, il suffit d'appliquer la proposition C.6.

Annexe D

$\mathbb{Z}$-polyèdres

Nous rappelons que l'image d'un polyèdre par une dépendance quelconque n'est pas un polyèdre mais une union infinie de polyèdres. Pour illustrer, prenons l'exemple suivant. Soit $\mathcal{D}$ le domaine $\{t, i \mid 0 \leq i \leq t; 0 \leq i \leq N\}$ où N est un paramètre, et prenons la dépendance d suivante $(t, i \rightarrow t - 1, 2i)$. Cette dépendance est affine. Nous avons représenté en figure D.1 le polyèdre $\mathcal{D}$ et son image d. Clairement, l'image n'est pas un polyèdre, mais une union infinie de polyèdres. Cependant, nous remarquons que cette union est « régulière », c'est l'ensemble des points (t, i) appartenant à $\{t, 2i \mid 0 \leq i \leq t + 1; i \leq N\}$. On retrouve la structure d'un polyèdre, les contraintes sont linéaires, mais au lieu de contenir les points (t, i) respectant les contraintes, l'ensemble contient les points $(t, 2i)$ où (t, i) respecte les contraintes. C'est ce que l'on appelle un $\mathbb{Z}$-polyèdre [85].



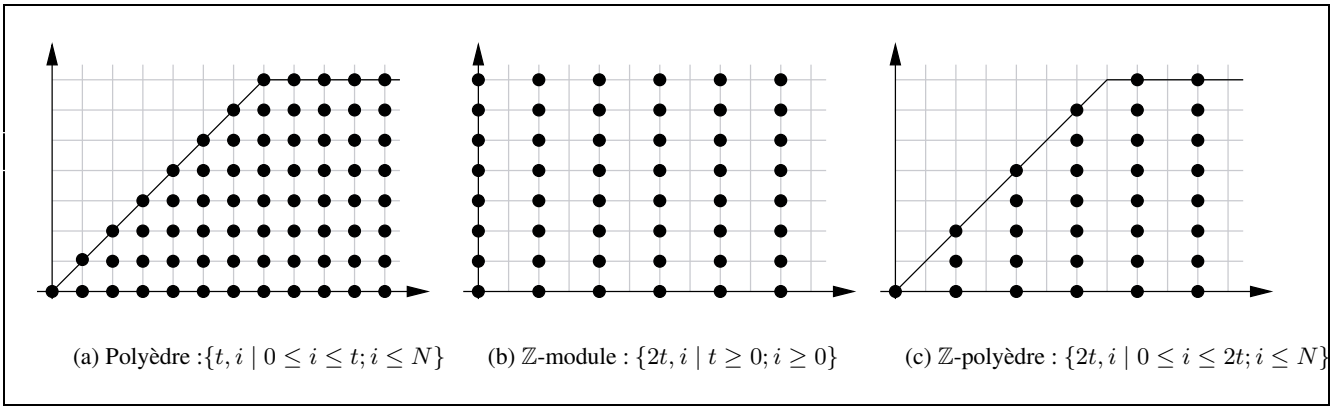
(a) Polyèdre :$\mathcal{D} = \{t, i \mid 0 \leq i \leq t; i \leq N\}$ (b) Image de $\mathcal{D}$ par d

FIG. D.1 – Polyèdre et son image par une dépendance uniforme

Donnons maintenant une définition formelle de cette notion.

Définition D.1 ($\mathbb{Z}$-Polyèdre) *Un $\mathbb{Z}$-Polyèdre est l'intersection d'un polyèdre et d'un $\mathbb{Z}$-module (cf annexe B).*

La librairie polyédrique a été étendue aux $\mathbb{Z}$-polyèdres. Ainsi, nous pouvons calculer les images et préimages de $\mathbb{Z}$ polyèdres par des dépendances, affines, calculer l'union, l'intersection, la différences



(a) Polyèdre : $\{t, i \mid 0 \leq i \leq t; i \leq N\}$ (b) $\mathbb{Z}$-module : $\{2t, i \mid t \geq 0; i \geq 0\}$ (c) $\mathbb{Z}$-polyèdre : $\{2t, i \mid 0 \leq i \leq 2t; i \leq N\}$

FIG. D.2 – Polyèdre, $\mathbb{Z}$-module, $\mathbb{Z}$-polyèdre

de deux $\mathbb{Z}$-polyèdres. Les différentes règles de preuve que nous avons implémentés ne s'appliquent actuellement qu'à des polyèdres, mais leur extension aux $\mathbb{Z}$-polyèdres ne devrait pas poser de problème.

Annexe E

Liste des fonctions implémentant les règles de preuve

- **substitution_constante :** Cette fonction implémente la règle de substitution par constante. Elle se définit de la manière suivante :

$$\textbf{substitution\_constante} : (\langle X, \mathcal{D}, v\rangle, S) \longrightarrow (\langle X', \mathcal{D}, v\rangle, S')$$

où X' est la variable définie par l'expression de X après substitution et où S' est le système S contenant la variable X' et l'équation définissant X'. S'il existe une preuve pour la formule étendue $\mathcal{D} : X' \downarrow v$ dans le système S', alors il existe une preuve de $\mathcal{D} : X \downarrow v$.

- **équation :** Cette fonction correspond à la règle de l'équation.

$$\textbf{équation} : (\langle X, \mathcal{D}, v\rangle, S) \longrightarrow (\langle e, \mathcal{D}, v\rangle, S)$$

où e est l'expression définissant X.

- **simplification :** Cette fonction implémente la règle de simplification.

$$\textbf{simplification} : (\langle e, \mathcal{D}, v\rangle, S) \longrightarrow (\langle e', \mathcal{D}, v\rangle, S)$$

où e' correspond à la simplification de l'expression e.
Plutôt que de réécrire une fonction de simplification pour les expressions polyédriques, nous avons préféré utiliser la simplification offerte par Mathematica. Pour cela, nous commençons par traduire l'expression polyédrique représentée par un arbre syntaxique en une expression booléenne Mathematica. Puis, sur cette expression booléenne Mathematica, nous utilisons la simplification propositionnelle de Mathematica. Enfin, nous retraduisons le résultat en une expression ALPHA.

- **séparation_branches :** Cette fonction implémente la règle de séparation des branches. Cette séparation génère plusieurs fils.

$$\textbf{séparation\_branches}(\langle e, \mathcal{D}, v\rangle, S) \longrightarrow ([\langle e_1, \mathcal{D}_1 \cap \mathcal{D}, v\rangle, \ldots, \langle e_n, \mathcal{D}_n \cap \mathcal{D}, v\rangle], S)$$

où e est l'expression définie par :

$$e = \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \\ \mathcal{D}_n : e_n \end{cases}$$

- **ajout_variable :** Cette fonction implémente la règle d'ajout de variable.

$$\textbf{ajout\_variable} : (\langle e, \mathcal{D}, v\rangle, S) \longrightarrow (\langle X', \mathcal{D}, v\rangle, S')$$

Le système S' correspond au système S auquel a été ajoutée la variable X' définie par e et son équation. L'expression e est une sous-expression de la variable X.

– **recherche_auto_dépendance :** Cette fonction implémente la règle de recherche d'auto-dépendance. Pour cette recherche, il faut prendre quelques précautions avant d'appliquer la fonction et s'assurer que la recherche d'auto-dépendance itérée avec une substitution par constante ne va pas engendrer une construction infinie. Deux cas peuvent se produire :
  – soit la recherche d'auto-dépendance peut engendrer une construction infinie et dans ce cas nous n'appliquons pas la fonction,
  – soit la recherche d'auto-dépendance n'engendre pas de comportement infini et la construction continue.

  Si la fonction est appliquée, le système de sortie est différent du système d'entrée.

$$\textbf{recherche\_auto\_dépendance} : (\langle X, \mathcal{D}, v\rangle, S) \longrightarrow (\langle X', \mathcal{D}, v\rangle, S')$$

  où la variable X' est sémantiquement équivalent à X, mais est définie à l'aide d'auto-dépendances. Le système S' correspond au système S et contient de plus la variable X' et l'équation définissant cette variable.
– **séparation_opérandes :** Cette fonction implémente la règle de séparation des opérandes. Si l'expression e est définie par une $\bowtie$-expression ($e_1 \bowtie e_2$) et que v est neutre pour $\bowtie$ nous générons une liste de deux fils correspondant aux expressions e_1 et e_2. Le système de sortie est identique au système d'entrée.

$$\textbf{séparation\_opérande.} : (\langle e_1 \bowtie e_2, \mathcal{D}, v\rangle, S) \longrightarrow ([\langle e_1, \mathcal{D}, v\rangle, \langle e_2, \mathcal{D}, v\rangle], S)$$

– **négation :** Cette fonction correspond à implémente la règle de négation.

$$\textbf{négation} : (\langle \neg e, \mathcal{D}, v\rangle, S) \longrightarrow (\langle e, \mathcal{D}, \neg v\rangle, S)$$

– **dépendance :** Cette fonction implémente la règle de la dépendance.

$$\textbf{dépendance} : (\langle Y.d, \mathcal{D}, v\rangle, S) \longrightarrow (\langle Y, d^{-1}(\mathcal{D}), v\rangle, S)$$

– **forme_normale :** Cette fonction implémente la règle de mise sous forme normale. Cette fonction est utilisée avant la fonction de séparation des opérandes, elle transforme l'expression dans la forme normale telle que v soit neutre.

$$\textbf{forme\_normale} : (\langle e, \mathcal{D}, v\rangle, S) \longrightarrow (\langle e', \mathcal{D}, v\rangle, S)$$

Annexe F

Systèmes de règles de preuve

1 Systèmes de règles de preuve des formules

Dans le chapitre 4, lorsque nous avons donné notre premier système de règles de preuve, nous avons précisé que ce système n'était pas complet : les règles que nous avons proposées ne s'appliquaient qu'à la partie droite du séquent. Le but était simplement de donner le principe de ces différentes règles. Nous allons donc maintenant compléter notre système en donnant le résultat des règles de preuve qui s'appliquent au contexte. Comme toutes nos règles de preuve sont inversibles, nous allons pouvoir les utiliser sur le contexte. Voici donc les règles de preuve s'appliquant au contexte.

Les formules sont représentées par f. Les systèmes sont notés S, S', S'', les listes de motifs $\mathcal{L}, \mathcal{L}', \mathcal{L}''$. Enfin, $\mathcal{D}$ représente un domaine polyédrique, et e, e_1, e_2 des expressions polyédriques. Remarquons que la liste de formules n'est modifiée que par les règles de recherche de motifs et de recherche d'auto-dépendances.

– Axiomes à droite :

$$\frac{}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : \mathit{True} \downarrow \mathit{tt}; S, \mathcal{L}}\ \text{AX1 DROITE} \qquad \frac{}{\Gamma; S, \mathcal{L} \vdash \{\} : e \downarrow v; S, \mathcal{L}}\ \text{AX2 DROITE}$$

$$\frac{}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : \mathit{False} \downarrow \mathit{ff}; S, \mathcal{L}}\ \text{AX3 DROITE}$$

– Axiomes à gauche :

$$\frac{}{\mathcal{D} : \mathit{True} \downarrow \mathit{ff}, \Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}\ \text{AX1 GAUCHE} \qquad \frac{}{\mathcal{D} : \mathit{False} \downarrow \mathit{tt}, \Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}\ \text{AX3 GAUCHE}$$

– Élimination à gauche :

$$\frac{\Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}{\mathcal{D} : \mathit{True} \downarrow \mathit{tt}, \Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}\ \text{EL1} \qquad \frac{\Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}{\{\} : e \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}\ \text{EL2}$$

$$\frac{\Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}{\mathcal{D} : \mathit{False} \downarrow \mathit{ff}, \Gamma; S, \mathcal{L} \vdash f; S, \mathcal{L}}\ \text{EL3}$$

Il n'y a pas d'élimination à droite car notre système ne permet pas d'avoir des ensembles de formules à droite pour le moment.

– Règles de décompositions des opérandes à droite :

$$\frac{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e_1 \downarrow \mathit{tt}; S', \mathcal{L}', \quad \Gamma; S', \mathcal{L}' \vdash \mathcal{D} : e_2 \downarrow \mathit{tt}; S'', \mathcal{L}''}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e_1 \wedge e_2 \downarrow \mathit{tt}; S'', \mathcal{L}''} \text{ ET DROITE}$$

$$\frac{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e_1 \downarrow \mathit{ff}; S', \mathcal{L}', \quad \Gamma; S', \mathcal{L}' \vdash \mathcal{D} : e_2 \downarrow \mathit{ff}, S'', \mathcal{L}''}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e_1 \vee e_2 \downarrow \mathit{ff}; S'', \mathcal{L}''} \text{ OU DROITE}$$

– Règles de décompositions des opérandes à gauche :

$$\frac{\mathcal{D} : e_1 \downarrow \mathit{tt}, \mathcal{D} : e_2 \downarrow \mathit{tt}, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'}{\mathcal{D} : e_1 \wedge e_2 \downarrow \mathit{tt}, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}''} \text{ ET GAUCHE}$$

$$\frac{\mathcal{D} : e_1 \downarrow \mathit{ff}, \mathcal{D} : e_2 \downarrow \mathit{ff}, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'}{\mathcal{D} : e_1 \vee e_2 \downarrow \mathit{ff}, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}''} \text{ OU GAUCHE}$$

– Règle de la négation à droite :

$$\frac{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e \downarrow \mathit{ff}; S', \mathcal{L}'}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : \neg e \downarrow \mathit{tt}; S', \mathcal{L}'} \text{ NEG1 DROITE} \qquad \frac{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e \downarrow \mathit{tt}, S', \mathcal{L}'}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : \neg e \downarrow \mathit{ff}; S', \mathcal{L}'} \text{ NEG2 DROITE}$$

– Règle de la négation à gauche :

$$\frac{\mathcal{D} : e \downarrow \mathit{ff}, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'}{\mathcal{D} : \neg e \downarrow \mathit{tt}, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'} \text{ NEG1 GAUCHE} \qquad \frac{\mathcal{D} : e \downarrow \mathit{tt}, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'}{\mathcal{D} : \neg e \downarrow \mathit{ff}, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'} \text{ NEG2 GAUCHE}$$

– Règle de la dépendance à droite :

$$\frac{\Gamma; S, \mathcal{L} \vdash d(\mathcal{D}) : e : v; S', \mathcal{L}'}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : e.d \downarrow v; S', \mathcal{L}'} \text{ DEP DROITE}$$

– Règle de la dépendance à gauche :

$$\frac{d(\mathcal{D}) : e : v, S', \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'}{\mathcal{D} : e.d \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'} \text{ DEP GAUCHE}$$

– Règle de l'équation à droite :

$$\frac{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad \downarrow v; S', \mathcal{L}' \\ \mathcal{D}_n : e_n \end{cases}}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S', \mathcal{L}'} \text{ EQ DROITE} \quad \text{si} \quad X = \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad \text{est une équation de } S \\ \mathcal{D}_n : e_n \end{cases}$$

– Règle de l'équation à gauche :

$$\frac{\mathcal{D} : \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}' \\ \mathcal{D}_n : e_n \end{cases}}{\mathcal{D} : X \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'} \text{ EQ GAUCHE} \quad \text{si} \quad X = \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad \text{est une équation de } S \\ \mathcal{D}_n : e_n \end{cases}$$

– Règle de séparation des branches à droite :

$$\frac{\Gamma; S, \mathcal{L} \vdash \mathcal{D} \cap \mathcal{D}_1 : e_1 \downarrow v, S_1, \mathcal{L}_1 \quad \ldots \quad , \Gamma; S_{n-1}, \mathcal{L}_{n-1} \vdash \mathcal{D} \cap \mathcal{D}_n : e_n \downarrow v; S_n, \mathcal{L}_n}{} \text{ SEP DROITE}$$

$$S, \mathcal{L} \vdash \begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad\qquad \downarrow v; S_n, \mathcal{L}_n \\ \mathcal{D}_n : e_n \end{cases}$$

– Règle de séparation des branches à gauche :

$$\frac{\mathcal{D} \cap \mathcal{D}_1 : e_1 \downarrow v, \ldots, \mathcal{D} \cap \mathcal{D}_n : e_n \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}'}{} \text{ SEP GAUCHE}$$

$$\begin{cases} \mathcal{D}_1 : e_1 \\ \vdots \qquad\qquad \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S', \mathcal{L}' \\ \mathcal{D}_n : e_n \end{cases}$$

– Règle de l'ajout de variable à droite :

$$\frac{S', \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}'}{S, \mathcal{L} \vdash \mathcal{D} : e \downarrow v; S'', \mathcal{L}'} \text{ NOUV DROITE}$$

où S' est le système S auquel on a ajouté la variable X définie par l'expression e sur le domaine de e.

– Règle de l'ajout de variable à gauche :

$$\frac{\mathcal{D} : X \downarrow v, \Gamma; S', \mathcal{L} \vdash f; S'', \mathcal{L}'}{\mathcal{D} : e \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}'} \text{ NOU GAUCHE}$$

où S' est le système S auquel on a ajouté la variable X définie par l'expression e sur le domaine de e.

– Règle de la simplification à droite :

$$\frac{\Gamma; S', \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}'}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}'} \text{ SIMP DROITE}$$

Le système S' est le système S dans lequel toutes les constantes ont été supprimées en appliquant récursivement les techniques de simplification vues dans le chapitre 4.

– Règle de la simplification à gauche :

$$\frac{\mathcal{D} : X \downarrow v, \Gamma; S', \mathcal{L} \vdash f; S'', \mathcal{L}'}{\mathcal{D} : X \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}'} \text{ SIMP GAUCHE}$$

Le système S' est le système S dans lequel toutes les constantes ont été supprimées en appliquant récursivement les techniques de simplification vues dans le chapitre 4.

– Règle de substitutions par constante à droite :

$$\frac{\Gamma; S', \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}'}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}'} \text{ SUBS DROITE}$$

où toutes les occurrences de X sur le domaine le système $\mathcal{D}$ ont été substituées par v dans le système S'.

– Règle de substitutions par constante à gauche :

$$\frac{\mathcal{D} : X \downarrow v, \Gamma; S', \mathcal{L} \vdash f; S'', \mathcal{L}'}{\mathcal{D} : X \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}'} \text{ SUBS GAUCHE}$$

où toutes les occurrences de X sur le domaine le système $\mathcal{D}$ ont été substituées par v dans le système S'.

– Règle de recherche d'auto-dépendances à droite :

$$\frac{S', \mathcal{L}' \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}''}{S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}''} \text{ AUTO DROITE}$$

où S' est le système S dans lequel la définition de la variable X est substituée par la définition récursive obtenue par la recherche d'auto-dépendance et $\mathcal{L}'$ contient la liste $\mathcal{L}$ et le motif associé à X.

– Règle de recherche d'auto-dépendances à gauche :

$$\frac{\mathcal{D} : X \downarrow v, \Gamma; S', \mathcal{L}' \vdash f; S'', \mathcal{L}''}{\mathcal{D} : X \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}''} \text{ AUTO GAUCHE}$$

où S' est le système S dans lequel la définition de la variable X est substituée par la définition récursive obtenue par la recherche d'auto-dépendance et $\mathcal{L}'$ contient la liste $\mathcal{L}$ et le motif associé à X.

– Règle de recherche de motifs à droite :

$$\frac{\Gamma; S', \mathcal{L}' \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}''}{\Gamma; S, \mathcal{L} \vdash \mathcal{D} : X \downarrow v; S'', \mathcal{L}''} \text{ MOTIF DROITE}$$

Sur le domaine $\mathcal{D}$ la variable X est définie par un motif e. Le système S' correspond au système dans lequel toutes les occurrences du motif e ont été substituées par X, puis une substitution par constante de X sur le domaine $\mathcal{D}$ a été effectuée. La liste de motifs $\mathcal{L}'$ contient la liste $\mathcal{L}$ et le motif associé à X.

– Règle de recherche de motifs à gauche :

$$\frac{\mathcal{D} : X \downarrow v, \Gamma; S', \mathcal{L}' \vdash f; S'', \mathcal{L}''}{\mathcal{D} : X \downarrow v, \Gamma; S, \mathcal{L} \vdash f; S'', \mathcal{L}''} \text{ MOTIF GAUCHE}$$

Sur le domaine $\mathcal{D}$ la variable X est définie par un motif e. Le système S' correspond au système dans lequel toutes les occurrences du motif e ont été substituées par X, puis une substitution par constante de X sur le domaine $\mathcal{D}$ a été effectuée. La liste de motifs $\mathcal{L}'$ contient la liste $\mathcal{L}$ et le motif associé à X.

Comme les règles de preuve s'appliquant au contexte et à la partie droite du séquent sont identiques l'implémentation est donc identique.

2 Systèmes de règles de preuve des formules généralisées

Dans cette section nous nous contentons de donner les règles de preuve des formules généralisées. Nous ne donnerons aucune définition pour la sûreté de ces règles et nous ne prouverons pas plus qu'elles sont sûres. Ces règles sont données dans un but purement informatif.

Définition F.1 (règles de preuve des formules généralisées) *Soient $\mathcal{F}_1$ et $\mathcal{F}_2$ des formules généralisées. Soient Γ, Δ des ensembles de formules, soient S, S', S'' des systèmes et soient $\mathcal{L}, \mathcal{L}', \mathcal{L}''$ des listes de motifs. Les règles de preuve de bases sont définies de la manière suivante :*

– *règle de la conjonction*

$$\frac{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1\} \qquad \Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_2\}}{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1 \wedge \mathcal{F}_2\}} \text{ CONJ. DROITE}$$

$$\frac{\{\mathcal{F}_1, \mathcal{F}_2\} \cup \Gamma; S \Vdash^{\ast} \Delta}{\{\mathcal{F}_1 \wedge \mathcal{F}_2\} \cup \Gamma; S \Vdash^{\ast} \Delta} \text{ CONJ. GAUCHE}$$

– *règle de la disjonction*

$$\frac{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1, \mathcal{F}_2\}}{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1 \wedge \mathcal{F}_2\}} \text{ DISJ. DROITE}$$

$$\frac{\{\mathcal{F}_1\} \cup \Gamma; S \Vdash^{\ast} \Delta \qquad \{\mathcal{F}_2\} \cup \Gamma; S \Vdash^{\ast} \mathcal{F}}{\{\mathcal{F}_1 \vee \mathcal{F}_2 2\} \cup \Gamma; S \Vdash^{\ast} \Delta} \text{ DISJ. GAUCHE}$$

– *règle de l'implication*

$$\frac{\{\mathcal{F}_1\} \cup \Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_2\}}{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1 \implies \mathcal{F}_2\}} \text{ IMP. DROITE}$$

$$\frac{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1\} \qquad \{\mathcal{F}_2\} \cup \Gamma; S \Vdash^{\ast} \Delta}{\{\mathcal{F}_1 \implies \mathcal{F}_2\} \cup \Gamma; S \Vdash^{\ast} \Delta} \text{ IMP. GAUCHE}$$

– *règle de la négation*

$$\frac{\{\mathcal{F}_1\} \cup \Gamma; S \Vdash^{\ast} \Delta}{\Gamma; S \Vdash^{\ast} \Delta \cup \{\neg \mathcal{F}_1\}} \text{ NÉG. DROITE}$$

$$\frac{\Gamma; S \Vdash^{\ast} \Delta \cup \{\mathcal{F}_1\}}{\{\neg \mathcal{F}_1\} \cup \Gamma; S \Vdash^{\ast} \Delta} \text{ NÉG. GAUCHE}$$

– *règle du passage aux formules. Si Γ et Δ sont des ensembles de formules, et $\Delta = f_1, \ldots, f_n$, nous avons alors pour $i \in \{1, \ldots, n\}$:*

$$\frac{\Gamma; S \vdash f_i\}}{\Gamma; S \Vdash^{\ast} \{f_1, \ldots, f_n\}} \text{ FORMULE } i$$

Bibliographie

[1] ANSI/IEEE Std 1076-1987 : *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, USA, mars 1988.

[2] P.A ABDULLA : *Decision Problems in Systolic Circuit Verification*. Thèse de doctorat, Uppsala University, 1990.

[3] R. ALUR, R. GROSU, Y. HUR, V. KUMAR et I. LEE : Modular specification of hybrid systems in CHARON. *In* N.LYNCH et B.KROGH, éditeurs : *Proceedings of the 3rd International Workshop on Hybrid Systems : Computation and Control*, volume 1790, pages 6–19, Pittsburgh, PA, mar 2000. Lecture Notes in Computer Science.

[4] K. R. APT et D. C. KOZEN : Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[5] T. ARONS, A. PNUELI, S. RUAH, J. XU et L. ZUCK : Parameterized verification with automatically computed inductive assertions. *Lecture Notes in Computer Science*, 2102, 2001.

[6] E. ASARIN, T. DANG et O. MALER : The $\mathbf{d/dt}$ tool for verification of hybrid systems. *Lecture Notes in Computer Science*, 2404:365–369, 2002.

[7] R. BAGNARA, P. M. HILL, E. RICCI et E. ZAFFANELLA : Precise widening operators for convex polyhedra. Quaderno 312, Dipartimento di Matematica, Università di Parma, Italy, 2003.

[8] D. BARTHOU, P. FEAUTRIER et X. REDON : On the equivalence of two systems of affine recurrence equations. Rapport technique 4285, INRIA, 2001.

[9] D. BERRY : Formal methods : The very idea. some thoughts about why they work when they work. *Science of computer Programming*, 42(1), jan 2002.

[10] F. BESSON, T. JENSEN et J.-P. TALPIN : Polyhedral analysis for synchronous languages. *In* A. CORTESI et G. FILÉ, éditeurs : *Static Analysis*, volume 1694 de *Lecture Notes in Computer Science*, pages 51–68, Berlin, 1999. Springer-Verlag.

[11] B. BOIGELOT : *Symbolic Methods for Exploring Infinite State Spaces*. Thèse de doctorat, Université de Liège, 1998.

[12] B. BOIGELOT et P. WOLPER : Symbolic verification with periodic sets. *In Proceedings of the 6th International Conference on Computer-Aided Verification*, volume 818, pages 55–67. Lecture Notes In Computer Science Springer-Verlag, 1994.

[13] D. BORRIONE, L. PIERRE et A.M. SALEM : Formal verification of VHDL descriptions in the PREVAIL environment. *IEEE Design & Test of Computers*, pages 42–55, juin 1992.

[14] R.S. BOYER et J.S. MOORE : *A computational Logic*. Academic Press, New-York, 1979.

[15] R.E. BRYANT : Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, aug 1986.

[16] J.R. BURCH, E.M. CLARKE, K.L. MCMILLAN, D.L. DILL et L.J. HWANG : Symbolic model checking : 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[17] D. CACHERA : *Validation formelle des langages à parallélisme de données*. Thèse de doctorat, École normale supérieur de Lyon, 1998.

[18] D. CACHERA et K. MORIN-ALLORY : Verification of control properties in the polyhedral model. *In Proceedings of Memocode 2003*, IEEE, Mont-St-Michel, France, jun 2003.

[19] D. CACHERA et K. MORIN-ALLORY : Verification of control properties in the polyhedral model. Rapport technique 4756, INRIA, 2003.

[20] D. CACHERA et K. MORIN-ALLORY : Verification of safety properties for parameterized regular systems. *Transactions on Embedded Computing Systems, ACM*, 2004. à paraître.

[21] D. CACHERA et D. PICHARDIE : Embedding of systemes of affine recurrence equations in Coq. *In Proceedings of Theorem Proving in Higher Order Logics 2003*, volume 2758 de *LNCS*, Roma, sep 2003.

[22] D. CACHERA, P. QUINTON, S. RAJOPADHYE et T. RISSET : Proving properties of multidimensional recurrences with application to regular parallel algorithms. *In FMPPTA'01*, San Francisco, CA, avril 2001.

[23] E.M. CLARKE, E.A. EMERSON et A.P. SISTLA : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[24] E.M. CLARKE, O. GRUMBERG et S. JHA : Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):726–750, 1997.

[25] E.M. CLARKE et J.M. WING : Formal methods : state of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[26] J. COLLARD, T. RISSET et P. FEAUTRIER : Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, septembre 1995.

[27] P. COUSOT et N. HALBWACHS : Automatic discovery of linear restraints among variables of a program. *In Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM Press, 1978.

[28] C. DEZAN et P. QUINTON : Verification of regular architectures using Alpha : a case study. Internal publication 823, IRISA, Campus de Beaulieu, Rennes, France, 1994.

[29] F. DUPONT DE DINECHIN : *Systèmes stucturés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. Thèse de doctorat, Université de Rennes 1, 1997.

[30] E.A. EMERSON : Temporal and model logic. *In* J. VAN LEEUWEN, éditeur : *Handbook of Theoritical Computer*, volume B, pages 995–1072. The MIT Press/Elsevier Science Publisher, 1990.

[31] E.A. EMERSON et J.Y. HALPERN : "sometimes" and "not never" revisited : on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[32] E.A. EMERSON et K.S. NAMJOSHI : Reasoning about rings. *In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–94. ACM Press, 1995.

[33] E.A. EMERSON et K.S. NAMJOSHI : Automatic verification of parameterized synchronous systems. *In* R. ALUR et T.A. HENZINGER, éditeurs : *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 de *Lecture Notes in Computer Science*, pages 87–98, Berlin, juillet/août 1996. Springer Verlag.

[34] E.A. EMERSON et A.P. SISTLA : Utilizing symmetry when model-checking under fairness assumptions : An automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, juillet 1997.

[35] P. FEAUTRIER : Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[36] Paul FEAUTRIER : Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[37] Paul FEAUTRIER : Automatic parallelization in the polytope model. *In The Data Parallel Programming Model*, pages 79–103, 1996.

[38] S.M. GERMAN et A.P. SISTLA : Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 39(3):675–735, 1992.

[39] B. GOLDBERGAND, L. ZUCK et C. BARRETT : Into the loops : Practical issues in translation validation for optimizing compilers. *In Third International Workshop on Compiler Optimization meets Compiler Verificaiton (COCV)*, volume apr, Barcelona, Spain, apr 2004.

[40] M.J.C. GORDON : Hol : A proof generating system for higher-order logic. *In* G. BIRTWISTLE et P.A. SUBRAHMANYAM, éditeurs : *VLSI Specification, Verifcation and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1987.

[41] A. GUPTA : Formal hardware verification methods : A survey. *Journal of Formal Methods in System Design*, pages 151–238, 1992.

[42] V. HALAVA : Decidable and undecidable problems in matrix theory. Technical Report TUCS-TR-127, Turku Centre for Computer Science, Finland, septembre 30, 1997.

[43] N. HALBWACHS : *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. Thèse de doctorat, Université scientifique et médicale de Grenoble, mar 1979.

[44] N. HALBWACHS, F. LAGNIER et C. RATEL : An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6–7):523–543, novembre 1992.

[45] N. HALBWACHS, Y.-E. PROY et P. RAYMOND : Verification of linear hybrid systems by means of convex approximations. *Lecture Notes in Computer Science*, 864:223–237, 1994.

[46] S. HAYKIN : *Adaptive Filter Theory*. Prentice-Hall Information and System Science Series. Prentice Hall, Englewood Cliffs,New Jersey, 4th édition, 2002.

[47] M. HENNESSY : Proving systolic systems correct. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):344–387, 1986.

[48] T. A. HENZINGER, J. PREUSSIG et H. WONG-TOI : Some lessons from the HYTECH experience. *In Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892. IEEE Computer Society Press, 2001.

[49] T.A. HENZINGER et P.-H. HO : A note on abstract interpretation strategies for hybrid automata. *In* P.J. ANTSAKLIS, W. KOHN, A. NERODE et S. SASTRY, éditeurs : *Hybrid Systems II*, volume 999, pages 252–264. Springer-Verlag, Berlin, lecture notes in computer science édition, 1995.

[50] G.E. HUGHES et M.J. CRESWELL : *An introduction to modal logic*. London, 1977.

[51] R.M. KARP, R.E. MILLER et S. WINOGRAD : The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, juillet 1967.

[52] M. KAUFMANN et J.S. MOORE : ACL2 : an industrial stength version of Nqthm. *In Proceeding 11th Annual Conference on Computer Assurance (COMPASS'96)*, pages 23–34. IEEE Computer Society Press, jun 1996.

[53] W. KELLY, V. MASLOV, W. PUGH, E. ROSSER, T. SHPEISMAN et D. WONNACOTT : The Omega Library Interface Guide. Rapport technique, Dept. of Computer Science, Univ. of Maryland, College Park, avril 1996.

[54] C. KERN et M. R. GREENSTREET : Formal verification in hardware design : a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[55] Y. KESTEN, O. MALER, M. MARCUS, A. PNUELI et E. SHAHAR : Symbolic model checking with rich assertional languages. *In Proceedings of 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 de *Lecture Notes in Computer Science*, pages 424–435, 1997.

[56] Y. KESTEN, O. MALER, M. MARCUS, A. PNUELI et E. SHAHAR : Symbolic model checking with rich assertional languages. *TCS : Theoretical Computer Science*, 256:93–112, 2001.

[57] E. KINDLER : Safety and liveness properties : A survey. *EATCS-Bulletin*, 53:268–272, jun 1994.

[58] D. KOZEN : Results on the propositional mu-calculus. *Theoritical Computer Science*, 27:33–354, December 1983.

[59] H.T. KUNG : Why systolic architectures. *Computer 15*, pages 37–46, 1982.

[60] R. P. KURSHAN et K. MCMILLAN : A structural induction theorem for processes. *In Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 239–247. ACM Press, 1989.

[61] B. LE CHARLIER et P. FLENER : Specifications are necessarily informal or : Somme more myths of formal methods. *The Journal of Systems and Software*, 40(3):275–296, mar 1998.

[62] P. LE MOËNNER : A propos du langage ALPHA et de la vérificaiton d'architectures régulières synchrones. Mémoire de D.E.A., Université de Rennes 1, 1994.

[63] H. LE VERGE : *Un environnement de transformations de programmes pour la synthèse d'architectures régulières*. Thèse de doctorat, Université de Rennes 1, 1992.

[64] D. LESENS, N. HALBWACHS et P. RAYMOND : Automatic verification of parameterized linear networks of processes. *In Conference Record of POPL '97 : The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346–357, Paris, jan 1997. ACM SIGACT and SIGPLAN, ACM Press.

[65] D. LESENS, N. HALBWACHS et P. RAYMOND : Automatic verification of parameterized linear networks of processes. *Theoretical Computer Science*, 255, 2001.

[66] N. LING et M. BAYOUMI : *Specification and Verification of Systolic Arrays*. World Scientific, 1999.

[67] M. MANJUNATHAIAH, G.M. MEGSON, S. RAJOPADHYE et T. RISSET : Uniformization tool for systolic array designs. *In ICPP 01*, 2001.

[68] Z. MANNA et A. PNUELI : Verification of concurrent programs : The temporal framework. *In* R.S. BOYER et J.S. MOORE, éditeurs : *Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1982.

[69] C. MAURAS : *Alpha : un langage équationnel pour la conception et la programmation d'architectures systoliques*. Thèse de doctorat, Univ. Rennes I, France, décembre 1989.

[70] M. MÜLLER-OLM, D. SCHMIDT et B. STEFFEN : Model-checking : A tutorial introduction. *In* A. CORTESI et G. FILÉ, éditeurs : *Static Analysis*, volume 1694 de *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999.

[71] P. NAUDIN et C. QUITTÉ : *Algorithmique Algébrique*, chapitre III, pages 211–220. Masson, 1992.

[72] G. NAUMOVICH et L.A. CLARKE : Classifying properties : An alternative to the safety-liveness classification. *In Eighth International Symposium on the Foundations of Software Engineering*, pages 159–168, November 2000.

[73] G. NECULA : Translation validation of an optimizing compiler. *In Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI)*, 2000.

[74] D.C. OPPEN : A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, juin 1978.

[75] L. PIERRE : An automatic generalization method for the inductive proof of replicated and parallel architectures. *In* T. KROPF et R. KUMAR, éditeurs : *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 de *Lecture Notes in Computer Science*, pages 72–91, Bad Herrenalb, Germany, septembre 1994. Springer-Verlag. published 1995.

[76] L. PIERRE : "describing and verifying synchronous circuits with the boyer-moore theorem prover". *In* P. CAMURATI et H. EVEKING, éditeurs : *Correct Hardware Design and Verification Methods*, LNCS 987. Springer-Verlag, 1995.

[77] A. PNUELI, S. RUAH et L. ZUCK : Automatic deductive verification with invisible invariants. *Lecture Notes in Computer Science*, 2031, 2001.

[78] A. PNUELI et E. SHAHAR : Liveness and acceleration in parameterized verification. *In* A. EMERSON et P.S. SISTLA, éditeurs : *Conference on Computer Aided Verification (CAV'00)*, volume 1855 de *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, 2000.

[79] A. PNUELI, O. SHTRICHMAN et M. SIEGEL : The code validation tool CVT : Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[80] M. PRESBURGER : Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. *In Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.

[81] M. PRESBURGER : On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12:225–233, dec. 1991. translation of and commentary by A. Jacquette.

[82] W. PUGH et D. WONNACOTT : An exact method for analysis of value-based array data dependences. *Lecture Notes in Computer Science*, 768, 1994.

[83] F. QUILLERÉ et S. RAJOPADHYE : Optimizing memory usage in the polyhedral model. *ACM, Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[84] F. QUILLERÉ, S. RAJOPADHYE et D. WILDE : Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[85] P. QUINTON, S. RAJOPADHYE et T. RISSET : On manipulating Z-polyhedra using a canonical representation. *Parallel Processing Letters*, 7(2):181–194, juin 1997.

[86] P. QUINTON et Y. ROBERT : *Algorithmes et architectures systoliques*. Etudes et recherches en informatique, 0763-2770. Masson, Paris, 1989.

[87] Christophe REUTENAUER : *Aspects mathématiques des réseaux de Petri*. Masson, Paris, 1989.

[88] M. RINARD et D. MARINOV : Credible compilation with pointers. *In Proceedings of the Run-Time Result Verification Workshop*, Trento, jul 2000.

[89] Y. SAOUTER : *A propos de systèmes d'équations récurrentes*. Thèse de doctorat, Université de Rennes 1, 1992.

[90] Y. SAOUTER et P. QUINTON : Computability of recurrence equations. *Theoritical Computer Science*, 116, 1993.

[91] A. SCHRIJVER : *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986.

[92] N. SHANKAR, S. OWRE, J.M. RUSHBY et D.W.J. STRINGER-CALVERT : *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, décembre 2001.

[93] K.C. SHASHIDAR, M. BRUYNOOGHE, F. CATTHOOR (IMEC) et H. JANSSENS (K.U.LEUVEN) : Geometric model checking : An automatic verification technique for loop and data reuse transformations. *In International Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, pages 71–86, apr 2002.

[94] B. I. SILVA et B. H. KROGH : Formal verification of hybrid system using CheckMate : A case study. *In American Control Conference.*, jun 2000.

[95] I. SMARANDACHE : *Transformations affines d'horloges : application au codesign de systèmes temps-réél en utilisant les langages Signal et Alpha*. Thèse de doctorat, Université de Rennes 1, 1998.

[96] LogiCal Project The Coq Development TEAM : *The Coq proof Assistant, Reference Manual*.

[97] V. VAN DONGEN et P. QUINTON : Uniformization of linear reccurence equations : a step towards the automatic synthesis of systolic arrays. *In International Conference on Systolic Arrays*, San Diego (EU), Mai 1988.

[98] S. VERDOOLAEGE, R. SEGHIR, K. BEYLS, V. LOECHNER et M. BRUYNOOGHE : Analytical computation of Ehrhart polynomials : Enabling more compiler analyses and optimizations. *In Proceedings of the 2004 International Confernce on Compilers, Architectur, and Synthesis for Embedded Systems (CASES 2004)*, pages 248–258. ACM SIGMICRO, IEEE Computer Society TC uARCH, IBM Research, ACM SIGBED, 2004.

[99] D.K. WILDE : A library for doing polyhedral operations. Rapport technique 785, IRISA, Rennes, France, jan 1993.

[100] D.K. WILDE : The Alpha language. Rapport technique 999, IRISA, Rennes, France, jan 1994.

[101] P. WOLPER et V. LOVINFOSSE : Verifying properties of large sets of processes with network invariants. *In Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 68–80. Springer-Verlag New York, Inc., 1990.

[102] Z. ZHOU et W. BURLESON : Formal descriptions, semantics and verification of VLSI array processors. *In International Conference on Application-Specific Array Processors*, pages 321–332, 1993.

[103] L. ZUCK, A. PNUELI, B. GOLDBERG, C. BARRETT, Y. FANG et Y. HU : Translation and run-time validation of loop tranformations. *Formal Methods in System Design (FMSD)*, 2004. à paraître.

Table des matières

Table des figures

VU :
Le Directeur de Thèse
(Nom et Prénom)

VU :
Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Bertrand FORTIN

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)

Résumé

Les travaux présentés dans ce document sont orientés vers la vérification formelle de propriétés de sûreté dans le cadre de la conception des systèmes enfouis. Nous nous plaçons dans le formalisme du modèle polyédrique, combinaison des systèmes d'équations récurrentes affines avec les polyèdes entiers. Ce modèle permet de faire de la synthèse de haut niveau pour générer des architectures parallèles à partir de la description d'un système régulier dont les dimensions sont définies par des paramètres symboliques. Nous nous intéressons à la vérification de propriétés de sûreté portant sur des signaux booléens de contrôle, générés ou introduits manuellement lors de la synthèse. Les propriétés sur de tels signaux seront appelées propriétés de contrôle. Nous montrons dans ce document que le modèle polyédrique est adapté pour la vérification formelle de propriétés de contrôle.

Dans ce travail, nous développons une "logique polyédrique" qui nous permet de spécifier et prouver des propriétés dans le modèle polyédrique. La syntaxe et la sémantique des formules logiques s'appuient sur celles d'un langage de description de systèmes d'équations récurrentes affines sur des domaines polyédriques. Les règles de déduction sont de différents types : des règles "classiques" sur les connecteurs logiques, des règles de réécriture et des règles induites par des calculs dans le modèle. Nous développons des algorithmes pour automatiser la construction des preuves, ainsi que des techniques heuristiques permettant d'accélérer cette construction. Ces algorithmes nous permettent de prouver des propriétés simples, comme par exemple la propriété qu'un signal vaut toujours vrai pour un ensemble de processeurs et une durée déterminés. Nous présentons ensuite et commençons à développer des pistes afin d'enrichir notre logique pour exprimer des propriétés plus complexes, comme par exemple des propriétés d'exclusion mutuelle. Nous présentons quelques tactiques de preuve pour ces propriétés plus riches.

Mots Clefs Méthodes formelles de vérification, langages de spécification matérielle, synthèse de très haut-niveau, systèmes enfouis, co-design, systèmes paramétrés, modèle polyédrique.

Abstract

This document deals with formal verification of safety properties in the context of embedded systems design. This work is based on the polyhedral model formalism, *i.e.*, the combination of systems of affine recurrence equations with integer polyhedra. This model is used in high level synthesis fot generating parallel architectures from regular system descriptions, dimensions of which are defined by means of symbolic parameters. We are interested in the verification of safety properties about boolean control signals that are generated or manually introduced during the synthesis. We call control properties properties on such signals. We show in this document that the polyhedral model is well suited to the formal verification of control properties.

In this work, we present a "polyhedral logic" that allows for specifying and proving properties in the polyhedral model. The syntax and semantics of logical formulae are based on those of a description language designed for systems of affine recurrence equations on polyhedral domains. There are different kinds of deductions rules : "classical rules" on logical connectors, rewriting rules and rules induced by the computations in the model. We present an algorithm to automatize the proof construction, and heuristic techniques to speed up this construction. These algorithms allow for proving simple properties like the fact that a signal is always true for a given set of processors and time instants. We then sketch and begin to develop solutions that can be used to expand our logic so as to express and prove more complex properties, like mutual exclusion properties for instance. We give some proof tactics for this augmented formalism.

Keywords formal verification methods, hardware specification languages, high level synthesis, embedded systems, co-design, parametred systems, polyhedral model.