Raffinement et preuves de systèmes Lustre

Jan Mikac

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

T H È S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : « Informatique : Systèmes et Communications »

préparée au laboratoire VERIMAG
dans le cadre de **l'École doctorale *« Mathématiques, Sciences et Technologies de l'Information »***

présentée et soutenue publiquement

par

Jan MIKÁČ

le 14 novembre 2005

Titre :

Raffinements et preuves de systèmes Lustre

Directeur de thèse :

Paul CASPI

JURY

Mme. Florence Maraninchi	Présidente
Mme. Véronique Viguié Donzeau-Gouge	Rapporteuse
M. Robert De Simone	Rapporteur
M. Paul Caspi	Directeur de thèse
Mme. Marie-Laure Potet	Examinatrice
M. Pierre Chartier	Examinateur

Remerciements

Je souhaiterais remercier tout d'abord Paul Caspi, qui est à l'origine de mon travail et qui a été mon directeur de thèse pendant les trois années. Me lançant sur de nouvelles pistes lorsque cela était opportun, il m'a cependant laissé un large champ d'initiative personnelle, ce qui, j'en ai peur, est à l'origine de quelques cheveux blancs supplémentaires sur sa tête. Si la thèse a pu être menée jusqu'au bout, c'est grâce à ses conseils bienveillants et à sa science clairvoyante.

Je remercie également les membres du jury qui ont accepté d'étudier et d'évaluer mon travail. Les remarques constructives qu'ils ont formulées m'ont permis de voir plus loin.

Le laboratoire Vérimag a constitué pour moi un lieu de travail amical où règne une ambiance détendue. Je remercie son directeur, Joseph Sifakis, de m'y avoir accueilli, ainsi que tous les autres membres de m'avoir accepté parmi eux. Je remercie en particulier Liana et Lionel pour leur amitié et leurs conseils.

Une thèse n'est pas seulement un exercice académique : ce sont également trois années dans une vie qui mettent à l'épreuve le candidat et son entourage. Merci, Patou, de m'avoir supporté et soutenu durant tout ce temps. Merci, papy, de n'avoir cessé de croire en moi.

Et aussi un grand merci à tous ceux qui m'ont permis de ne pas m'enfermer dans la science et d'avancer également dans une autre voie : merci Alan, Antoine, Claire, Claude, Cyrille, Eric, Fiona, Françoise, Julien, Marilyne, Marine, Mimi et les autres.

Table des matières

I Introduction **9**

1 Introduction générale **11**
1.1 Contexte général de la thèse 11
1.1.1 Méthodes formelles 11
1.1.2 Systèmes réactifs synchrones 13
1.2 Contexte particulier 15
1.3 Plan du mémoire 15

2 Le langage Lustre **17**
2.1 Présentation informelle 17
2.1.1 Généralités 17
2.1.2 L'exemple de l'intégrateur discret 18
2.2 Définition formelle de Lustre 20
2.2.1 Les briques de base 20
2.2.2 Les nœuds 22
2.2.3 Les programmes 25
2.2.4 Les horloges 26
2.2.5 Conclusion 28

3 Méthode de preuve inductive de Cécile Dumas **29**
3.1 Position du problème 29
3.2 Méthode de preuve 30
3.3 Exemple de preuve 32
3.4 Règles supplémentaires 34
3.4.1 Le cas de l'opérateur when 34
3.4.2 Le cas des hypothèses supplémentaires 35
3.5 Règle d'induction continue revue 36
3.5.1 L'échec sur la suite de Fibonacci 36
3.5.2 La correction ratée 38
3.6 Bilan de la méthode 44

II Travail sur les preuves **47**

4 L'induction continue fragmentée **49**
4.1 Justification de la fragmentation 49
4.2 Dérivation d'une règle fragmentée 51
4.3 L'intérêt de l'induction continue fragmentée 52

5 Dépliages en bloc **55**
5.1 Règle de dépliage . . . 55
5.2 Combien de dépliages ? . . . 58
5.2.1 Formalisation . . . 58
5.2.2 Caractérisation des invariants prouvables par n dépliages . . . 60
5.3 Conclusion . . . 63

6 Outil de preuve Gloups **65**
6.1 Présentation générale . . . 65
6.1.1 Motivations . . . 65
6.1.2 Brève présentation . . . 66
6.2 Algorithme utilisé . . . 70
6.2.1 Dépliages et généralisations . . . 70
6.2.2 Avancements et induction . . . 71
6.3 Les fichiers PVS créés . . . 75
6.4 Optimisations . . . 76
6.4.1 Représentation optimisée d'expressions . . . 76
6.4.2 Optimisations des séquents . . . 78
6.4.3 Apport pratique des optimisations . . . 80
6.5 Exemple du synchronisateur d'horloges . . . 80
6.5.1 Présentation des circuits . . . 80
6.5.2 La preuve de l'équivalence . . . 81
6.6 Bilan . . . 82

III Le raffinement **85**

7 Introduction et motivations **87**
7.1 Le raffinement en général . . . 87
7.2 La méthode B . . . 88
7.2.1 Machines abstraites . . . 88
7.2.2 Raffinements . . . 90
7.3 La méthode B et Lustre . . . 92
7.3.1 Lustre en B . . . 92
7.3.2 Lustre sans B . . . 93

8 Raffinement pour Lustre **95**
8.1 Cadre général . . . 95
8.1.1 Systèmes réactifs non-déterministes . . . 95
8.1.2 Un premier raffinement . . . 96
8.1.3 Transitivité du raffinement . . . 98
8.1.4 Bilan du premier raffinement . . . 98
8.2 Raffinement adapté . . . 99
8.2.1 Sous-typage des domaines et co-domaines . . . 99
8.2.2 Prise en compte des flots . . . 101
8.2.3 Les relations de changement de variables . . . 103
8.2.4 Les relations en Lustre . . . 105
8.3 Bilan du raffinement . . . 106
8.3.1 Résumé du raffinement . . . 106
8.3.2 Analogie avec les circuits . . . 106

8.3.3 De l'aspect temporel du raffinement proposé . . . 107
8.3.4 Le problème de l'équité . . . 107

9 Exemples de raffinement **111**
9.1 Exemple du compteur modulo n . . . 111
9.1.1 Spécifications . . . 111
9.1.2 Premier raffinement . . . 112
9.1.3 Deuxième raffinement . . . 114
9.2 Exemple de l'île d'Abrial . . . 117
9.2.1 L'île . . . 117
9.2.2 Un contrôleur non-déterministe . . . 118
9.2.3 Raffinement du temps . . . 120
9.2.4 Le contrôleur déterministe . . . 122

10 Conclusion et perspectives **125**

IV Annexes **127**

A Résumé de Lustre **129**

B Processus de preuve **135**
B.1 Règles de preuve . . . 135
B.1.1 Règles de déduction . . . 135
B.1.2 Règles de simplification . . . 135
B.2 Algorithme de preuve . . . 136

C Fichiers pour PVS **139**
C.1 Obligations de preuve pour la suite de Fibonacci . . . 139
C.2 Théorie sous-jacente . . . 145
C.2.1 Les opérateurs div et mod . . . 145
C.2.2 Les opérateurs int et real . . . 147
C.2.3 L'opérateur 'au plus un' . . . 147
C.3 Stratégies de réécriture . . . 148
C.3.1 Réécriture des opérateurs Lustre . . . 148
C.3.2 Élimination de variables . . . 149
C.3.3 Élimination des constantes . . . 150

Bibliographie **153**

Résumé des chapitres **156**

Résumé de la thèse **157**

Première partie

Introduction

1 – Introduction générale

Pour délimiter le cadre de cette thèse, nous la replaçons dans le contexte des méthodes formelles appliquées aux systèmes réactifs synchrones. Nous citons également un travail antérieur, sur lequel elle s'appuie et qu'elle prolonge directement. Nous terminons en annonçant le plan du document.

1.1 Contexte général de la thèse

Avant de commencer l'exposé de nos travaux, nous allons d'abord situer la thèse dans un contexte plus large, afin de permettre au lecteur de s'orienter plus facilement. En effet, ce mémoire se situe à la rencontre de deux domaines de l'Informatique : celui des méthodes formelles et celui des systèmes réactifs synchrones. Nous allons présenter ces deux domaines tour à tour.

1.1.1 Méthodes formelles

Premier vol d'essai réussi pour l'A380

L'A380 a réussi son premier essai en vol. Le très gros porteur d'Airbus s'est posé aujourd'hui à 14h22 sur l'aéroport de Toulouse-Blagnac à l'issue d'un vol de 3h53. L'avion à deux ponts sera capable de transporter plus de 550 passagers en configuration trois classes et 840 en configuration charter. Airbus entend ainsi prendre une longueur d'avance sur son principal concurrent, l'Américain Boeing, dont le 747 est pour l'instant le plus gros porteur civil, capable d'emporter plus de 400 passagers en mode économique. [...]

Sciences et Avenir du 27 avril 2005

Premier vol décisif pour l'Airbus A380

[...] Pour gérer le problème de l'aéroélasticité mais aussi une vitesse inédite de Mach 0,86, les ingénieurs maison ont affûté, itération après itération, l'aérodynamique lourde de l'avion à deux ponts, et **les commandes de vol électriques.** Ces logiciels qui commandent les surfaces de contrôle de l'avion sont réglés pour prendre en compte les déformations dynamiques de l'avion afin de compenser le manque de contrôle des pilotes. Leur efficacité a même servi à réduire la taille de la dérive qui était initialement démesurément grande pour stabiliser l'avion sur son axe de lacet. **La technologie des commandes de vol électrique est éprouvée depuis quelques décennies mais pas à ce degré de sophistication. C'est pourquoi Airbus a collaboré plusieurs années avec l'Office national d'études et de recherches aérospatiales (Onera) pour préparer leurs méthodes de conception** [...]

Les Échos du 27 avril 2005

Le premier vol du nouvel Airbus A380 constitue une occasion pour attirer l'attention sur l'importance des méthodes formelles dans le développement des systèmes informatiques. En effet, sans la rigueur et l'appareillage scientifique que ces méthodes apportent, il n'aurait sans doute pas été possible de mener à bien un projet d'une telle envergure.

À la base de toute méthode formelle, se situe un cadre mathématique dans lequel sont décrits les comportements des systèmes et leurs propriétés. Grâce à ce formalisme, la description est précise et non-ambiguë, ce qui constitue déjà un but en soi : on dispose ainsi d'un modèle de ce que l'on demande au système de faire (les spécifications) et de ce qu'il fait effectivement (l'implémentation).

Vérification Une première application de cette formalisation est la vérification des systèmes : grâce au formalisme mathématique, on peut effectuer des preuves de correction de l'implémentation par rapport à sa spécification. De telles preuves fournissent alors la certitude que les propriétés exprimées dans les spécifications sont bien respectées par le système. Au-delà d'un gain de qualité indéniable, ces preuves formelles peuvent également avoir un rôle légal : en effet, certains systèmes, et en particulier ceux qui pourraient mettre des vies humaines en danger (les *systèmes critiques*), sont soumis aux obligations de certification. Ainsi, avant de pouvoir être exploité commercialement, l'A380 devra fournir les preuves d'un certain nombre de propriétés et le logiciel de la commande de vol sera examiné au cours de cette démarche[1].

Il existe plusieurs méthodes de vérification. Dans le cas où le système (et sa spécification) ne peuvent se trouver que dans un nombre fini d'états, la méthode de vérification de modèles (le *model checking*) s'applique : il s'agit d'explorer tous les états du systèmes en vérifiant qu'ils satisfont les spécifications. Le model checking de base a l'avantage d'être entièrement automatique (des exemples de model checkers populaires sont Spin [23] et SMV [31]), mais trouve ses limites dans la taille des systèmes, car plus le nombre d'états à explorer est élevé, plus l'exploration est longue.

Pour pallier cet inconvénient, on utilise des méthodes visant à réduire l'espace à explorer par des approximations qui décrivent le problème de manière plus simple, nécessitant moins d'états : on retrouve ici le model checking par abstraction [7] et plus généralement l'interprétation abstraite [15]. Ces méthodes s'appliquent également dans le cas de systèmes infinis, mais nécessitent une interaction de la part de l'utilisateur pour décider ou diriger la décision de l'approximation à prendre. Elles sont également incomplètes dans le sens où elles ne permettent pas de prouver toutes les propriétés vraies des systèmes.

Pour terminer, complètement à l'opposé du model checking automatique, se trouve la preuve entièrement interactive où l'utilisateur humain rédige lui-même une preuve de la propriété voulue et l'outil de preuve (p.ex. Coq [24] ou PVS [41]) se contente de vérifier la correction de cette preuve. Cette dernière méthode est donc très puissante, grâce à la créativité de l'esprit humain, et dans certains cas la seule applicable. Néanmoins, elle est également la plus difficile à mettre en œuvre, car l'automatisation se limite dans ce cas à la preuve de sous-propriétés triviales.

Une partie du travail présenté dans ce mémoire se situe dans le domaine des preuves interactives : notre but était d'alléger le travail de preuve pour une classe de propriétés en proposant à l'utilisateur d'effectuer un pré-traitement du problème. Ce pré-traitement a pour but d'exprimer le problème sous une forme différente, qui est plus facile à prouver. Cet aspect de notre travail est exposé dans la partie II de ce document.

Conception L'intérêt des méthodes formelles ne se limite pas à leur utilisation pour vérifier les systèmes. Il serait en effet peu judicieux d'essayer de prouver la correction d'un système une fois qu'il est entièrement implémenté, car outre les problèmes inhérents à la taille du système, qui rend le processus de preuve difficile, se pose le problème de la découverte d'une erreur à ce stade tardif du développement : corriger cette erreur risquerait d'être coûteux et difficile. C'est pour cette raison que la formalisation mathématique est mise à profit tout au long du développement, pour valider les étapes intermédiaires : par exemple, on peut prouver que les spécifications détaillées d'un module vérifient bien ses spécifications générales. De cette manière, les éventuelles erreurs sont détectées plus tôt, mais en contre-partie, le développement est plus lourd du fait des obligations de preuves supplémentaires.

Parmi ces méthodes de développement de programmes corrects par construction, on trouve le *raffinement*. Intuitivement, il s'agit d'une méthode qui part de spécifications fonctionnelles d'un système, spécifications qui seront raffinées, c'est-à-dire enrichies de détails supplémentaires et éventuellement réécrites, jusqu'à ce qu'elles deviennent implémentables. Le raffinement fonctionne donc par étapes, où

[1] Il est vrai que la plupart des propriétés exigées de l'A380 seront examinées par des tests exhaustifs, mais l'application des méthodes formelles à certains sous-systèmes reste possible, comme l'a montré Lionel Morel dans sa thèse [37].

chaque nouvelle étape décrit le comportement du système avec plus de précision que l'étape précédente. Chaque spécification (ou programme au sens large) décrit l'ensemble de comportements admis du système développé ; le raffinement consiste alors à choisir un sous-ensemble de ces comportements. De cette manière, le raffinement produit des programmes qui sont par construction corrects vis-à-vis des spécifications initiales. La correction du raffinement (c'est-à-dire le fait de ne pas ajouter de comportement illégal) est vérifiée formellement au moyen d'obligations de preuve.

Formellement, le raffinement établit donc un ordre sur les programmes : un programme raffiné pourra être utilisé à la place du programme qu'il raffine, *sans qu'un observateur extérieur puisse s'en rendre compte*, car toutes les sorties observables du programme raffiné seront également admissibles par le programme « brut ». Dans cette optique, un programme implémentable est vu simplement comme une spécification raffinée à l'extrême, de sorte que son comportement soit parfaitement déterministe et donc mécanisable.

Parmi les méthodes de raffinement les plus populaires, citons les méthodes B [3], Z [11] et VDM [26]. Outre la vision théorique d'ordre sur les programmes, elles offrent également une panoplie de constructions et de procédés au service du développeur, si bien qu'elles constituent plus qu'un système formel – un cadre de développement à part entière[2]. Des approches plus théoriques du raffinement se trouvent par exemple dans TLA [27, 2] ou chez Back [6].

Les systèmes développés dans le langage Lustre ou dans sa variante industrielle Scade sont souvent des systèmes critiques de taille non négligeable. Il pourrait donc être intéressant de disposer d'un mécanisme de raffinement pour ces langages, afin de mieux maîtriser le développement des programmes. L'idée d'utiliser les méthodes de raffinement existantes[3] s'avère malheureusement peu pratique : traduire Lustre en B revient à le compiler, c'est-à-dire passer du paradigme fonctionnel synchrone au paradigme impératif, ce qui annule les avantages pour lesquelles Lustre/Scade est utilisé dans l'industrie.

Partant de ce constat, nous avons étudié la possibilité tant théorique que pratique de définir un calcul de raffinement adapté à Lustre, un calcul inspiré, mais indépendant, des méthodes de raffinement pré-citées. Les résultats de ce travail font l'objet de la troisième partie de notre mémoire.

1.1.2 Systèmes réactifs synchrones

Notre travail dans le domaine des méthodes formelles décrit dans la partie précédente a été développé pour les systèmes réactifs synchrones. Comme ces systèmes constituent une branche assez spécifique de l'Informatique, nous allons les présenter plus en détail.

Systèmes réactifs Les systèmes informatiques peuvent être classés en trois catégories différentes selon le type d'interaction qu'ils ont avec leur environnement :

- Certains programmes interagissent peu ou pas du tout avec d'autres systèmes : ils effectuent les mêmes calculs sur toutes les entrées et c'est pourquoi on les désigne comme systèmes *tranformationnels*. Un encodeur MPEG qui transforme une série d'images en un flux d'octets optimisé pourrait être un exemple d'un tel système.
- Les programmes les plus connus sont ceux qui interagissent avec l'utilisateur : ce sont des programmes qui, à l'instar d'un traitement de texte, attendent une action de la part de l'utilisateur humain, agissent en fonction de l'action et se remettent à attendre. Nous les appellerons systèmes *interactifs*.
- Enfin, il existe la catégorie des systèmes *réactifs* qui, comme les systèmes précédents, attendent des actions de la part de leur environnement. Lorsqu'une action survient, le système réactif calcule une réponse et *cette réponse influe sur son environnement*. Les systèmes réactifs établissent donc

[2] Il existe par exemple pour la méthode B un environnement de développement intégré, AtelierB.

[3] Par exemple une traduction possible de Lustre vers B est proposée dans [17] ; la traduction inverse fait l'objet de [9].

une boucle d'interaction avec leur environnement, ce qui n'est pas le cas dans les deux classes précédentes.

Les systèmes réactifs ne manquent pas en informatique, même si leur présence est souvent discrète, surtout par rapport aux systèmes interactifs, beaucoup plus connus des utilisateurs. Citons par exemple des systèmes industriels (surveillance de la température dans une cuve de pasteurisation, surveillance de la distance minimale entre deux trains consécutifs) ou des « services » de bas niveau fournis aux applications (maintien de la connexion lors du déplacement dans un téléphone portable). Le noyau d'un système d'exploitation peut être vu comme un ensemble de systèmes réactifs.

Les systèmes réactifs constituent donc une partie importante du paysage informatique. Leur étude théorique présente, elle aussi, des particularités du fait de l'interaction continuelle de ces systèmes avec leur environnement : en fait, pour pouvoir modéliser un programme réactif, il est nécessaire de comprendre et de modéliser également l'environnement avec lequel il interagit, ce qui constitue une différence notable par rapport aux systèmes informatiques plus traditionnels. Nous verrons une application pratique de ce principe au chapitre 9.

Systèmes synchrones La programmation synchrone est un paradigme bien adapté aux systèmes réactifs. On pourrait résumer ses principes en une seule devise : « Les calculs ne prennent pas de temps ». Il est évident que dans un tel modèle, un programme est sûr de pouvoir réagir à une sollicitation de son environnement avant que ne survienne l'événement suivant, d'où la facilité à concevoir des programmes réactifs [20].

En pratique, les calculs « en temps zéro » n'existent pas. Les langages synchrones comme Lustre [22] ou Esterel [10] ont adopté la politique de maîtrise du temps de calcul maximum en rejetant des constructions comme la récursivité ou l'allocation dynamique de données, qui introduiraient une incertitude quant à la durée des calculs. Un programme Lustre est alors conçu avec l'hypothèse du calcul instantané et c'est a posteriori qu'on vérifie, sur une plate-forme d'implémentation donnée, si deux événements successifs sont toujours suffisamment espacés dans le temps pour permettre aux calculs de se terminer.

D'autres langages synchrones, comme Signal [29] ou Lucide-Synchrone [13] comportent des contraintes moins strictes sur les constructions permises. La contrepartie de leur expressivité accrue est alors une moindre maîtrise du temps d'exécution. Néanmoins, les langages synchrones en général présentent des schémas de compilation efficaces, si bien que les temps d'exécution sont généralement faibles.

Parmi les nombreuses utilisations industrielles des langages synchrones, citons le contrôleur d'arrêt d'urgence dans les centrales nucléaires Framatome [8], la signalisation du métro de Hong-Kong [28], la commande de vol électronique d'Airbus [12] ou la validation d'une nouvelle architecture de mémoires électroniques [5]. On remarque que dans tous ces exemples, les langages synchrones sont utilisés pour implémenter des tâches critiques : grâce à leur sémantique formelle simple et bien définie, ces langages se prêtent bien à la vérification formelle, qui est primordiale dans de telles applications. À l'appui de la vérification, il existe aujourd'hui un grand nombre d'outils pour les langages synchrones, parmi lesquels on trouve Scade Suite et Esterel Studio d'Esterel Technologies, des interfaces depuis d'autres outils comme Simulink/Stateflow [38], un compilateur certifié Do178B niveau A, mais aussi un outil pour interprétation abstraite (nbac [25]), un plug-in de Prover [40] ou un model-checker (Lesar).

Notre étude est plus particulièrement axée sur le langage Lustre, dont elle exploite notamment les aspects synchrone et flot-de-données. C'est pourquoi il est nécessaire de familiariser le lecteur avec ce langage, avant d'entrer dans le vif du sujet. Comme une présentation détaillée dépasse le cadre de cette introduction, nous reviendrons à Lustre au cours du chapitre 2.

1.2 Contexte particulier

Après avoir situé notre travail dans les domaines très généraux des méthodes formelles et des systèmes réactifs synchrones, nous allons à présent évoquer le contexte particulier de cette thèse, dans la mesure où il est nécessaire pour bien comprendre certaines de nos motivations.

La thèse s'inscrit bien sûr dans le cadre des recherches de l'équipe Synchrone du laboratoire Vérimag[4], au sein de laquelle cette thèse a été menée entre septembre 2002 et août 2005. Mais elle se situe plus particulièrement dans le prolongement de la thèse de Cécile Dumas [18], soutenue en novembre 2000, qui propose une méthode de preuve déductive pour le langage Lustre et qui amorce une réflexion sur le raffinement. Nous rappellerons les points importants de son travail au chapitre 3.

1.3 Plan du mémoire

Pour conclure cette présentation, nous allons maintenant évoquer l'organisation de ce document :

- La première partie est dévolue à l'introduction : après avoir situé le contexte de l'étude, nous allons aborder plus en détail le langage Lustre (chapitre 2), puis nous consacrerons un chapitre aux travaux de Cécile Dumas.
- Dans la deuxième partie, nous étudierons la problématique des preuves inductives des invariants en Lustre : après avoir brièvement évoqué une première méthode à l'intérêt pratique limité au chapitre 4, nous discuterons la méthode finalement retenue (chapitre 5) et nous présenterons une implémentation pratique de cet algorithme au chapitre 6.
- La troisième partie sera dédiée au raffinement. Nous commencerons par présenter les méthodes classiques (chapitre 7), ensuite nous expliquerons notre proposition de raffinement adapté à Lustre au chapitre 8 et nous en donnerons un exemple au chapitre 9, avant de conclure sur l'ensemble de nos travaux.

[4] `http://www-verimag.imag.fr/SYNCHRONE`

2 – Le langage Lustre

Dédié à la présentation du langage Lustre, ce chapitre commence par une introduction informelle avec exemple. Ensuite, la définition formelle décrit tour à tour la syntaxe et la sémantique des flots, des expressions sur les flots et des fonctions de flots. Pour terminer, on s'attaque à la question des horloges et du synchronisme.

2.1 Présentation informelle

Le travail de cette thèse est centré autour de Lustre. Pour permettre au lecteur de comprendre plus facilement les chapitres suivants, nous proposons ici une présentation du langage avec, pour commencer, une approche plutôt informelle.

2.1.1 Généralités

Dans le paysage des langages de programmation, Lustre se démarque par un certain nombre de caractéristiques, dont certaines ont déjà été annoncées au chapitre précédent :

Lustre est un langage fonctionnel Les programmes décrivent la fonction d'un système plutôt que la manière de l'obtenir. En particulier, les structures impératives, telle que l'affectation ou des boucles itératives, n'existent pas en Lustre.

Lustre est un langage à flots de données Les variables du langage représentent des flots, c'est-à-dire des suites (finies ou infinies) de valeurs. Ainsi, on parlera de flots d'entiers, de booléens... ou de types plus complexes.

Lustre est un langage synchrone Nous avons dit dans l'introduction que les langages synchrones faisaient abstraction du temps de calcul, mais en fait, cette formulation est seulement une simplification du caractère synchrone. Plus précisément, celui-ci décrit la manière dont les flots de données peuvent être combinés : il y a en effet une contrainte forte qui interdit de combiner des flots de différentes longueurs (sauf à travers des opérateurs spéciaux). Comme cette caractéristique est essentielle en Lustre, nous tâcherons de la rendre plus claire par la suite.

Lustre est un langage de premier ordre Les « briques de base » utilisées pour définir les fonctions dans un programme Lustre sont uniquement les flots de données. À la différence d'autres langages fonctionnels, on ne peut pas définir des fonctions de fonctions en Lustre. Cette restriction va de pair avec le synchronisme : elle est destinée à limiter le pouvoir d'expression du langage afin d'assurer le caractère borné des systèmes décrits.

Ces quatre caractéristiques décrivent un langage aux propriétés intéressantes. Grâce à son caractère fonctionnel, Lustre est bien adapté pour spécifier des systèmes, car le rôle des spécifications est bien d'indiquer la fonctionnalité souhaitée, plutôt que la manière précise de l'obtenir. La limitation d'expressivité au premier ordre, l'absence de fonctions récursives et de toute création dynamique de données assurent que tout programme pourra être exécuté en mémoire (statiquement) bornée. Le caractère synchrone du langage permet en plus d'assurer que les calculs élémentaires se feront en temps borné. L'une des conséquences directes de ces propriétés est la compilation efficace des programmes Lustre.

Une autre conséquence du caractère borné des systèmes décrits, et surtout de l'aspect flot de données, est la possibilité de représenter un programme Lustre sous forme de circuit électronique et vice versa. La capacité de représenter très naturellement des circuits sous forme de programmes est l'un des points forts de Lustre, et constitue d'ailleurs la motivation première de son existence.

L'image qui se dégage de cette description est donc celle d'un langage adapté au domaine des systèmes réactifs : les calculs étant par construction bornés en temps et en espace, tout est fait pour assurer la réactivité des programmes. Le développeur peut alors utiliser l'abstraction du temps de calcul zéro, rendue d'autant plus facile à adopter par l'aspect fonctionnel du langage.

Afin d'illustrer les principaux points soulevés dans cette partie, nous proposons un exemple introductif, qui servira aussi à familiariser le lecteur avec la syntaxe du langage.

2.1.2 L'exemple de l'intégrateur discret

Nous nous proposons de concevoir un programme Lustre qui implémente l'intégration de fonction avec un pas discret. Plus précisément, nous souhaitons obtenir un système qui reçoit en entrée une suite de valeurs, représentant les valeurs successives de la fonction à intégrer, et qui rend en sortie la suite des valeurs intégrées « jusque là ». Formellement, si la suite des entrées est

$$(e_i)_{i\in\mathbb{N}}$$

alors la suite des sorties doit être

$$(s_i)_{i\in\mathbb{N}} = \left(\sum_{j=0}^{i} e_j\right)_{i\in\mathbb{N}}$$

Le composant de base d'un programme Lustre est un nœud. Un nœud représente une fonction et est défini par un en-tête en un corps. L'en-tête spécifie le nom du nœud, ainsi que ses entrées et ses sorties. Dans le cas de l'intégrateur, nous proposons l'en-tête suivant :

```
node Integr(e:real) returns (s:real)
```

Nous venons de spécifier que le nœud s'appellera `Integr`, qu'il aura une entrée `e` et une sortie `s`, les deux étant des flots de réels (d'où le typage `:real`).

Ensuite, nous pouvons spécifier les calculs effectués par ce nœud en lui ajoutant un corps :

```
node Integr(e:real) returns (s:real)
var p:real;
let
  s = e → (e + p);
  p = pre s;
tel
```

Exemple 2.1

Le corps du nœud est formé par un système d'équations qui a une solution unique en les variables de sortie (ici `s`) et les variables locales (`p`), solution paramétrée par les entrées (`e`). Nous verrons plus loin comment cette unicité est assurée.

Que fait le nœud `Integr` ? L'équation « `p = pre s` » indique que la valeur courante de `p` est égale à la valeur précédente de la sortie `s`. Mathématiquement, on pourrait écrire

$$p_i = s_{i-1}$$

pour tout $i > 0$. La valeur de `p` au premier instant (c'est-à-dire p_0) n'est pas définie. Un exemple de trace donnée par cette équation est :

`s`	s_0	s_1	s_2	s_3	...
`p`	?	s_0	s_1	s_2	...

Dans l'équation « `s = e` $\rightarrow$ `(e + p)` », l'opérateur « flèche » est utilisé pour spécifier d'une part la valeur au tout premier instant (à gauche de $\rightarrow$) et d'autre part le comportement aux instants suivants (à droite). Ainsi, la valeur initiale de `s` est égale à la valeur initiale de `e`, puis la valeur courante de `s` est obtenue en additionnant la valeur courante de `e` et celle de `p` :

$$s_0 = e_0 \qquad \text{et} \qquad s_i = e_i + p_i \text{ pour } i > 0$$

Ces deux équations décrivent donc bien le comportement voulu, comme le prouve un court raisonnement par récurrence :

- $s_0 = e_0 = \sum_{j=0}^{0} e_i$
- $s_{i+1} = p_{i+1} + e_{i+1} = s_i + e_{i+1} \overset{HR}{=} \left(\sum_{j=0}^{i} e_j \right) + e_{i+1} = \sum_{j=0}^{i+1} e_j$

Finalement, un exemple de trace du nœud `Integr` entier est

`e`	2	1	15	-3	...
`p`	?	2	3	18	...
`s`	2	3	18	15	...

Même sur cet exemple simple, nous pouvons retrouver les caractéristiques fondamentales de Lustre :

- Le nœud `Integr` décrit bien une fonction qui associe une suite de réels `s` à toute suite de réels `e`. En particulier, cette fonction n'a pas d'effet de bord[1].
- Les calculs sont spécifiés dans le corps du nœud par un système d'équations. L'ordre des équations n'a pas d'importance. De même, on aurait pu spécifier `s` par « `s=e`$\rightarrow$`(e+pre s)` » et obtenir le même résultat grâce au principe de substitution, qui s'applique de la même manière qu'en mathématiques.
- La suite `s` a exactement autant d'éléments que la suite `e` : un nouvel élément de `s` est « fabriqué » exactement quand un élément de `e` est « consommé ». L'entrée et la sortie sont donc bien *synchrones*.
- On imagine aisément que l'implémentation de ce nœud est triviale : les calculs peuvent être effectués rapidement, en ayant recours à une seule mémoire intermédiaire. Ainsi, notre intégrateur pourra être réactif.
- D'ailleurs, l'implémentation peut se faire au moyen d'un circuit électronique, comme celui de la figure 2.1.

Après cette introduction informelle, nous allons définir le langage Lustre plus précisément dans la partie suivante.

[1] Si l'absence d'effet de bord semble évidente en Lustre, il en va autrement dans d'autres langages fonctionnels : par exemple dans OCaml, une fonction peut avoir des effets de bord.

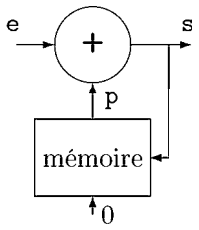


FIG. 2.1 – Circuit corespondant au programme de l'exemple 2.1

2.2 Définition formelle de Lustre

Dès le prochain chapitre, nous aurons besoin de nous appuyer sur la sémantique formelle du langage : c'est pourquoi nous allons la présenter en détail dans les lignes suivantes. Cependant, pour ménager le lecteur, nous allons procéder en plusieurs étapes. Une référence rapide du langage peut être consultée dans les annexes, page 129.

2.2.1 Les briques de base

À la base de tout programme Lustre, se trouvent des flots et des opérateurs qui permettent de les combiner entre eux. Nous avons rappelé dans l'introduction que les flots étaient des suites, finies ou infinies, de valeurs typées. Plus formellement, si D est un domaine de valeurs (un « type » informatique), alors on notera D^* l'ensemble des suites finies sur D, et D^ω celui des suites infinies. Un flot est alors élément de l'ensemble D^∞ où $D^\infty = D^* \cup D^\omega$.

Pour la suite, nous adopterons les conventions typographiques suivantes :

La syntaxe de Lustre	La sémantique	
	flots (éléments de D^∞)	valeurs (éléments de D)
`syntaxe`	flots	*valeurs*

Variables et constantes Les variables[2] sont des flots : ainsi, dans l'exemple 2.1, la variable `e`, de type `real`, dénote un flot `e` appartenant à l'ensemble $real^\infty$ où $real$ est l'ensemble des valeurs réelles représentables sur une machine donnée. Puisque `e` est une suite, nous pouvons désigner ses éléments avec la notation indexée : par exemple e_0, e_n.

Les constantes sont des flots particuliers, car tous les éléments de la suite associée ont la même valeur. Ainsi, la constante `1` dénote la suite $1, 1, 1, \ldots$ et si `c` est une constante, alors `c` représente la suite $c_0, c_0, c_0, \ldots$

Opérateurs combinatoires On peut combiner les flots au moyen de divers opérateurs. Les opérateurs les plus simples sont les fonctions arithmétiques et logiques usuelles, portées sur les flots par application « point-à-point ». Par exemple l'addition sur les entiers ($+ : int \times int \rightarrow int$) est portée sur les flots par

`x + y` dénote le flot `z` tel que $z_i = x_i + y_i$ pour tout i

[2]Nous appelons « variable » tout flot de base, qu'il s'agisse d'un paramètre d'entrée, de sortie ou de variable locale.

De manière équivalente, en utilisant la notation *valeur*.suite pour désigner la concaténation d'une valeur à une suite, nous pouvons définir l'addition de manière inductive par

$$x_0.\mathsf{x} + y_0.\mathsf{y} = (x_0 + y_0).(\mathsf{x} + \mathsf{y}) \qquad \text{et} \qquad \varepsilon + \varepsilon = \varepsilon$$

où ε dénote la suite vide.

L'addition entre deux flots d'entiers (+) est donc obtenue en appliquant l'addition d'entiers (+) entre des couples de valeurs « au même indice ». Un exemple :

`x`	2	1	−6	−3	...
`y`	7	2	3	18	...
`x+y`	9	3	−3	15	...

De la même manière, nous pouvons définir d'autres opérateurs combinatoires, dont certains figurent dans le tableau suivant :

syntaxe Lustre de l'opérateur	le flot associé, défini par une suite	le flot associé, défini de manière inductive
`x - y`	$(x_i - y_i)_i$	$x_0.\mathsf{x} - y_0.\mathsf{y} = (x_0 - y_0).\mathsf{x} - \mathsf{y}$
`x div y`	$(x_i \div y_i)_i$[3]	$x_0.\mathsf{x}$ div $y_0.\mathsf{y} = (x_0 \div y_0).\mathsf{x}$ div y
`x mod y`	$(x_i \mid y_i)_i$[4]	$x_0.\mathsf{x}$ mod $y_0.\mathsf{y} = (x_0 \mid y_0).\mathsf{x}$ mod y
`x <= y`	$(x_i \leq y_i)_i$	$x_0.\mathsf{x} \leq y_0.\mathsf{y} = (x_0 \leq y_0).\mathsf{x} \leq \mathsf{y}$
`x <> y`	$(x_i \neq y_i)_i$	$x_0.\mathsf{x} \neq y_0.\mathsf{y} = (x_0 \neq y_0).\mathsf{x} \neq \mathsf{y}$
`a and b`	$(a_i \wedge b_i)_i$	$x_0.\mathsf{x}$ and $y_0.\mathsf{y} = (x_0 \wedge y_0).\mathsf{x}$ and y
`not a`	$(\neg a_i)_i$	not $x_0.\mathsf{x} = (\neg x_0).$not x
cas de base pour tous ces opérateurs : ε op $\varepsilon = \varepsilon$		

La condition est gérée par l'opérateur `if then else`, qui s'applique à un flot booléen et deux flots d'un type `t`. C'est donc un opérateur ternaire : les deux branches (`then` et `else`) sont obligatoirement présentes. La signification de l'expression `if b then x else y` est

$$(si\ b_i\ alors\ x_i\ sinon\ y_i)_i \qquad \text{ou} \qquad \begin{cases} \text{if } t.\mathsf{b} \text{ then } x_0.\mathsf{x} \text{ else } y_0.\mathsf{y} &= x_0.\text{if b then x else y} \\ \text{if } f.\mathsf{b} \text{ then } x_0.\mathsf{x} \text{ else } y_0.\mathsf{y} &= y_0.\text{if b then x else y} \\ \text{if } \varepsilon \text{ then } \varepsilon \text{ else } \varepsilon &= \varepsilon \end{cases}$$

Avant de poursuivre, arrêtons-nous pour noter le fait suivant : un élément du flot-résultat d'une de ces opérations est défini si et seulement si il y a un élément correspondant dans chaque flot-argument de cette opération. Par exemple, l'élément numéro 100 du flot qui additionne x et y existera si x_{100} **et** y_{100} existent, et vaudra $x_{100} + y_{100}$. Ainsi, pour pouvoir appliquer un opérateur combinatoire à plusieurs flots, il faut que ceux-ci soient de la même longueur, qu'il soient *synchrones*. Par ailleurs, le flot-résultat de l'opération sera synchrone avec les arguments.

C'est à cause de ce synchronisme que nous avons donné comme cas de base ε op $\varepsilon = \varepsilon$, sans mentionner d'autres cas, en principe possibles, par ex. « $x_0.\mathsf{x}$ op ε » : en fait, cette expression n'est pas synchrone, donc elle n'a pas de sens en Lustre. Nous détaillerons cette notion, ainsi que la manière dont on s'assure qu'un programme est rigoureusement synchrone dans la section 2.2.4.

[3] division euclidienne
[4] reste de la division euclidienne

Opérateurs temporels Une autre catégorie d'opérateurs Lustre est formée par les fonctions spécifiques aux flots, qui permettent de « décaler les indices ». Nous avons déjà rencontré deux d'entre elles dans l'exemple 2.1 : `pre` et $\rightarrow$. Il existe un troisième opérateur temporel, `fby`. Pour définir ces opérateurs, nous aurons besoin de l'opérateur auxiliaire pre' qui n'a pas de syntaxe en Lustre, mais qui est nécessaire à la sémantique :

$$\begin{cases} \mathsf{pre}'(v, x_0.\mathsf{x}) &= v.\mathsf{pre}'(x_0, \mathsf{x}) \\ \mathsf{pre}'(v, \varepsilon) &= \varepsilon \end{cases}$$

Enfin, voici les définitions de `pre`, $\rightarrow$ et `fby` :

`pre x`	$(x_{i-1})_i$ où x_{-1} est une valeur indéfinie	$\mathsf{pre}\ \mathsf{x} = \mathsf{pre}'(?, \mathsf{x})$
`x -> y`	$(z_i)_i$ où $z_0 = x_0$ et $z_{i+1} = y_{i+1}$	$\begin{cases} x_0.\mathsf{x} \rightarrow y_0.\mathsf{y} = x_0.\mathsf{y} \\ \varepsilon \rightarrow \varepsilon = \varepsilon \end{cases}$
`x fby y`	$(z_i)_i$ où $z_0 = x_0$ et $z_{i+1} = y_i$	$\begin{cases} x_0.\mathsf{x}\ \mathsf{fby}\ \mathsf{y} = \mathsf{pre}'(x_0, \mathsf{y}) \\ \varepsilon\ \mathsf{fby}\ \mathsf{y} = \varepsilon \end{cases}$

L'opérateur `pre` représente une mémoire : sa valeur actuelle est la valeur précédente de son argument (`pre` comme « previous »). La toute première valeur rendue par `pre` n'est pas définie, d'où l'importance de l'opérateur $\rightarrow$ qui permet d'initialiser un flot avec une valeur donnée. Enfin, `fby` (« followed by ») est une combinaison de ces deux opérateurs.

Nous venons de voir l'essentiel des éléments qui permettent de construire un programme Lustre. Même si nous n'avons donné que des définitions faisant intervenir des variables, les opérateurs s'appliquent aux expressions arbitrairement compliquées. Ainsi, par exemple « `if x-y<0 then y-x else x-y` » est une expression qui calcule la valeur absolue de `x-y`.

Nous allons à présent passer à l'assemblage des briques de base en des unités fonctionnelles.

2.2.2 Les nœuds

L'unité de base d'un programme Lustre est un nœud (« node » en anglais). Comme nous allons voir, chaque nœud décrit une fonction (au sens mathématique du terme), en spécifiant un système d'équations qui la définit de manière unique. Les expressions que nous avons vues à la section précédente servent à écrire les équations à l'intérieur du nœud ; elles n'existent pas en tant que quantités indépendantes.

L'en-tête De manière générale, un nœud se présente comme ceci :

```
node <nom du nœud> (<liste des paramètres d'entrée>)
returns (<liste des paramètres de sortie>);
var <liste des variables locales>
let
  <liste d'équations>
tel;
```

L'en-tête du nœud déclare en fait la signature de la fonction réalisée par ce nœud, car tous les paramètres sont typés. Ainsi, dans

```
node valeurAbsolueDiff (x:int ; y:int)
returns (va:int)
let
  va = if x>y then x-y else y-x;
tel;
```

les deux premières lignes annoncent une fonction *valeurAbsolueDiff* : $int^\infty \times int^\infty \rightarrow int^\infty$ où int est le type entier. Comme toute fonction, elle pourra être appliquée à des arguments effectifs, si bien qu'un autre nœud pourra par exemple faire l'appel `valeurAbsolueDiff(x,0)` pour calculer la valeur absolue de `x`. Ainsi, l'appel de nœud s'ajoute à la liste des expressions déjà rencontrées.

Le corps En ce qui concerne le corps du nœud, il est formé par une liste d'équations. Une équation est simplement une égalité de la forme

équation : <*variable*> = <*expression*>;

Les expressions utilisées dans les équations portent sur les flots définis localement par le nœud (les entrées, les sorties et les éventuelles variables locales) et sur les constantes globales. La liste des équations forme donc un système qui peut être récursif : une variable peut dépendre d'elle-même, dans une certaine mesure.

En effet, on impose un certain nombre de contraintes au système équationnel :

- Il doit y avoir exactement une équation par sortie et par variable locale pour la définir : c'est pourquoi l'exemple de la `valeurAbsolueDiff` contient la ligne `va = ...`
- Les variables peuvent dépendre les unes des autres (ou d'elles-mêmes), à condition de *ne pas créer de dépendance circulaire instantanée*. Pour éviter cela, chaque cycle de dépendance entre variables doit contenir au moins un opérateur `pre`.

Ces deux conditions sont suffisantes pour garantir l'existence et l'unicité d'une solution du système d'équations [22]. Nous pouvons donner une intuition des raisons de ce résultat en faisant appel aux circuits électroniques : nous avons vu que les opérateurs en Lustre agissent comme des composants de base des circuits (additionneur, multiplicateur, registre etc.). Les variables sont en fait les fils qui relient les divers opérateurs en un réseau. Dans cette optique, la première condition assure que le fil `va` sera relié à la sortie d'un seul opérateur. La deuxième assure que si la valeur sur le fil `va` dépend de `va`, la dépendance se fait par rapport à une valeur passée, stockée dans une mémoire. En fin de compte, les deux conditions empêchent la formation de courts-circuits.

Sémantique du nœud Nous avons déjà annoncé qu'un nœud réalisait une fonction, plus particulièrement la fonction qui associe aux entrées la solution du système d'équations du nœud. Nous allons devoir préciser cette définition et pour cela, nous allons approfondir le contexte théorique.

Les flots, éléments de base des programmes Lustre, sont des éléments du domaine D^∞. Or cet ensemble peut être muni de l'élément neutre ε (la suite vide), de l'opération de « concaténation en tête ». et de l'ordre préfixe $\preceq$ tel que

$$\forall (x,y) \in D^\infty.\ x \preceq y \ \Leftrightarrow\ \exists z \in D^\infty.\ x \bullet z = y \qquad \text{où } \bullet \text{ est la concaténation de suites}$$

Alors $\langle D^\infty, \varepsilon, ., \preceq \rangle$ est un ordre partiel complet. Cela signifie en particulier que toute suite croissante d'éléments (ou « chaîne ») $C = \{x_0 \preceq x_1 \ldots \preceq x_n \preceq \ldots\}$ a une borne supérieure unique (le plus petit majorant), notée $\sup C$.

Dans un ordre partiel complet, on dit qu'une fonction f est *continue* si pour toute chaîne croissante C, $f(C)$ est une chaîne croissante et $f(\sup C) = \sup f(C)$. Les fonctions continues sont importantes à cause du théorème suivant :

Théorème de Kleene Dans un ordre partiel complet, si f est une fonction continue, alors l'équation $x = f(x)$ possède une plus petite solution μf, égale à $\mu f = \sup_{n\geq 0}\{f^n(\varepsilon)\}$. De plus, si $\mu_x f(x,y)$ est la plus petite solution de l'équation $x = f(x,y)$, alors la fonction $y \mapsto \mu_x f(x,y)$ est continue.

Or les opérateurs Lustre que nous avons vus sont tous continus [17], donc le théorème de Kleene s'applique :

Les expressions d'un nœud Lustre forment une fonction

$$f : D^\infty \times D^\infty \times D^\infty \to D^\infty \times D^\infty$$

telle que le système d'équations mutuellement récursives s'écrit

$$(s, v) = f(e, s, v)$$

où e sont les entrées, s les sorties et v les variables locales (ainsi, en toute généralité, e, s, et v sont des vecteurs de flots). Le nœud entier réalise alors la fonction

$$e \mapsto \mu_{s,v} f(e, s, v)$$

où $\mu_{s,v} f(e, s, v)$ est le plus petit point fixe de f par rapport à ses composantes s et v.

Sémantique d'un nœud Lustre

Ainsi, le nœud `Integr` de l'exemple 2.1 (page 18) définit la fonction

$$\begin{array}{lrcl} \texttt{Integr}: & real^\infty & \to & real^\infty \\ & e & \mapsto & \mu_s.\ (e\ \texttt{->}\ e + \texttt{pre}(s)) \end{array}$$

ce qui s'écrit, en notant hd et tl les destructeurs de suites (« tête » et « queue »)

$$\begin{array}{lrcl} \texttt{Integr}: & real^\infty & \to & real^\infty \\ & e & \mapsto & \mu_s.\ (hd(e).(tl(e) + s)) \end{array}$$

Il est intéressant de noter qu'en plus d'une définition formelle de la sémantique du nœud, le théorème de Kleene nous fournit une manière effective de calculer le point fixe $\mu_{s,v} f(e, s, v)$, selon la formule $\mu_{s,v} f(e, s, v) = \sup_n\{f^n(e, \varepsilon, \varepsilon)\}$. Ainsi, la solution consiste à calculer de manière itérative

$$\begin{array}{l} \cdot\ f(\varepsilon, \varepsilon, \varepsilon) \\ \cdot\ f(e_0, f(\varepsilon, \varepsilon, \varepsilon)) \\ \cdot\ f(e_0.e_1, f(e_0, f(\varepsilon, \varepsilon, \varepsilon))) \\ \ldots \end{array}$$

ce qui produit le flot de sortie. L'application pratique de cette propriété est le principe de compilation des programmes Lustre en « boucle simple » :

```
initialisations
tant que vrai faire
  - lire les entrées
  - calculer les sorties
  - mettre à jour les mémoires
fin tant que
```

On peut remarquer que les `mémoires` dont il est question sur le schéma ci-dessus *représentent* les variables locales et les sorties du programme, mais *ne sont pas identiques* à ces flots. En effet, les `mémoires` du programme compilé sont des entités scalaires, tandis que les variables du programme source sont des suites.

La compilation de suites en `mémoires` ponctuelles est rendue possible par le synchronisme de Lustre[5] qui assure que la valeur courante d'un flot dépend d'un nombre fini (et connu) de valeurs passées. Ce sont ces quelques valeurs passées nécessaires au calcul de la valeur courante qui sont stockées dans les `mémoires` du programme compilé. Nous appelerons ces `mémoires` également *variables d'état* du programme, car leurs valeurs définissent de manière unique l'état du programme.

À titre d'exemple, le nœud `Integr` possède une variable d'état, qui est la valeur courante de `p` (la valeur précédente de `s`). On peut noter que sur le circuit électronique équivalent (figure 2.1), cette variable d'état correspond à une mémoire (un registre). En fait, on peut généraliser cette remarque : une variable d'état correspond à une mémoire dans le circuit équivalent.

2.2.3 Les programmes

Outre les nœuds, un programme Lustre peut comporter d'autres déclarations au niveau global : des définitions de types et de constantes, ainsi que des déclarations de fonctions externes.

Les types Lustre connaît trois types de base : les entiers `int`, les réels `real` et les booléens `bool`. L'utilisateur peut importer d'autres types, en déclarant leur nom

importation de type : `type` <*nom*>`;`

Les types de base et les types importés peuvent servir à construire d'autres types, comme dans cet exemple :

```
type registre8 = bool^8;          type tableau de 8 booléens
type reponse   = { bool, int };   type enregistrement
```

Ensuite, on pourra utiliser ces types par exemple dans la déclaration d'un nœud :

```
node bin2int (bin : registre8)
returns (i : reponse);
...
```

Les constantes Il est possible de déclarer des constantes nommées, de portée globale, soit en donnant leur type (mais pas la valeur), soit leur valeur.

importation de constante : `const` <*nom*> `:` <*type*>`;`
définition de constante : `const` <*nom*> `=` <*valeur*>`;`

On peut utiliser les constantes dans les expressions, mais aussi dans la définition des types :

```
const dim = 4 ;
const listeVide : Liste ;
type Liste ;
type espace = real^dim ;
type hyperEspace = real^(dim-1) ;
```

[5]et plus particulièrement par le calcul d'horloges abordé en section 2.2.4

Les fonctions externes Les programmes Lustre peuvent faire appel à des fonctions externes, c'est-à-dire implémentées dans un autre langage. Pour cela, il suffit de déclarer le profil d'une telle fonction :

déclaration de fonction : `function` <*nom*> `(`<*entrées*>`) returns (`<*sorties*>`);`

Par exemple, on peut imaginer un traitement externe des listes d'entiers

```
function concat (i:int ; l:Liste) returns (l2:Liste);
```

Ce traitement pourrait être appelé dans un nœud, comme toute autre fonction

```
...
queue = concat(0,listeVide) ->
        concat(x, pre queue) ;
...
```

Pour résumer, un programme Lustre est une suite de déclarations, dont l'ordre n'est pas important. Ces déclarations décrivent des types, des constantes, des fonctions externes et des fonctions effectivement implémentées dans le programme (les nœuds). Le corps de ces derniers est formé par un système d'équations, dont la solution (unique) est le résultat de la fonction.

2.2.4 Les horloges

Lustre, tel que nous l'avons dépeint jusqu'ici, permet d'écrire des programmes qui se comportent comme des circuits électroniques, certes, mais des circuits où toutes les valeurs évoluent à la même cadence. Or, dans un système électronique complexe, il existe souvent des sous-parties qui effectuent des calculs à leur propre rythme et ne communiquent avec les autres sous-systèmes que de temps en temps.

Les horloges Pour tenir compte de cette réalité, Lustre propose un mécanisme spécial, celui des *horloges*. Intuitivement, deux signaux (flots) qui évoluent à la même vitesse seront « sur la même horloge », alors que d'autres seront « sur des horloges différentes ».

En pratique, n'importe quel flot booléen b peut agir comme horloge d'un autre flot x. « x est sur b » signifie alors que

- si la valeur courante de b est *vrai*, alors le flot x est présent, c'est-à-dire qu'il a une valeur définie,
- lorsque b passe à *faux*, x n'existe pas : demander sa valeur à ce moment-là n'a tout simplement pas de sens.

Pour illustrer ce point, prenons l'exemple suivant où on suppose que le flot x est sur l'horloge b :

b	t	t	f	t	f	f	...
x	x_0	x_1		x_2			...

On peut constater qu'à l'instant où b devient *faux*, x n'existe pas : le trou entre les valeurs x_1 et x_2 est présent uniquement pour des raisons de lisibilité du diagramme, mais ne représente aucune valeur.

Pour définir qu'un flot est sur une horloge donnée, on utilise en Lustre la formulation suivante, accolée à la définition du type du flot :

déclaration d'un flot sur une horloge : `when` <*horloge*>

Ainsi, un nœud qui prend en entrée un flot booléen ck et un flot entier i sur l'horloge ck et qui rend en sortie le flot i sans changement, serait déclaré comme

```
node id (ck:bool ; i:int when ck)
returns (o:int when ck) ;
let
  o = i ;
tel ;
```

Exemple 2.2 Déclaration de flots sur horloge

On remarque que le flot ck n'est pas déclaré comme étant sur une horloge. En fait, il existe une horloge particulière, la constante true, qui est l'horloge par défaut : comme true vaut *vrai* à tout instant, les flots sur true sont toujours présents. On aurait donc pu déclarer « `ck:bool when true` » dans l'exemple précédent.

Les opérateurs when et current Nous disposons donc de flots sur différentes horloges. Lorsque deux flots sont sur la même horloge, on peut continuer à les combiner avec les opérateurs déjà rencontrés. Le problème qui se pose est celui du calcul sur des flots qui sont sur des horloges différentes, car ces flots ne sont pas nécessairement présents en même temps et n'ont pas nécessairement le même nombre d'éléments. Bref, ces flots *ne sont pas synchrones*, et c'est pourquoi Lustre interdit de les combiner directement.

Pourtant, il est possible de faire communiquer des flots sur différentes horloges, en les ramenant soit sur une horloge plus rapide, soit sur une horloge plus lente, grâce aux opérateurs `when`[6] et `current`.

when L'opérateur `when` permet de ramener un flot rapide sur une horloge plus lente, en l'échantillonnant par rapport à cette horloge. Sa syntaxe est

échantillonnage : <*flot*> `when` <*horloge*>

Lorsque l'horloge est *vraie*, la valeur du flot « <*flot*> `when` <*horloge*> » est égale à la valeur du <*flot*> échantillonné ; lorsque l'horloge est fausse, le flot-résultat n'existe pas. Ainsi, le résultat est bien sur l'<*horloge*> spécifiée, comme dans l'exemple suivant :

x	x_0	x_1	x_2	x_3	x_4	x_5	...
b	t	t	f	f	t	f	...
x when b	x_0	x_1			x_4		...

La sémantique précise de `when` est

x_0.x	when	t.b	=	x_0.(x when b)
x_0.x	when	f.b	=	x when b
ε	when	ε	=	ε

current L'opérateur `current` effectue l'opération inverse de `when`, c'est-à-dire qu'il ramène un flot lent sur une horloge plus rapide, en comblant les trous avec la dernière valeur connue. Nous allons illustrer son comportement en prolongeant l'exemple précédent :

x	x_0	x_1	x_2	x_3	x_4	x_5	...
b	t	t	f	f	t	f	...
x when b	x_0	x_1			x_4		...
current(x when b)	x_0	x_1	x_1	x_1	x_4	x_4	...

[6] Le mot-clé `when` est surchargé : on l'utilise aussi bien pour les déclarations de flots que pour les calculs.

Pour définir la sémantique de `current`, nous allons mettre en évidence, outre le flot lent y, son horloge h et sa dernière valeur connue d :

$$\begin{array}{lcl} \mathsf{current}(d, t.\mathsf{h}, y_0.\mathsf{y}) & = & y_0.\mathsf{current}(y_0, \mathsf{h}, \mathsf{y}) \\ \mathsf{current}(d, f.\mathsf{h}, \mathsf{y}) & = & d.\mathsf{current}(d, \mathsf{h}, \mathsf{y}) \\ \mathsf{current}(d, \varepsilon, \varepsilon) & = & \varepsilon \end{array}$$

Alors, la sémantique de l'opérateur `current` unaire, qui est utilisé en Lustre, est la suivante :

$$\mathsf{current}(\mathsf{y}) = \mathsf{current}(?^{7}, \mathrm{horloge\_de}(\mathsf{y}), \mathsf{y})$$

Calcul d'horloges Nous venons de voir que pour calculer current(y), on a besoin de connaître l'horloge de y. De manière plus générale, il faut connaître les horloges de tous les flots d'un programme Lustre pour s'assurer que tous les calculs portent sur des flots synchrones. En fait, cette vérification est effectuée à la compilation, lors de l'étape appelée *calcul d'horloges*.

Dans la section 2.2.2, nous avons vu que tout flot de base d'un nœud (entrée, sortie, variable locale) doit être typé. De même, si l'horloge du flot est différente de true, elle doit être déclarée et doit figurer dans la liste des arguments du nœud. Chaque flot en Lustre possède donc un double typage : son type classique[8] est celui des valeurs qu'il porte (p.ex. `int`) ; son *type temporel* est l'horloge associée.

De ce point de vue, le calcul d'horloges est donc une partie de la vérification des types. Les règles précises du typage de Lustre figurent dans l'annexe A ; contentons-nous de les résumer ici en disant que le typage empêche de combiner deux flots sur deux horloges différentes de la même manière qu'il empêche de combiner des booléens avec les entiers dans une addition.

Le calcul d'horloges est correct, dans le sens où un programme déclaré synchrone l'est effectivement. Il n'est pas complet, car l'équivalence des horloges utilisée est purement syntaxique, si bien que certains programmes synchrones peuvent être rejetés.

2.2.5 Conclusion

Au début de ce chapitre, nous avons cité un certain nombre de propriétés qui caractérisent le langage Lustre : les aspects fonctionnel, flots-de-données et synchrone. Nous avons vu que ces caractéristiques permettaient de spécifier facilement des systèmes réactifs apparentés aux circuits électroniques.

À la lumière de la suite de ce chapitre, nous pouvons maintenant ajouter deux autres caractéristiques importantes, le typage fort et la sémantique formelle. C'est notamment grâce à la sémantique bien définie que nous pourrons nous intéresser à la preuve de propriétés des programmes Lustre. Mais avant d'exposer nos propres travaux dans ce domaine, nous allons faire l'état d'un travail précurseur au chapitre suivant.

[7] valeur indéfinie

[8] Nous l'appellerons le *type scalaire*.

3 – Méthode de preuve inductive de Cécile Dumas

Pour présenter la méthode de preuve de Cécile Dumas, nous procédons en plusieurs étapes : après avoir défini le problème auquel nous nous attaquons, nous énumérons les règles de base utilisées dans la stratégie de preuve. Vient ensuite un exemple d'application, suivi du reste des règles, celles d'usage plus restreint. Pour terminer, nous évoquons des limitations de la méthode et une tentative incorrecte pour y remédier.

Cette thèse se situe par certains aspects dans le prolongement direct de la thèse de Cécile Dumas [17]. Nous consacrons ce chapitre à la présentation du point le plus important pour nos travaux, la méthode de preuve inductive de propriétés invariantes. Cécile Dumas décrit cette méthode au chapitre 8 de son mémoire et la résume également dans un article séparé [16].

3.1 Position du problème

Les propriétés invariantes, ou propriétés de sûreté, décrivent le fait qu'un système évolue dans un ensemble d'états donné, sans jamais en sortir. Les propriétés critiques imposées aux programmes écrits en Lustre/Scade sont souvent de cette forme : par exemple « deux trains consécutifs sur une voie seront toujours séparés par une distance minimum donnée » ou « la température à l'intérieur du réacteur ne dépassera jamais un certain seuil ». Il est donc important de pouvoir prouver ces propriétés.

La méthode de preuve de Cécile Dumas est une méthode interactive : on demande à l'utilisateur de fournir la preuve. Cependant, la propriété de départ, exprimée en termes du comportement global du système, est d'abord réduite à des sous-problèmes ponctuels. Ainsi, l'utilisateur n'aura pas à prouver le problème sous sa forme initiale, mais sous une forme plus abordable. C'est donc un travail de prétraitement que Cécile Dumas propose.

Nous allons maintenant définir le problème plus formellement, en reprenant les notations de la section 2.2. Si D est un domaine de valeurs et P un prédicat sur ce domaine ($P \subseteq D$), alors l'invariant associé à P est le prédicat sur flots $\Box P \subseteq D^\infty$ défini par :

$$\begin{array}{lcl} \Box P(\varepsilon) & = & \mathit{true} \\ \Box P(x.\mathsf{X}) & = & P(x) \wedge \Box P(\mathsf{X}) \end{array}$$

Il est notable que dans le cas de programmes Lustre, tous les invariants peuvent être exprimés sous forme de nœuds Lustre spéciaux, les observateurs synchrones [21]. Par exemple, l'invariant stipulant que la valeur d'une variable entière `n` est toujours supérieure à `10` peut être écrit sous la forme du nœud suivant :

```
node obs (n:int) returns (b:bool);
let
  b = (n >= 10) ;
tel;
```

Le nœud `obs` prend en entrée la variable observée `n` et rend un flot booléen, dont les valeurs successives expriment la valeur de vérité de la propriété recherchée : prouver que `n >= 10` est bien un invariant, c'est prouver que la sortie `b` est toujours vraie ($\Box b$).

Ainsi, Lustre peut être utilisé à la fois comme langage de programmation et comme cadre formel pour exprimer des propriétés sur les programmes. Mieux, ces propriétés sont en fait des programmes comme les autres.

Étant donné un nœud Lustre N, le problème auquel on s'intéresse est de prouver que P est un invariant de N. Formellement, si f est la fonction décrite par le corps de N, on cherchera à prouver

$$\Box P(\mu_x.x = f(x)) \tag{3.1}$$

En effet, nous avons vu au chapitre précédent que le flot calculé par le nœud N est $\mu_x.x = f(x)$.

3.2 Méthode de preuve

La méthode de preuve proposée par Cécile Dumas se résume à l'application de quelques règles. Nous allons les présenter dans l'ordre dans lequel elles sont normalement utilisées.

Règle de continuité Le premier constat que l'on peut faire en regardant le but (3.1) est qu'il porte sur une suite potentiellement infinie ($\mu_x.x = f(x) \ \subseteq D^\infty$). La règle suivante, dite de continuité, permet de se ramener aux suites finies :

$$(\text{cont})\frac{\Box P(\varepsilon) \qquad \forall y \in D^*.\ \Box P(y) \Rightarrow \Box P(f(y))}{\Box P(\mu_x.x = f(x))} \tag{3.2}$$

La justification de cette règle se trouve dans le théorème de Kleene (cf. section 2.2.2), qui stipule que $\mu_x.\ x = f(x)$ est égal à $\sup_{n\geq 0}\{f^n(\varepsilon)\}$. Ainsi, le but (3.1) devient $\Box P(\sup_{n\geq 0}\{f^n(\varepsilon)\})$. Or P est décrite par un programme Lustre, ce qui signifie que P est continue, si bien que

$$\begin{array}{lcll} (3.1) & \equiv & \Box P(\mu_x.x = f(x)) & \\ & \equiv & \Box P(\sup\limits_{n\geq 0}\{f^n(\varepsilon)\}) & \text{théorème de Kleene} \\ & \equiv & \sup\limits_{n\geq 0}\Box P(\{f^n(\varepsilon)\}) & \text{continuité de } \Box P \end{array}$$

La règle (3.2) est donc une règle *suffisante* pour établir ce dernier but : prouver que d'une part $\Box P(\varepsilon)$ (ce qui est trivialement vrai) et que d'autre part $\forall y \in D^*.(\Box P(y) \Rightarrow \Box P(f(y)))$, c'est prouver par induction que $\forall n \in \mathbb{N}.\Box P(f^n(\varepsilon))$, d'où par *continuité*, $\sup_{n\geq 0}\Box P(\{f^n(\varepsilon)\})$.

La règle de continuité permet donc de transformer le but (3.1) en un nouveau but :

$$\forall y \in D^*.\ \Box P(y) \Rightarrow \Box P(f(y)) \tag{3.3}$$

Règle d'induction Le but (3.3) porte sur des suites finies. On peut donc lui appliquer la règle d'induction structurelle

$$(\text{ind})\frac{Q(\varepsilon) \qquad \forall z \in D^+.\ Q(tl\ z) \Rightarrow Q(z)}{\forall y \in D^*.\ Q(y)} \qquad \text{où } tl \text{ est l'opération « queue de liste »} \tag{3.4}$$

En posant $Q(v) = \Box P(v) \Rightarrow \Box P(f(v))$ et en appliquant la règle d'induction à (3.3), on obtient donc deux nouveaux buts :

- $\Box P(\varepsilon) \Rightarrow \Box P(f(\varepsilon))$ (3.5)
- $\forall z \in D^+.\ (\Box P(tl\ z) \Rightarrow \Box P(f(tl\ z))) \Rightarrow (\Box P(z) \Rightarrow \Box P(f(z)))$ (3.6)

Le but 3.5 se simplifie grâce à $\Box P(\varepsilon) = true$ en

$$\frac{}{\Box P(f(\varepsilon))}$$

(OPε)

Cette propriété constitue la première obligation de preuve (OP) donnée à l'utilisateur.

Après avoir effectué une skolémisation, on peut réécrire le but (3.6) sous une forme équivalente

$$\frac{\Box P(tl\ z) \Rightarrow \Box P(f(tl\ z)) \qquad \Box P(z)}{\Box P(f(z))}$$

Or la suite finie z se décompose en $z = (hd\ z).(tl\ z)$, donc on peut appliquer l'égalité $\Box P((hd\ z).(tl\ z)) = P(hd\ x) \wedge \Box P(tl\ z)$, simplifier et obtenir encore une autre forme :

$$\frac{\Box P(f(tl\ z)) \qquad \Box P(z)}{\Box P(f(z))} \tag{3.7}$$

Règles d'avancement La variable z du but (3.7) est une suite finie d'éléments de D. On peut appliquer aux suites finies les règles d'avancement suivantes :

$$(\text{cons}_1)\frac{}{\Box P(\varepsilon)} \qquad (\text{cons}_2)\frac{x \qquad \Box P(X)}{\Box P(x.X)} \qquad (\text{hd})\frac{\Box P(x.X)}{x} \qquad (\text{tl})\frac{\Box P(x.X)}{\Box P(X)}$$

Les deux premières règles permettent de prouver des buts faisant intervenir des suites, tandis que les deux autres sont utilisées pour décomposer les hypothèses.

Comment utiliser les règles d'avancement en pratique ? Le but général est de décomposer la conclusion du séquent (3.7), $\Box P(f(z))$, par la règle (cons_2). Il faut répéter la décomposition un nombre suffisant de fois pour pouvoir utiliser l'hypothèse de récurrence $\Box P(f(tl\ z))$. Pendant ce processus, on crée des obligations de preuve correspondant aux cas scalaires dans la décomposition.

On commence par poser $z = z_0.Z_1$. On applique la règle (hd) à la deuxième hypothèse de (3.7) et on obtient

$$\frac{\Box P(f(Z_1)) \qquad P(z_0) \qquad \Box P(Z_1)}{\Box P(f(z_0.Z_1))}$$

Supposons maintenant que la fonction f soit purement combinatoire : sa sortie ne dépend que de la valeur présente de l'entrée, si bien que $f(z_0.Z_1) = f(z_0).f(Z_1)$. Si tel est le cas, en appliquant la règle (cons_2), on arrive à

$$\frac{\Box P(f(Z_1)) \qquad P(z_0) \qquad \Box P(Z_1)}{P(f(z_0)) \wedge \Box P(f(Z_1))}$$

Ce séquent se décompose en deux : une nouvelle obligation de preuve à la charge de l'utilisateur et un séquent trivialement vrai par l'application de l'hypothèse d'induction.

$$\frac{P(z_0)}{P(f(z_0))}\ (\mathrm{OP}_0) \qquad \text{et} \qquad \frac{\Box P(f(Z_1)) \quad P(z_0) \quad \Box P(Z_1)}{\Box P(f(Z_1))}$$

Dans le cas d'une fonction f combinatoire, le processus s'arrête donc ici : l'utilisateur devra prouver $(\mathrm{OP}_\varepsilon)$ et (OP_0). Si ces deux propriétés sont vraies, alors la propriété de départ (3.1) l'est aussi.

Que se passe-t-il lorsque f n'est pas une fonction combinatoire ? Le premier pas reste le même : comme f est une fonction Lustre, la valeur de chaque sortie ne dépend, de manière générale, que du passé et du présent des entrées, si bien que $f(z_0.Z_1) = f(z_0).S$ où $S = tl(f(z_0.Z_1))$. Ce cas-là va donc aussi produire l'obligation de preuve (OP_0), mais cette fois-ci, le séquent restant ne permettra pas de conclure immédiatement :

$$\frac{\Box P(f(Z_1)) \quad P(z_0) \quad \Box P(Z_1)}{\Box P(S)}$$

Il sera nécessaire d'appliquer l'enchaînement des règles (hd) et (cons_2) plusieurs fois, créant des obligations de preuve (OP_1), (OP_2) etc., avant d'être en mesure de conclure par induction.

Un point important à noter est l'arrêt « garanti » de cet algorithme. En effet, le caractère synchrone de Lustre, et donc de P et de f, permet de prouver la terminaison du processus de preuve. Nous n'allons pas donner la justification technique de cette propriété (cf. [17]), mais uniquement un aperçu intuitif : f est une fonction Lustre et comme telle, elle ne contient qu'un nombre borné de mémoires, si bien que toute sortie n'est calculée qu'en fonction des n dernières valeurs de l'entrée. Le processus de preuve s'arrêtera alors après au plus $n+1$ étapes, car après les n cas d'initialisation, on retrouvera le cas général de l'induction.

Pour rendre les idées plus claires, nous proposons au lecteur un exemple.

3.3 Exemple de preuve

Énoncé Soit le nœud suivant[1] :

```
node naturals(rien:int) returns (n:int);
let
  n = 0 -> ((pre n) + 1) ;
tel;
```

Exemple 3.1

La sortie de `naturals` vaut 0 au premier instant et augmente de 1 à chaque instant suivant. En fait, ce nœud calcule l'ensemble des entiers naturels. On se propose de montrer que la sortie est toujours positive :

$$\Box(\mu_n.\ (n = 0.(n+1)) \ \geq 0)$$

Induction continue L'application de la règle de continuité donne le séquent suivant :

$$(\mathrm{cont})\frac{\forall y \in \mathbb{N}^*.\ \Box(y \geq 0) \Rightarrow \Box(0.(y+1) \geq 0)}{\Box(\mu_n.\ n = 0.(n+1) \ \geq 0)}$$

[1] L'entrée `rien` ne sert pas dans les calculs du nœud, mais la syntaxe de Lustre nous oblige à en déclarer au moins une.

Induction structurelle Le nouveau but à prouver est donc $\forall y \in \mathbb{N}^*.\ \Box(y \geq 0) \Rightarrow \Box(0.(y+1) \geq 0)$. On continue en lui appliquant le règle d'induction :

$$\text{(cont)}\ \dfrac{\text{(ind)}\ \dfrac{\bullet\ \Box(\varepsilon \geq 0) \Rightarrow \Box(0.(\varepsilon+1) \geq 0) \quad \bullet\ \forall z \in \mathbb{N}^+.(\Box(tl(z) \geq 0) \Rightarrow \Box(0.(tl(z)+1) \geq 0)) \Rightarrow (\Box(z \geq 0) \Rightarrow \Box(0.(z+1) \geq 0))}{\forall y \in \mathbb{N}^*.\ \Box(y \geq 0) \Rightarrow \Box(0.(y+1) \geq 0)}}{\Box(\mu_n.\ n = 0.(n+1)\ \geq 0)}$$

(a) Cas de base Le premier sous-but obtenu, $\Box(\varepsilon \geq 0) \Rightarrow \Box(0.(\varepsilon+1) \geq 0)$, se simplifie en $true \Rightarrow \Box(0.\varepsilon \geq 0)$, puis en $\Box(0.\varepsilon \geq 0)$. En appliquant un déroulement sur cette forme, on obtient l'obligation de preuve (OP_ε), qui sera à la charge de l'utilisateur, et un séquent trivialement vrai :

$$\boxed{\dfrac{}{0 \geq 0}}\ (\text{OP}_\varepsilon) \qquad \text{et} \qquad \Box\ \varepsilon$$

(b) Cas inductif Après skolémisation, le deuxième sous-but devient :

$$\dfrac{\Box(tl(z) \geq 0) \Rightarrow \Box(0.(tl(z)+1) \geq 0) \qquad \Box(z \geq 0)}{\Box(0.(z+1) \geq 0)}$$

En décomposant z selon $z = (hd\ z).(tl\ z)$ et en appliquant les règles d'avancement (hd) et (tl) sur l'hypothèse $\Box(z \geq 0)$, on obtient :

$$\dfrac{\Box(tl(z) \geq 0) \Rightarrow \Box(0.(tl(z)+1) \geq 0) \qquad hd(z) \geq 0 \qquad \Box(tl(z) \geq 0)}{\Box(0.(z+1) \geq 0)}$$

et donc

$$\dfrac{\Box(0.(tl(z)+1) \geq 0) \qquad hd(z) \geq 0 \qquad \Box(tl(z) \geq 0)}{\Box(0.(z+1) \geq 0)}$$

Avancement En faisant avancer la conclusion, le séquent devient :

$$\dfrac{\Box(0.(tl(z)+1) \geq 0) \qquad hd(z) \geq 0 \qquad \Box(tl(z) \geq 0)}{(hd(0.(z+1)) \geq 0) \wedge \Box(tl(0.(z+1)) \geq 0)}$$

ce qui se simplifie en

$$\dfrac{\Box(0.(tl(z)+1) \geq 0) \qquad hd(z) \geq 0 \qquad \Box(tl(z) \geq 0)}{(0 \geq 0) \wedge \Box((z+1) \geq 0)}$$

Après un avancement des deux antécédents de type « suite », on arrive à

$$\dfrac{0 \geq 0 \qquad \Box(tl(z)+1 \geq 0) \qquad hd(z) \geq 0 \qquad hd(tl(z)) \geq 0 \qquad \Box(tl^2(z) \geq 0)}{(0 \geq 0) \wedge \Box((z+1) \geq 0)}$$

On sépare le séquent en deux, chaque nouveau but prouvant l'une des branches de la conjonction : on crée ainsi la deuxième obligation de preuve (OP_0) et un nouveau séquent :

$$\boxed{\dfrac{0 \geq 0 \qquad hd(z) \geq 0 \qquad hd(tl(z)) \geq 0}{0 \geq 0}}\ (\text{OP}_0)$$

$$\frac{0 \geq 0 \qquad \Box(tl(z)+1 \geq 0) \qquad hd(z) \geq 0 \qquad hd(tl(z)) \geq 0 \qquad \Box(tl^2(z) \geq 0)}{\Box((z+1) \geq 0)}$$

Dans cet exemple, la fonction de sortie n'est pas combinatoire : la valeur de sortie actuelle dépend de la sortie précédente. On retrouve ce fait en constatant que l'hypothèse d'induction ne peut pas être appliquée à ce stade et qu'il faut donc continuer de faire avancer le séquent.

Comme précédemment, nous faisons avancer le conséquent.

$$\frac{0 \geq 0 \qquad \Box(tl(z)+1 \geq 0) \qquad hd(z) \geq 0 \qquad hd(tl(z)) \geq 0 \qquad \Box(tl^2(z) \geq 0)}{(hd(z)+1 \geq 0) \wedge \Box(tl(z)+1 \geq 0)}$$

Ensuite, après avoir fait avancer certaines[2] hypothèses, on sépare à nouveau le séquent en deux. Comme résultat, une nouvelle obligation de preuve (OP_1) et un nouveau séquent :

$$\boxed{\frac{0 \geq 0 \qquad hd(z) \geq 0 \qquad hd(tl(z)) \geq 0 \qquad hd(tl^2(z)) \geq 0}{hd(z)+1 \geq 0}}$$

(OP_1)

$$\frac{0 \geq 0 \qquad \Box(tl(z)+1 \geq 0) \qquad hd(z) \geq 0 \qquad hd(tl(z)) \geq 0 \qquad \Box(tl^3(z) \geq 0) \qquad hd(tl^2(z) \geq 0}{\Box(tl(z)+1 \geq 0)}$$

On peut remarquer que l'hypothèse d'induction $\Box(tl(z)+1 \geq 0)$ se retrouve maintenant à la conclusion, si bien qu'on s'arrête ici : le dernier séquent est trivialement vrai.

Ainsi, par la méthode de Cécile Dumas, nous avons transformé la preuve de la propriété de départ, définie en termes de suites infinies, en la preuve de trois propriétés scalaires. On peut raisonnablement supposer que ces dernières sont plus faciles à prouver.

3.4 Règles supplémentaires

Il convient d'ajouter deux autres règles à celles déjà évoquées.

3.4.1 Le cas de l'opérateur when

La stratégie de preuve que nous venons de décrire s'applique bien aux programmes où tous les flots évoluent à la même vitesse. Mais dès que plusieurs horloges entrent en jeu, la méthode n'est plus applicable. Intuitivement, c'est l'utilisation de l'opérateur `when` qui bloque les rouages : lorsque l'horloge `h` est fausse, le flot `x when h` n'a pas de sortie, si bien qu'on ne peut pas avancer (au sens des règles d'avancement), le but n'évolue pas et on n'arrive jamais au cas inductif.

Pour remédier à cet inconvénient, Cécile Dumas propose d'appliquer une règle de généralisation (gen) avant l'induction continue (cont). Cette nouvelle règle a pour but d'éliminer un opérateur `current` du programme étudié : on applique la règle autant de fois qu'il faut pour les éliminer tous.

Pourquoi éliminer les `current`, si le problème vient des `when` ? La raison est double : d'une part, on ne peut pas remédier au problème des `when` directement, car c'est leur sémantique de ne pas avoir de sortie tout le temps. Et d'autre part, on sait que tous les `when` seront encapsulés dans des `current`. En effet, si le programme Lustre dont on veut prouver un invariant comporte des flots sur une horloge différente de l'horloge de base, alors ces flots figureront dans la propriété à prouver dans (au moins) un opérateur

[2] Intuitivement, on veut garder l'hypothèse d'induction intacte, pour pouvoir conclure au pas suivant. En réalité, le choix des hypothèses à avancer ou non est plus compliqué et dépasse le cadre de cette présentation. On y reviendra au chapitre 6.

`current`, car le propriété, elle, est sur l'horloge de base. Par conséquent, tous les `when` éventuels seront aussi dans des `current`.

On définit un nouvel opérateur ternaire `default(`<*flot lent*>`,`<*flot rapide*>`,`<*horloge*>`)` tel que

$$\begin{array}{lcl}
\mathsf{default}(\mathsf{x}, \mathsf{y}, \varepsilon) & = & \varepsilon \\
\mathsf{default}(\mathsf{x}, \varepsilon, \mathsf{h}) & = & \varepsilon \\
\mathsf{default}(\mathsf{x}, y_0.\mathsf{y}, f.\mathsf{h}) & = & y_0.\mathsf{default}(\mathsf{x}, \mathsf{y}, \mathsf{h}) \\
\mathsf{default}(\varepsilon, \mathsf{y}, t.\mathsf{h}) & = & \varepsilon \\
\mathsf{default}(x_0.\mathsf{x}, y_0.\mathsf{y}, t.\mathsf{h}) & = & x_0.\mathsf{default}(\mathsf{x}, \mathsf{y}, \mathsf{h})
\end{array}$$

Le nouvel opérateur sert de définition récursive (et continue) à `current` :

$$\mathsf{current}(v, \mathsf{h}, \mathsf{x}) = \mathsf{default}(\mathsf{x}, v.\mathsf{current}(v, \mathsf{h}, \mathsf{x}), \mathsf{h})$$

La règle de généralisation s'énonce alors ainsi :

$$(\text{gen})\frac{P(\varepsilon) \qquad \forall m.\ P(m) \Rightarrow P(default(x, v.m, h))}{P(current(v, c, x))}$$

3.4.2 Le cas des hypothèses supplémentaires

Jusqu'à maintenant, nous avons travaillé avec des nœuds Lustre sans entrées : l'exemple 3.1 définit une fonction dont la sortie ne dépend pas des valeurs de l'entrée `rien`. Or de telles fonctions sont plutôt rares, car la plupart des systèmes effectuent un traitement des entrées.

Au niveau de la méthode générale, l'existence des entrées ne change rien, car, nous l'avons vu, le théorème de Kleene affirme la continuité de la fonction $e \mapsto \mu_x.\ x = f(e, x)$ où e représente les entrées. Ainsi, la règle d'induction continue peut toujours être appliquée, et donc les autres règles aussi.

Cependant, il existe une autre différence par rapport au cas sans entrées : on aimerait pouvoir formuler des hypothèses sur les entrées et en profiter dans la preuve.

Le langage propose une construction spéciale pour exprimer ces hypothèses dans le nœud lui-même : il s'agit de la clause

assertion : `assert` <*expression booléenne*> `;`

De telles clauses font partie du corps du nœud, au même titre que les équations. Leur signification est que « l'expression booléenne est considérée comme étant toujours vraie ». `assert` permet donc d'exprimer des invariants.

À titre d'illustration, nous pouvons reprendre l'exemple d'intégrateur discret de la page 18 et lui rajouter l'hypothèse selon laquelle l'entrée ne prend que des valeurs strictement positives.

```
node Integr(e:real) returns (s:real)
var p:real;
let
  s = e → (e + p);
  p = pre s;
  assert e > 0 ;
tel;
```

Exemple 3.2

Soit $Q(e, s, p)$ une propriété sur ce programme que l'on voudrait établir comme invariant (par exemple « la sortie est toujours positive » ou « la sortie est strictement croissante »), sous l'hypothèse $H = (e > 0)$. Dans un premier temps, nous pouvons formuler le problème ainsi : prouver $\Box Q$ sous l'hypothèse $\Box H$, c'est prouver

$$\text{formulation initiale} \qquad \Box H \Rightarrow \Box Q$$

Cependant, cette formulation n'est pas synchrone (car pas continue) : la valeur courante de cette implication dépend de toutes les valeurs, passées, présente et futures de l'hypothèse H. C'est pourquoi nous appliquons une règle de renforcement des hypothèses :

$$(\text{renf}) \frac{\Box(H \Rightarrow Q)}{\Box H \Rightarrow \Box Q}$$

De cette manière, nous obtenons une propriété continue, celle à laquelle nous appliquons la méthode de preuve décrite précédemment :

$$\text{formulation réelle} \qquad \Box(H \Rightarrow Q)$$

3.5 Règle d'induction continue revue

Une autre particularité relie les exemples que nous avons rencontrés jusqu'alors : le fait que la valeur courante de la sortie dépende d'au plus une valeur mémorisée. En fait, la méthode de preuve, telle que nous l'avons présentée, n'est pas bien adaptée aux programmes à plusieurs mémoires. Nous allons illustrer ce fait sur un exemple, avant de présenter la proposition de Cécile Dumas pour renforcer sa méthode.

3.5.1 L'échec sur la suite de Fibonacci

Un programme Un exemple simple d'un programme à plusieurs mémoires est le calcul de la suite de Fibonacci :

```
node fibo (rien:int) returns (s:int);
var m:int;
let
  s = 1 -> (pre(s) + pre(m)) ;
  m = 0 -> pre(s) ;
tel;
```

Exemple 3.3 Suite de Fibonacci

On se propose de démontrer que la sortie s est strictement positive (noté informellement $\Box(s > 0)$).

Formalisation En termes mathématiques, soit $fibo$ la fonction suivante :

$$\begin{array}{rccc} fibo: & (int \times int)^\infty & \rightarrow & (int \times int)^\infty \\ & \begin{bmatrix} s \\ m \end{bmatrix} & \mapsto & \begin{bmatrix} 1.(s+m) \\ 0.s \end{bmatrix} \end{array}$$

Par construction, la fonction $fibo$ décrit les calculs effectués dans le nœud `fibo` sur le vecteur de variables entières $\begin{bmatrix} s \\ m \end{bmatrix}$. Soit alors μ_{fibo} le plus petit point fixe de cette fonction ($\mu_{fibo} \equiv \mu. \begin{bmatrix} s \\ m \end{bmatrix} = \begin{bmatrix} 1.(s+m) \\ 0.s \end{bmatrix}$). Soit $\pi_1(\mu_{fibo})$ la projection de μ_{fibo} sur sa première composante. La propriété que nous voulons prouver s'écrit donc formellement

$$\Box(\pi_1(\mu_{fibo}) > 0)$$

Application de la continuité Rappelons la règle de continuité (3.2) de la page 30 :

$$(\text{cont})\frac{\Box P(\varepsilon) \qquad \forall y \in D^*.\ \Box P(y) \Rightarrow \Box P(f(y))}{\Box P(\mu_x.x = f(x))}$$

Pour pouvoir appliquer cette règle, il nous faut d'abord unifier les expressions $\Box P(\mu_x.x = f(x))$ et $\Box(\pi_1(\mu_{fibo}) > 0)$. C'est possible, en posant

- le couple $\left[\begin{smallmatrix} s \\ m \end{smallmatrix}\right]$ correspond à la variable (vectorielle) x ; par ailleurs, ceci implique que le domaine D est en fait $int \times int$
- la fonction $fibo$ vient se substituer à f
- la propriété $\Box P(\_)$ s'exprime dans notre cas comme $\Box(\pi_1(\_) > 0)$

Il est donc possible d'appliquer la règle de continuité sur l'exemple de la suite de Fibonacci. Cette règle affirme que pour prouver la propriété recherchée, il suffit de prouver $\Box P(\varepsilon)$ et $\forall y \in D^*.\ \Box P(y) \Rightarrow \Box P(f(y))$. Concentrons-nous, comme Cécile Dumas, sur la seconde propriété : dans notre cas, elle s'écrit

$$\forall y \in (int \times int)^*.\Box(\pi_1(y) > 0) \Rightarrow \Box(\pi_1(fibo(y)) > 0)$$

Comme y est une variable vectorielle à deux composantes, nous pouvons naturellement poser $y = \left[\begin{smallmatrix} y_1 \\ y_2 \end{smallmatrix}\right]$. Ainsi, l'expression ci-dessus se réécrit en

$$\forall \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \in (int \times int)^*.\Box(\pi_1(\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}) > 0) \Rightarrow \Box(\pi_1(fibo(\begin{bmatrix} y_1 \\ y_2 \end{bmatrix})) > 0)$$

puis en

$$\forall \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \in (int \times int)^*.\Box(y_1 > 0) \Rightarrow \Box(\pi_1(\begin{bmatrix} 1.(y_1 + y_2) \\ 0.y_1 \end{bmatrix}) > 0)$$

et finalement en

$$\forall \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \in (int \times int)^*.\Box(y_1 > 0) \Rightarrow \Box(1.(y_1 + y_2) > 0)$$

Après skolémisation, nous obtenons simplement

$$\Box(y_1 > 0) \Rightarrow \Box(1.(y_1 + y_2) > 0)$$

Cependant, Cécile Dumas propose de conserver les noms des variables du programme, afin de souligner son lien avec la preuve. Sur notre exemple, le renommage aboutit à l'expression suivante :

$$\Box(s > 0) \Rightarrow \Box(1.(s + m) > 0)$$

Dans cette expression, les variables s et m sont des variables libres représentant des suites finies ; en particulier, (s, m) n'est pas le point fixe associé à la fonction $fibo$.[3]

Avant de continuer, nous appliquons une dernière transformation pour passer de l'implication ci-dessus au séquent suivant :

$$\frac{\Box(s > 0)}{\Box(1.(s + m) > 0)}$$

Cette transformation exploite de manière triviale le théorème de la déduction (vrai en déduction naturelle), d'après lequel $\vdash A \Rightarrow B$ est équivalent à $A \vdash B$.[4]

[3] Ce renommage est utile du point de vue pratique, car il permet de partir de la propriété de départ ($\Box(s > 0)$) pour arriver directement à l'expression dépliée ($\Box(s > 0) \Rightarrow \Box(1.(s + m) > 0)$) par un simple jeu de réécriture. Néanmoins, nous verrons plus loin que ce renommage peut mener à une confusion dangereuse.

[4] En effet, $\frac{A \vdash B}{\vdash A \Rightarrow B}$ est la règle même d'introduction de $\Rightarrow$. Inversement, ayant $\vdash A \Rightarrow B$, on peut déduire que $A \vdash A \Rightarrow B$. Comme par ailleurs $A \vdash A$ est un axiome, on peut conclure que $A \vdash B$.

Induction sur les suites finies Avant d'entrer dans le détail de cette phase de la preuve, annonçons tout de suite le résultat : il n'est pas possible de prouver le séquent ci-dessus, car s et m sont des variables libres, mais seule s est contrainte dans les hypothèses.

En particulier, la méthode des avancements mène bien à une impasse... Pour commencer, appliquons la règle (cons_2) de la page 31 à la conclusion du séquent ci-dessus. Nous obtenons deux nouveaux séquents :

$$(seq_1)\frac{\Box(s > 0)}{1 > 0} \qquad (seq_2)\frac{\Box(s > 0)}{\Box(s + m > 0)}$$

Après affaiblissement de l'hypothèse, le séquent (seq_1) donne la première obligation de preuve (triviale)

$$(OP_1)\frac{}{1 > 0}$$

Ensuite, le séquent (seq_2) peut être séparé en deux nouveaux séquents par application de la règle (cons_2) :

$$(seq_{2.1})\frac{\Box(s > 0)}{hd(s) + hd(m) > 0} \qquad (seq_{2.2})\frac{\Box(s > 0)}{\Box(tl(s) + tl(m) > 0)}$$

L'hypothèse du premier de ces séquents peut être affaiblie[5] par la règle (hd) de la page 31 en $hd(s) > 0$, si bien que la deuxième obligation de preuve est

$$(OP_2)\frac{hd(s) > 0}{hd(s) + hd(m) > 0}$$

Il est clair qu'il n'est pas nécessaire de poursuivre le processus de preuve plus loin : comme l'obligation (OP_2) n'est pas prouvable, toute la preuve échoue.

Dans cet exemple, nous avons illustré le fait que la règle de continuité (cont) est très loin d'être complète : même sur un exemple aussi simple que celui de la suite de Fibonacci, elle mène à une impossibilité. C'est pourquoi Cécile Dumas a proposé un renforcement de cette règle, discuté dans la section suivante.

3.5.2 La correction ratée

On « voit » bien que le problème soulevé sur l'exemple de la suite de Fibonacci est celui de l'oubli de certaines parties de la dynamique du système. Dans ce cas précis, c'est l'une des équations qui est occultée, si bien qu'aucune information sur ses valeurs ne parvient à l'obligation de preuve.

a) Une règle aménagée

Cécile Dumas propose alors d'utiliser une règle de continuité fragmentée, plusieurs fois de suite : à chaque utilisation, seule une équation du système est « déroulée », si bien que toutes font leur chemin jusque dans les obligations de preuve. La règle est énoncée ainsi :

$$(\text{cont}_i)\frac{P(\varepsilon) \qquad P(X) \Rightarrow P(E_i(\mu_X.\ X = E(X)))}{P(\mu_X.\ X = E(X))} \qquad \text{où } E_i \text{ est la } i^{\text{ème}} \text{ équation}$$

Même si cette formulation n'est pas totalement correcte du point de vue syntaxique, son sens est bien celui de « l'induction continue appliquée équation par équation ».

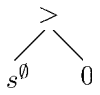
En étudiant les exemples d'utilisation de la règle ci-dessus dans la thèse de Cécile Dumas et son implémentation dans l'outil de preuve Gloups, nous sommes en mesure de proposer une formulation plus précise de cette règle, sous forme de règles de réécriture :

[5] L'affaiblissement des hypothèses se justifie intuitivement par le caractère causal de Lustre : une valeur ponctuelle ne peut pas dépendre d'une valeur future, si bien qu'il est inutile de conserver les hypothèses au-delà d'un certain point.

1. Soit n le nombre de flots calculés par le nœud Lustre qui nous intéresse. Il y a donc n sorties et/ou variables locales, toutes confondues. Appelons $x_1, \ldots, x_n$ ces flots. Par ailleurs, le nœud en question décrit un système de n équations mutuellement récursives, $E_1, \ldots, E_n$, chacune portant potentiellement sur l'ensemble des flots. Ainsi, le système s'écrit

$$\begin{array}{rcl} x_1 & = & E_1(x_1, \ldots, x_n) \\ \vdots & & \vdots \\ x_n & = & E_n(x_1, \ldots, x_n) \end{array}$$

2. L'idée générale de la règle d'induction fragmentée est de pouvoir utiliser les équations comme des règles de réécriture, indépendamment les unes des autres. Pour cela, chaque équation est orientée de gauche à droite : $x_j = E_j(x_1, \ldots, x_n)$ devient $x_j \to E_j(x_1, \ldots, x_n)$, ce qui signifie que x_j se réécrit en $E_j(x_1, \ldots, x_n)$.
   Ainsi, appliquer la règle (cont_i) sur un séquent de la forme $\frac{H}{C}$, c'est recopier la conclusion C dans les hypothèses H, puis réécrire la conclusion par l'une des règles de réécriture. On obtient alors un nouveau séquent de la forme $\frac{H \quad C}{C'}$ où $C' = C[x_i \to E_i(x_1, \ldots, x_n)]$.[6]
3. Cependant, le système de réécriture exposé au point précédent doit être associé à une stratégie, sous peine d'être non-terminant. Cette stratégie n'est pas exprimée dans la règle (cont_i), mais elle figure de manière informelle dans la thèse de Cécile Dumas et elle est implémentée dans son outil Gloups.
   La stratégie de réécriture suit deux principes : elle est en profondeur d'abord et chaque équation est déroulée seulement une fois pour chaque occurrence de la variable concernée.
   Pour exprimer cette stratégie de manière plus formelle, nous allons considérer qu'une expression Lustre (par exemple celle qu'on souhaite prouver) est un arbre (syntaxique) dont les feuilles sont des flots. De plus, on associe à chaque feuille qui représente une variable un ensemble de (noms de) variables, qui sera initialement vide. Nous appelerons cet ensemble la trace de la feuille.
   Ainsi, le schéma suivant représente l'expression $s > 0$, avec la trace vide :



   La stratégie consiste alors à
   - choisir la feuille la plus à gauche (appelée φ), telle que la trace de cette feuille ne contient pas le nom de la feuille (si x_j est la variable représentée par la feuille φ, on veut $x_j \notin trace(\varphi)$)
   - réécrire l'expression entière avec la règle de réécriture associée à φ ($x_j \to E_j(x_1, \ldots, x_n)$)
   - mettre à jour les traces de toutes les feuilles : si une feuille vient d'être réécrite, on ajoute à sa trace la variable x_j

Illustrons ce procédé sur l'exemple de Fibonacci :

$$\text{le système équationnel } \left\{ \begin{array}{rcl} s & = & 1.(s+m) \\ m & = & 0.s \end{array} \right. \quad \text{devient le système de réécriture } \begin{array}{rcl} s & \to & 1.(s+m) \\ m & \to & 0.s \end{array}$$

La propriété de départ, $\Box(s > 0)$, devient le séquent $\overline{\Box(s > 0)}$. La conclusion de ce séquent est l'expression dont l'arbre figure ci-dessus ; en particulier, la trace de s est vide. Nous pouvons l'écrire sous une forme

[6] En effet, on part de $\frac{H}{C}$ que l'on assimile à la conclusion de (cont_i), $\frac{H}{\Box P(\mu_X.\ X = E(X))}$. Ainsi, l'expression $\Box P(X)$ est syntaxiquement la même que C, mais sémantiquement, les variables liées sont devenues libres, sans changer de nom. Puis $\Box P(E_i(\mu_X.\ X = E(X)))$ est ce qu'on a appelé C', c'est-à-dire C réécrit par l'équation E_i. Le séquent de départ, $\frac{H}{C}$ est donc devenu $\frac{H}{C \Rightarrow C'}$, puis $\frac{H \quad C}{C'}$.

compacte :

$$\frac{}{\Box(s^{\emptyset} > 0)}$$

La stratégie de réécriture s'applique donc sur la seule feuille possible, indiquée par un soulignage :

$$\frac{}{\Box(\underline{s^{\emptyset}} > 0)} \longrightarrow \frac{\Box(s > 0)}{\Box(1.(s^{\{s\}} + m^{\{s\}}) > 0)}$$

Puis, une deuxième réécriture peut s'appliquer sur m :

$$\frac{\Box(s > 0)}{\Box(1.(s^{\{s\}} + \underline{m^{\{s\}}}) > 0)} \longrightarrow \frac{\Box(s > 0) \quad \Box(1.(s + m) > 0)}{\Box(1.(s^{\{s\}} + 0.s^{\{s,m\}}) > 0)}$$

La réécriture s'arrête ici, car désormais les traces de toutes les feuilles contiennent le nom de la feuille.

Nous venons d'établir deux application de la règle (cont_i) : après deux dépliages, nous arrivons au séquent

$$\frac{\Box(s > 0) \qquad \Box(1.(s + m) > 0)}{\Box(1.(s + 0.s) > 0)}$$

Ce séquent donne naissance aux trois obligations de preuve suivantes, toutes les trois faciles à prouver :

$$\frac{}{1 > 0} \quad , \quad \frac{1 > 0 \qquad hd(s) > 0}{hd(s) + 0 > 0} \quad , \quad \frac{1 > 0 \qquad hd(s) > 0 \qquad hd(tl(s)) > 0 \qquad hd(s) + hd(m) > 0}{hd(tl(s)) + hd(s) > 0}$$

Nous ne souhaitons pas détailler la manière dont Cécile Dumas obtient ces obligations, ni l'argument qui lui permet d'affirmer que ces trois-là sont suffisantes pour prouver la propriété de départ. D'une part, nous souhaitons seulement illustrer le gain en puissance que la règle (cont_i) représente aux yeux de son auteur et, d'autre part, l'exemple suivant nous donnera l'occasion d'une étude plus détaillée.

b) Un pavé dans la mare

Malheureusement, nous avons constaté que **la règle d'induction continue fragmentée n'est pas correcte**, car elle permet de prouver des propriétés fausses.

Un programme Comme contre-exemple, voici un nœud à deux sorties :

```
node faux(i : bool) returns (o, p : bool);
var s : bool ;
let
  s = false -> not pre s ;
  o = false -> if i and pre i then pre o else s ;
  p = false -> if i           then pre p else s ;
tel;
```

Exemple 3.4 Contre-exemple pour la règle (cont_i)

Il est clair que les sorties `o` et `p` ne sont pas égales. Il suffit en effet de faire une exécution sur deux pas pour s'en rendre compte :

i	f	t	...
s	f	t	...
o	f	$\boxed{f}$	...
p	f	$\boxed{t}$	...

Induction continue fragmentée Pourtant, la méthode de preuve de Cécile Dumas, avec utilisation de la règle de continuité fragmentée, permet de « prouver » que $\Box(o = p)$! En effet, partant de ce but et en appliquant (cont_i), on obtient en déroulant quatre fois :

$$(seq_A)\frac{}{\Box(\underline{o^{\emptyset}} = p^{\emptyset})}$$

$$\downarrow$$

$$(seq_B)\frac{\Box(o = p)}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ tl(\underline{s^{\{o\}}})) = p^{\emptyset})}$$

$$\downarrow^{7}$$

$$(seq_C)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ \neg s^{\{o,s\}}) = \underline{p^{\emptyset}})}$$

$$\downarrow$$

$$(seq_D)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ \neg s^{\{o,s\}}) = false.(if\ tl(i)\ then\ p^{\{p\}}\ else\ tl(\underline{s^{\{p\}}})))}$$

$$\downarrow$$

$$(seq_E)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = false.(if\ tl(i)\ then\ p\ else\ tl(s)))\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ \neg s^{\{o,s\}}) = false.(if\ tl(i)\ then\ p^{\{p\}}\ else\ \neg s^{\{p,s\}}))}$$

À la fin des déroulements, nous obtenons donc le séquent suivant (sans les traces) :

$$(seq_E)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = false.(if\ tl(i)\ then\ p\ else\ tl(s)))\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = false.(if\ tl(i)\ then\ p\ else\ \neg s))}$$

C'est bien le même séquent qui est généré par Gloups.

Avancements Ensuite, arrive la phase des avancements, qui crée les obligations de preuve. D'abord, (seq_E) est divisé en deux séquents, (seq_1) et (seq_2), par l'application de la règle (cons_2) de la page 31. Nous allons commencer par (seq_1) et nous nous intéresserons à (seq_2) plus tard :

$$(seq_1)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = false.(if\ tl(i)\ then\ p\ else\ tl(s)))\end{array}}{false = false}$$

[7] Pour plus de lisibilité, nous simplifions $tl(false.(\neg s))$ en $\neg s$.

Après avoir affaibli toutes les hypothèses de seq_1 grâce à la règle (hd) de la page 31, nous obtenons notre première obligation de preuve :

$$(OP_a)\frac{hd(o) = hd(p) \qquad false = hd(p) \qquad false = hd(p) \qquad false = false}{false = false}$$

Revenons à (seq_2) :

$$(seq_2)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = false.(if\ tl(i)\ then\ p\ else\ tl(s)))\end{array}}{\Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = if\ tl(i)\ then\ p\ else\ \neg s)}$$

En affaiblissant les hypothèses par la règle (tl) de la page 31, il vient :

$$(seq_{2'})\frac{\begin{array}{l}\Box(tl(o) = tl(p))\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ tl(s) = p)\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = p)\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = if\ tl(i)\ then\ p\ else\ tl(s))\end{array}}{\Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = if\ tl(i)\ then\ p\ else\ \neg s)}$$

Le séquent $(seq_{2'})$ donne à nouveau deux séquents par l'application de la règle (cons_2) : $(seq_{2''})$ et (seq_3). Comme précédemment, étudions d'abord celui qui donnera lieu à une obligation de preuve :

$$(seq_{2''})\frac{\begin{array}{l}\Box(tl(o) = tl(p))\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ tl(s) = p)\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = p)\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = if\ tl(i)\ then\ p\ else\ tl(s))\end{array}}{if\ hd(tl(i)) \wedge hd(i)\ then\ hd(o)\ else\ \neg hd(s) = if\ hd(tl(i))\ then\ hd(p)\ else\ \neg hd(s)}$$

L'affaiblissement des hypothèses par (hd) crée la deuxième obligation de preuve :

$$(OP_b)\frac{\begin{array}{l}hd(tl(o)) = hd(tl(p))\\ if\ hd(tl(i)) \wedge hd(i)\ then\ hd(o)\ else\ hd(tl(s)) = hd(p)\\ if\ hd(tl(i)) \wedge hd(i)\ then\ hd(o)\ else\ \neg hd(s) = hd(p)\\ if\ hd(tl(i)) \wedge hd(i)\ then\ hd(o)\ else\ \neg hd(s) = if\ hd(tl(i))\ then\ hd(p)\ else\ hd(tl(s))\end{array}}{if\ hd(tl(i)) \wedge hd(i)\ then\ hd(o)\ else\ \neg hd(s) = if\ hd(tl(i))\ then\ hd(p)\ else\ \neg hd(s)}$$

Enfin, le séquent (seq_3) est celui-ci :

$$(seq_3)\frac{\begin{array}{l}\Box(tl(o) = tl(p))\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ tl(s) = p)\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = p)\\ \Box(if\ tl(i) \wedge i\ then\ o\ else\ \neg s = if\ tl(i)\ then\ p\ else\ tl(s))\end{array}}{\Box(if\ tl^2(i) \wedge tl(i)\ then\ tl(o)\ else\ \neg tl(s) = if\ tl^2(i)\ then\ tl(p)\ else\ \neg tl(s))}$$

Après affaiblissement des hypothèses par (tl), nous obtenons :

$$(seq_{3'})\frac{\begin{array}{l}\Box(tl^2(o) = tl^2(p))\\ \Box(if\ tl^2(i) \wedge tl(i)\ then\ tl(o)\ else\ tl^2(s) = tl(p))\\ \Box(if\ tl^2(i) \wedge tl(i)\ then\ tl(o)\ else\ \neg tl(s) = tl(p))\\ \Box(if\ tl^2(i) \wedge tl(i)\ then\ tl(o)\ else\ \neg tl(s) = if\ tl^2(i)\ then\ tl(p)\ else\ tl^2(s))\end{array}}{\Box(if\ tl^2(i) \wedge tl(i)\ then\ tl(o)\ else\ \neg tl(s) = if\ tl^2(i)\ then\ tl(p)\ else\ \neg tl(s))}$$

Le point important est que $(seq_{3'})$ et $(seq_{2'})$ sont de la même forme : on passe simplement de celui-ci à celui-là en appliquant l'opérateur $tl(\_)$ à toutes les variables. Cela signifie que si on continue le processus de preuve, c'est-à-dire la séparation en deux par (cons_2), puis affaiblissement etc., toutes les obligations de preuve obtenues seront de la forme de (OP_b), seulement avec $tl^n(\_)$ à la place de $tl(\_)$ (avec $n > 1$). Ainsi, (OP_b) est le *prototype* de toutes les obligations de preuve que nous pourrions obtenir en poursuivant le processus : si nous savons prouver (OP_b), nous savons également prouver toutes ces obligations. En conclusion, il suffit de s'arrêter ici, avec seulement deux obligations de preuve.

L'argument d'arrêt que nous venons d'évoquer est en fait celui de la preuve infinie de Coquand [14]. Le théorème cité assure qu'une preuve dont l'arbre est rationnel (une branche de la preuve reboucle sur elle-même comme sur la figure 3.1) peut être rendue finie (seul un cas *typique* ou *prototype* est prouvé sur la branche qui boucle) à condition que ledit rebouclage passe par l'avancement d'au moins un constructeur. Ici, c'est le cas : entre $(seq_{2'})$ et $(seq_{3'})$, le séquent avance par le constructeur des suites, ce qui se traduit par l'application de $tl(\_)$.[8]

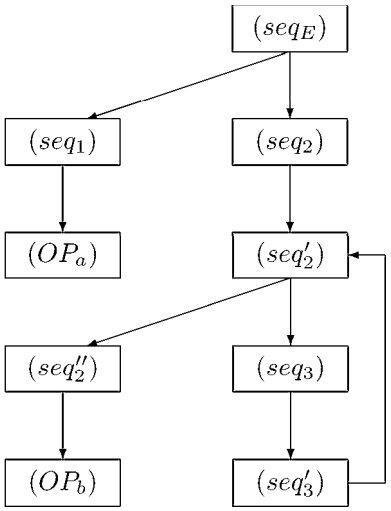


FIG. 3.1 – Arbre de preuve de $o = p$

Conclusion En fin de compte, le programme `faux` de départ et la propriété à prouver $\Box(o = p)$ mènent à deux obligations de preuve, (OP_a) et (OP_b). Comme les deux sont prouvables (pour (OP_b), il suffit d'examiner les quatre cas engendrés par les valeurs de $hd(i)$ et $hd(tl(i))$), on peut en conclure que la méthode de l'induction fragmentée « prouve » la propriété recherchée.

Étant donné que toutes les règles utilisées dans le paragraphe **Avancements** sont soit triviales (comme (cons_2), (hd) et (tl)), soit communément admises (Coquand), la phase des avancements n'a pas pu introduire d'erreur dans la raisonnement. Par conséquent, la seule règle qui puisse ne pas être correcte est celle de l'induction continue fragmentée. Nous pouvons donc conclure que le contre-exemple que nous venons d'étudier établit bien que la règle (cont_i) est incorrecte.

[8] Le chapitre 6 présente un argument d'arrêt plus général, l'hypothèse d'induction implicite. Nous n'en avons pas besoin ici et nous croyons que l'argument de Coquand est plus intuitif. Par ailleurs, le chapitre 6 donne tous les détails concernant l'application des règles d'affaiblissement.

Explication Pour quelle raison la règle (cont_i) est incorrecte ? Au vu des résultats que nous présenterons au chapitre 4, nous pouvons d'ores et déjà affirmer que le problème principal est celui de la capture de variables libres. Nous allons essayer d'expliquer ce point sur l'exemple que nous venons de voir.

Lors des déroulements, les expressions contenues dans les hypothèses des séquents ne contiennent que des variables libres. Par contre, la conclusion contient un mélange de variables libres (déjà déroulées, c'est-à-dire celles dont la trace contient leur nom) et liées (non encore déroulées). Ainsi, par exemple dans le séquent (seq_C) précédent :

$$(seq_C)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ \neg s^{\{o,s\}}) = \underline{p^{\emptyset}})}$$

$\Big\}$ ici o, p et s sont libres

$\Big\}$ ici o et s sont libres, p est liée[9]

Regardons alors de plus près le dernier déroulement :

$$(seq_D)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ \neg s^{\{o,s\}}) = false.(if\ tl(i)\ then\ p^{\{p\}}\ else\ tl(\underline{s^{\{p\}}})))}$$

$$\downarrow$$

$$(seq_E)\frac{\begin{array}{l}\Box(o = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ tl(s)) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = p)\\ \Box(false.(if\ tl(i) \wedge i\ then\ o\ else\ \neg s) = false.(if\ tl(i)\ then\ p\ else\ tl(s)))\end{array}}{\Box(false.(if\ tl(i) \wedge i\ then\ o^{\{o\}}\ else\ \neg s^{\{o,s\}}) = false.(if\ tl(i)\ then\ p^{\{p\}}\ else\ \neg s^{\{p,s\}}))}$$

Dans (seq_D), la variable qu'on déroule ($s^{\{p\}}$) est liée. Après le déroulement, elle devient libre. Mais le séquent, et en particulier sa conclusion, contenait déjà une variable libre s : en déroulant $s^{\{p\}}$ en $s^{\{p,s\}}$, nous avons rendu la seconde variable libre s égale à la première ($s^{\{o,s\}}$). Ainsi, le renommage systématique prôné par Cécile Dumas a conduit à une capture de variable. Cette capture s'est traduit par l'introduction d'une contrainte supplémentaire (« $s^{\{o,s\}} = s^{\{p,s\}}$ »), qui n'était pas présente dans le programme de départ. À notre avis, ceci est la raison principale de la non-correction de la règle d'induction continue fragmentée.

3.6 Bilan de la méthode

La méthode de preuve de Cécile Dumas, telle que nous l'avons présentée dans les sections 3.2 et 3.4 est une méthode correcte, qui s'appuie sur la sémantique de Lustre pour transformer une propriété de suites infinies en une série de propriétés scalaires. Ainsi, cette méthode réduit un problème exprimé en fonction d'un comportement infini en quelques problèmes ponctuels, qu'on peut supposer plus faciles à prouver.

La méthode utilise l'induction comme outil principal ; en fait, deux inductions, continue et structurelle, sont utilisées à la suite. Mais nous avons constaté sur l'exemple 3.3 que l'expressivité de cette méthode était limitée : même des propriétés simples sur des systèmes peu compliqués à plusieurs mémoires ne peuvent pas être prouvées.

Cécile Dumas propose alors une règle d'induction continue plus forte, qui semble résoudre les problèmes de pouvoir d'expression. Malheureusement, cette nouvelle règle (la seule qui ne soit pas justifiée

[9]En fait, $p^{\emptyset}$ dans la conclusion resprésente l'une des composantes du point fixe qui définit la sémantique du nœud `faux`.

dans son mémoire) s'est avérée être incorrecte, ce qui est d'autant plus dommageable qu'elle faisait déjà partie d'un outil de preuve, Gloups.

La deuxième partie de ce mémoire est dédiée aux recherches que nous avons entreprises afin de corriger, de manière théorique et pratique, la règle erronée.

Deuxième partie

Travail sur les preuves

4 – L'induction continue fragmentée

Nous essayons ici de corriger la règle incorrecte d'induction continue fragmentée. Nous justifions qu'il est possible de fragmenter et nous déduisons une règle correcte, mais elle s'avère d'un intérêt pratique très limité.

Nous avons expliqué au chapitre précédent que la méthode de preuve de Cécile Dumas était intéressante, mais limitée dans son usage à cause d'une règle d'induction continue trop faible. La règle d'induction fragmentée, celle qui permettait d'établir des raisonnements plus puissants, s'était avérée fausse : c'est pourquoi nous avons entrepris d'élaborer une version correcte de cette règle, afin d'améliorer l'expressivité de la méthode.

Nous verrons qu'il est possible de définir une règle fragmentée correcte sur le plan théorique, mais que son usage est peu commode en pratique. Néanmoins, nous pensons qu'il est important de présenter notre démarche, pour que le lecteur comprenne les raisons qui nous ont poussé à rechercher une autre solution.

4.1 Justification de la fragmentation

Énoncé du problème Pour commencer, nous allons exposer une justification théorique de la possibilité d'appliquer les équations séparément. En effet, la sémantique du système d'équation d'un nœud Lustre est définie en termes d'un point fixe global, qui concerne toutes les équations à la fois. Il n'est donc pas évident de savoir s'il est correct de s'intéresser aux équations l'une après l'autre, plutôt que toutes ensemble.

Tout au long de ce chapitre, nous allons nous intéresser au cas d'un système à deux équations, sachant que les résultats pourraient être généralisés à un nombre quelconque d'équations en itérant le procédé :

$$\text{Système d'étude :} \quad \begin{cases} x &= g(x,y) \\ y &= h(x,y) \end{cases} \qquad \text{où } x,y \in D^\infty \text{ et } f,\ g \text{ sont continues}$$

Nous nous intéressons à la preuve d'une propriété invariante

$$\Box P(\mu_{x,y}.\ x,y = g(x,y),h(x,y)) \tag{4.1}$$

Soit (μ_1,μ_2) le plus petit point fixe qui vérifie l'équation $x,y = g(x,y),h(x,y)$: ainsi, $\mu_1 = g(\mu_1,\mu_2)$, $\mu_2 = h(\mu_1,\mu_2)$ et la propriété qui nous intéresse est $\Box P(\mu_1,\mu_2)$.

Soit $H(x)$, G et F les suites de D^∞ telles que

$$\begin{array}{rcl} H(x) &=& \mu_y.\ (y = h(x,y)) \\ G &=& \mu_x.\ (x = g(x,H(x))) \\ F &=& \mu_y.\ (y = h(G,y)) \end{array}$$

En particulier, la suite G est bien définie, car la fonction $x \mapsto H(x)$ est continue, en vertu du théorème de Kleene (cf. page 24).

Nous nous proposons alors de démontrer que le problème initial (4.1) peut se réduire à

$$\Box P(G, F) \tag{4.2}$$

Sous cette forme, nous aurons séparé le point fixe $\mu_{x,y}$ en des points fixes selon x et y et il deviendra possible d'effectuer l'induction continue séparément sur chaque équation.

$$(4.2) \equiv \Box P(\mu_x.\ (x = g(x, \mu_y.\ (y = h(x, y)))), \mu_y.\ (y = h(\mu_x.\ (x = g(x, \mu_y.\ (y = h(x, y)))), y)))$$

Réduction du problème Par définition, F est le plus petit point fixe tel que $F = h(G, F)$. Or, par définition de $H(x)$, $H(G)$ est le plus petit point fixe tel que $H(G) = h(G, H(G))$, donc nécessairement $F = H(G)$ (c'est le même plus petit point fixe).

Comme par ailleurs $G = g(G, H(G))$, nous avons

$$\left.\begin{array}{rcl} G & = & g(G, H(G)) \\ F & = & h(G, F) \\ F & = & H(G) \end{array}\right\} \text{ alors } (G, F) = (g(G, F), h(G, F))$$

En outre, la définition de (μ_1, μ_2) est d'être le plus petit point fixe tel que $(\mu_1, \mu_2) = (g(\mu_1, \mu_2), h(\mu_1, \mu_2))$. Par conséquent, comme (G, F) est un point fixe satisfaisant la même équation,

$$(\mu_1, \mu_2) \preceq (G, F)$$

Cette inégalité justifie notre réduction du problème de départ :

$$\text{prouver } \Box P(G, F) \text{ (4.2) est suffisant pour prouver } \Box P(\mu_1, \mu_2) \text{ (4.1)}$$

Équivalence On pourrait s'arrêter à la condition suffisante que l'on vient d'énoncer. Néanmoins, il est intéressant de signaler que cette condition est également nécessaire. Pour le prouver, nous allons utiliser une propriété importante des fonctions continues, qui est la monotonie :

Une fonction f sur $\langle D^\infty, \varepsilon, ., \preceq \rangle$ est *monotone* si

$$\forall x, y \in D^\infty.\ (x \preceq y) \Rightarrow (f(x) \preceq f(y))$$

Une fonction f continue est alors forcément monotone, car si $x \preceq y$, alors $\{x, y\}$ est une chaîne croissante, et donc, par continuité, $f(y) = f(\sup\{x, y\}) = \sup f(\{x, y\})$, d'où $f(x) \preceq f(y)$.

Nous allons procéder en plusieurs raisonnements par induction continue.

1) Cas de H(x) Soit la fonction $\lambda_x(\alpha)$ où $x, \alpha \in D^\infty$ telle que

$$\lambda_x(\alpha) = h(x, \alpha)$$

De plus, on pose $\lambda_x^0(\alpha) = \alpha$ et $\lambda_x^{n+1}(\alpha) = \lambda_x(\lambda_x^n(\alpha))$. On note que $\lambda_x : \alpha \mapsto h(x, \alpha)$ est continue, car h est continue. En utilisant λ_x, nous allons prouver que si $x \preceq \mu_1$, alors $H(x) \preceq \mu_2$.

- D'abord, remarquons que $\lambda_x^0(\varepsilon) = \varepsilon \preceq \mu_2$.
- Ensuite, si $x \preceq \mu_1$ et $\lambda_x^n(\varepsilon) \preceq \mu_2$, alors par monotonie $\lambda_x(\lambda_x^n(\varepsilon)) \preceq \lambda_x(\mu_2)$. Or $\lambda_x(\mu_2) = h(x, \mu_2)$ et $(x, \mu_2) \preceq (\mu_1, \mu_2)$, donc par monotonie de h, $h(x, \mu_2) \preceq h(\mu_1, \mu_2) = \mu_2$. On en déduit que $\lambda_x(\mu_2) \preceq \mu_2$ et par transitivité $\lambda_x^{n+1}(\varepsilon) \preceq \mu_2$.

- En raisonnant par induction sur $\mathbb{N}$, on déduit des deux points précédents que pour tout $x \preceq \mu_1$, $\forall n \in \mathbb{N}.\lambda_x^n(\varepsilon) \preceq \mu_2$.
- En appliquant le théorème de Kleene, on trouve que $H(x)$, qui est le plus petit point fixe de l'équation $y = h(x, y)$ est égal à $H(x) = \sup\{\varepsilon, h(x,\varepsilon), h(x, h(x,\varepsilon)), \ldots\}$. Avec les notations précédentes, $H(x) = \sup\{\lambda_x^0(\varepsilon), \lambda_x^1(\varepsilon), \ldots, \lambda_x^n(\varepsilon), \ldots\}$. Comme chaque élément de cet ensemble est inférieur à μ_2 (à condition que $x \preceq \mu_1$), par continuité $H(x) \preceq \mu_2$.
  Ainsi, nous avons prouvé que si $x \preceq \mu_1$, alors $H(x) \preceq \mu_2$.

□

2) Cas de G À partir du résultat précédent, nous allons prouver que $G \preceq \mu_1$. Pour cela, soit la fonction κ telle que

$$\kappa(x) = g(x, H(x)) \text{ avec } x \in D^\infty$$

- $\kappa^0(\varepsilon) = g(\varepsilon, H(\varepsilon))$: Or $\varepsilon \preceq \mu_1$, donc $H(\varepsilon) \preceq \mu_2$ par le théorème précédent. D'où on déduit $(\varepsilon, H(\varepsilon)) \preceq (\varepsilon, \mu_2) \preceq (\mu_1, \mu_2)$. Par monotonie de g, $\underline{\kappa^0(\varepsilon)} = g(\varepsilon, H(\varepsilon)) \preceq g(\mu_1, \mu_2) = \underline{\mu_1}$.
- Supposons maintenant que $\kappa^n(\varepsilon) \preceq \mu_1$. Le théorème précédent nous permet donc de conclure que $H(\kappa^n(\varepsilon)) \preceq \mu_2$, d'où on déduit que $(\kappa^n(\varepsilon), H(\kappa^n(\varepsilon))) \preceq (\mu_1, H(\kappa^n(\varepsilon))) \preceq (\mu_1, \mu_2)$. Par monotonie de g, on obtient $\kappa^{n+1}(\varepsilon) = g(\kappa^n(\varepsilon), H(\kappa^n(\varepsilon))) \preceq g(\mu_1, \mu_2) = \mu_1$.
- Par induction sur $\mathbb{N}$, on conclut que $\forall n \in \mathbb{N}.\kappa^n(\varepsilon) \preceq \mu_1$.
- Le théorème de Kleene affirme que $G = \sup\{\varepsilon, g(\varepsilon, H(\varepsilon)), g(g(\varepsilon, H(\varepsilon)), H(g(\varepsilon, H(\varepsilon)))), \ldots\}$, si bien que $G = \sup_{n\geq 0}\{\kappa^n(\varepsilon)\}$. Par continuité, nous pouvons conclure que $G \preceq \mu_1$.

□

3) Conclusion De la même manière que pour $H(x)$ et G, nous pourrions démontrer que $F \preceq \mu_2$. Ces inégalités nous permettent alors d'affirmer que

$$(G, F) \preceq (\mu_1, \mu_2)$$

Étant donné que par ailleurs, $(\mu_1, \mu_2) \preceq (G, F)$, nous pouvons en déduire que

$$(G, F) = (\mu_1, \mu_2)$$

si bien que le problème (4.2) est **équivalent** au problème (4.1).

4.2 Dérivation d'une règle fragmentée

Nous venons de voir que considérer le problème de la preuve de l'invariant $\Box P$ en termes d'un point fixe sur l'ensemble des équations était équivalent à le considérer en termes de plusieurs points fixes, portant chacun sur une seule équation. De plus, la formalisation (4.2) nous permettra de dériver une règle d'induction continue fragmentée.

Rappelons ici la règle sous sa forme non fragmentée (3.2) :

$$(\text{cont})\frac{\Box P(\varepsilon) \qquad \forall y \in D^*.\ \Box P(y) \Rightarrow \Box P(f(y))}{\Box P(\mu_x.x = f(x))}$$

Notre problème

$$\Box P(G, F) \ = \ \Box P(G, \mu_y.(y = h(G, y)))$$

se transforme par application de la règle (cont) en deux sous-problèmes :

1. $\Box P(G, \varepsilon)$
   Ce premier but s'écrit également $\Box P(\mu_x.\ x = g(x, H(x)), \varepsilon)$ et se décompose à son tour en deux sous-buts par la règle (cont) :
   (a) $\Box P(\varepsilon, \varepsilon)$
   (b) $\forall x \in D^*.\ \Box P(x, \varepsilon) \Rightarrow \Box P(g(x, H(x)), \varepsilon)$
   À nouveau, ce but se réécrit en $\forall x \in D^*.\ \Box P(x, \varepsilon) \Rightarrow \Box P(g(x, \mu_y.\ y = h(x, y)), \varepsilon)$. On applique une dernière fois la règle d'induction continue :
      i. $\forall x \in D^*.\ \Box P(x, \varepsilon) \Rightarrow \Box P(g(x, \varepsilon), \varepsilon)$
      ii. $\forall x, y \in D^*.\ \Box P(x, \varepsilon) \wedge \Box P(g(x, y), \varepsilon) \Rightarrow \Box P(g(x, h(x, y)), \varepsilon)$
2. $\forall y \in D^*.\ \Box P(G, y) \Rightarrow \Box P(G, h(G, y))$
   Ce deuxième but principal peut se mettre sous la forme suivante :
   $\forall y \in D^*.\ \Box P(G, y) \Rightarrow \Box P(\mu_x.\ x = g(x, H(x)), h(\mu_x.\ x = g(x, H(x)), y))$
   La règle d'induction continue donne à nouveau deux sous-buts :
   (a) $\forall y \in D^*.\ \Box P(G, y) \Rightarrow \Box P(\varepsilon, h(\varepsilon, y))$
   (b) $\forall x, y \in D^*.\ \Box P(G, y) \wedge \Box P(x, h(x, y)) \Rightarrow \Box P(g(x, H(x)), h(g(x, H(x)), y))$
   Réécrite en $\forall x, y \in D^*.\ \Box P(G, y) \wedge \Box P(x, h(x, y)) \Rightarrow \Box P(g(x, \mu_y.\ y = h(x, y)), h(g(x, \mu_y.\ y = h(x, y)), y))$, cette propriété se décompose par (cont) en
      i. $\forall x, y \in D^*.\ \Box P(G, y) \wedge \Box P(x, h(x, y)) \Rightarrow \Box P(g(x, \varepsilon), h(g(x, \varepsilon), y))$
      ii. $\forall x, y, z \in D^*.\ \Box P(G, y) \wedge \Box P(x, h(x, y)) \wedge \Box P(g(x, z), h(g(x, z), y))$
      $\Rightarrow \Box P(g(x, h(x, z)), h(g(x, h(x, z)), y))$

Il existe donc une règle d'induction continue fragmentée telle que

La propriété $\Box P(\mu_{x,y}.\ x, y = g(x, y), h(x, y))$ est transformée en les six obligations de preuve suivantes	
1.	$\Box P(\varepsilon, \varepsilon)$
2.	$\forall x \in D^*.\ \Box P(x, \varepsilon) \Rightarrow \Box P(g(x, \varepsilon), \varepsilon)$
3.	$\forall x, y \in D^*.\ \Box P(x, \varepsilon) \wedge \Box P(g(x, y), \varepsilon) \Rightarrow \Box P(g(x, h(x, y)), \varepsilon)$
4.	$\forall y \in D^*.\ \Box P(G, y) \Rightarrow \Box P(\varepsilon, h(\varepsilon, y))$
5.	$\forall x, y \in D^*.\ \Box P(G, y) \wedge \Box P(x, h(x, y)) \Rightarrow \Box P(g(x, \varepsilon), h(g(x, \varepsilon), y))$
6.	$\forall x, y, z \in D^*.\ \Box P(G, y) \wedge \Box P(x, h(x, y)) \wedge \Box P(g(x, z), h(g(x, z), y))$ $\Rightarrow \Box P(g(x, h(x, z)), h(g(x, h(x, z)), y))$
où	$G = \mu_x.\ (x = g(x, H(x)))$ et $H(x) = \mu_y.\ (y = h(x, y))$

Règle d'induction continue fragmentée

4.3 L'intérêt de l'induction continue fragmentée

Nous pouvons constater que l'utilité pratique d'une règle applicable équation par équation est bien limitée, et ce pour plusieurs raisons :

- Pour un système à deux équations, la règle présentée crée six nouveaux sous-buts. Pour trois équations, ce sont trente-quatre sous-buts, etc. Le passage à l'échelle constitue donc un premier problème.

- Toutes les suites infinies ne sont pas éliminées par cette règle : la suite G subsiste dans les hypothèses des sous-buts 4, 5 et 6. Il se pose alors la question de l'utilisation pratique d'une telle hypothèse. De manière générale, on ne peut guère qu'affaiblir $\Box P(G, y)$ en $\Box P(\varepsilon, y)$[1].
- L'ordre d'application des équations devient important, car il apporte une dissymétrie dans les sous-buts. Ainsi, la fonction h a en quelque sorte « plus d'impact » que g. Il faudrait alors trouver un bon ordre entre les différentes équations pour tirer le maximum de l'application de la règle d'induction fragmentée.
- Enfin, un point particulièrement important est la « désagrégation de la dynamique du système ». En effet, on peut observer dans le but 6 (le but correspondant au cas général), qu'il apparaît une troisième variable indépendante z. La variable y s'est pour ainsi dire dédoublée, si bien que le comportement qui était le sien est maintenant partagé entre y et z. Nous pensons que ce fait limite sérieusement l'intérêt pratique de la méthode, parce qu'il demande de prouver une propriété bien plus générale que celle que nous avions au départ.

Les raisons que nous venons d'exposer nous ont poussé à rechercher une autre manière de renforcer la règle d'induction continue. Le résultat de cette recherche fait l'objet du chapitre suivant.

[1] Comme G est un point fixe, tout préfixe fini de ce point fixe vérifie la même propriété. Mais comment déterminer *a priori* la longueur du préfixe fini qu'il faudra employer ? Le seul choix sensé est celui de préfixe de longueur nulle...

5 – Dépliages en bloc

Nous prouvons l'équivalence entre la preuve d'une propriété invariante et la preuve de la même propriété dépliée n fois. Nous en déduisons une règle d'induction continue généralisée aux dépliages. Se pose alors la question du nombre dépliages : nous donnons une caractérisation des propriétés prouvables par induction et en déduisons une règle pour fixer ce paramètre.

Au chapitre 3, et notamment dans les sections 3.2 et 3.4, nous avons présenté la méthode de preuve inductive développée par Cécile Dumas. Nous avons expliqué ses limites pratiques, dues notamment à l'absence d'une règle correcte qui prenne en compte la dynamique du système mieux que ne le fait la règle de continuité (3.2).

Ensuite, nous avons fait une première proposition d'une règle plus expressive au chapitre 4. En adoptant la démarche initiée par Cécile Dumas, nous avons prolongé son travail et abouti à une règle d'induction continue fragmentée. Cependant, celle-ci s'est révélée peu utile en pratique.

C'est pourquoi nous allons définir une nouvelle règle, censée remplacer et améliorer l'induction continue. Nous allons l'énoncer dans la section 5.1 et nous verrons que son application pratique dépend d'un paramètre. En section 5.2, nous proposerons une manière de fixer la valeur du paramètre à l'utilisation.

Le travail que nous présentons dans ce chapitre a été partiellement mené dans le cadre du projet Next-TTA. Nos résultats ont fait l'objet d'un rapport d'avancement et d'un rapport de recherche Vérimag [33] qui a été présenté dans une version préliminaire au workshop Synchrone en décembre 2004.

5.1 Règle de dépliage

Avant d'arriver à l'énoncé de notre nouvelle règle, nous allons parcourir plusieurs étapes, afin de bien démontrer qu'elle est correcte. Pour fixer les notations, nous allons nous intéresser à la preuve d'une propriété invariante $\Box P$ sur un système Lustre qui définit une fonction f : notre propos sera donc de démontrer que

$$\Box P(\mu_x.\ x = f(x))$$

Un point fixe de $\mathbf{f^n}$ Soit une suite $M \subseteq D^\infty$ telle que

$$M \ \equiv \ \mu_x.\ x = f(x)$$

M est alors le (plus petit) point fixe de l'équation $x = f(x)$, si bien que

$$M = f(M) \tag{5.1}$$

En appliquant f, nous obtenons

$$f(M) = f^2(M)$$

Par transitivité avec (5.1), nous pouvons conclure que

$$M = f^2(M)$$

En fait, un raisonnement inductif très simple nous permettrait de généraliser le résultat précédent à

$$\forall n \in \mathbb{N}.\ M = f^n(M) \qquad (\text{avec } f^0(M) = M) \tag{5.2}$$

Ainsi, M est un point fixe de l'équation $x = f^n(x)$, pour tout n.

Le plus petit point fixe de $\mathbf{f^n}$ Cependant, nous pouvons caractériser M encore plus précisément par rapport à f^n. D'abord, rappelons qu'en tant que plus petit point fixe de l'équation $x = f(x)$, M est défini grâce au théorème de Kleene (cf. page 24) comme

$$M = \sup\{\varepsilon, f(\varepsilon), \ldots, f^k(\varepsilon), \ldots\} \tag{5.3}$$

En vertu de ce même théorème, le plus petit point fixe de l'équation $x = f^n(x)$, pour un n donné, est $\mu_n \in D^\infty$ tel que

$$\mu_n = \sup\{\varepsilon, f^n(\varepsilon), \ldots, f^{nk}(\varepsilon), \ldots\} \tag{5.4}$$

Or $C_1 = \{\varepsilon, f(\varepsilon), \ldots, f^k(\varepsilon), \ldots\}$ et $C_n = \{\varepsilon, f^n(\varepsilon), \ldots, f^{nk}(\varepsilon), \ldots\}$ sont des chaînes croissantes :

$$\varepsilon \preceq f(\varepsilon) \preceq \ldots \preceq f^k(\varepsilon) \preceq \ldots$$
$$\varepsilon \preceq f^n(\varepsilon) \preceq \ldots \preceq f^{nk}(\varepsilon) \preceq \ldots$$

donc

$$\forall x \in C_1.\ \exists y \in C_n.\ x \preceq y \tag{5.5}$$

(En effet, $f^i(\varepsilon) \in C_1$ est majoré par $f^{(n+1)\lfloor i/n \rfloor}(\varepsilon) \in C_n$.) De (5.3), (5.4) et (5.5), on déduit que

$$M \preceq \mu_n \tag{5.6}$$

Les deux propriétés (5.2) et (5.6) permettent de conclure que M est non seulement un point fixe de $x = f^n(x)$, mais que c'est également *le plus petit*, pour un n quelconque : $M = \mu_n$.

Dépliages Du point de vue logique, prouver la propriété $\Box P(\mu_x.\ x = f(x))$ est donc *équivalent* à prouver $\Box P(\mu_x.\ x = f^n(x))$ pour $n > 1$. En fait, le problème initial est même équivalent à prouver la conjonction de ces deux propriétés : $\Box P(\mu_x.\ x = f(x)) \wedge \Box P(\mu_x.\ x = f^n(x))$.

Mieux encore, pour $n > 1$, le problème de preuve de $\Box P(\mu_x.\ x = f(x))$ est logiquement équivalent à la preuve de

$$\Box P(\mu_x.\ x = f(x)) \wedge \Box P(\mu_x.\ x = f^2(x)) \wedge \ldots \wedge \Box P(\mu_x.\ x = f^n(x)) \tag{5.7}$$

Nous dirons que nous avons *déplié* le système n fois en une forme *logiquement* équivalente. Cependant, nous verrons un peu plus loin que du point de vue *pratique*, la nouvelle forme est plus intéressante.

La propriété dépliée peut être reformulée en

$$\Box(P(\mu_x.\ x = f(x)) \wedge P(\mu_x.\ x = f^2(x)) \wedge \ldots \wedge P(\mu_x.\ x = f^n(x)))$$

ce qui nous permet d'appliquer la règle d'induction continue (3.2) et d'obtenir

$$\frac{\begin{array}{l}\cdot\ \ \Box(P(\varepsilon) \wedge P(f(\varepsilon)) \wedge \ldots \wedge P(f^{n-1}(\varepsilon)))\\ \cdot\ \ \forall x \in D^*.\Box(P(x) \wedge P(f(x)) \wedge \ldots \wedge P(f^{n-1}(x))) \Rightarrow \Box(P(f(x)) \wedge P(f^2(x)) \wedge \ldots \wedge P(f^n(x)))\end{array}}{\Box(P(\mu_x.\ x = f(x)) \wedge P(\mu_x.\ x = f^2(x)) \wedge \ldots \wedge P(\mu_x.\ x = f^n(x)))}$$

Nous créons ainsi deux nouveaux sous-buts. Le premier peut se mettre sous la forme

$$\Box P(\varepsilon) \wedge \Box P(f(\varepsilon)) \wedge \ldots \wedge \Box P(f^{n-1}(\varepsilon))$$

et le deuxième est équivalent à

$$\forall x \in D^*.\Box P(x) \wedge \Box P(f(x)) \wedge \ldots \wedge \Box P(f^{n-1}(x)) \Rightarrow \Box P(f(x)) \wedge \Box P(f^2(x)) \wedge \ldots \wedge \Box P(f^n(x))$$

ce qui se simplifie en

$$\forall x \in D^*.\Box P(x) \wedge \Box P(f(x)) \wedge \ldots \wedge \Box P(f^{n-1}(x)) \Rightarrow \Box P(f^n(x))$$

Règle de dépliage Nous pouvons alors formuler une nouvelle règle d'induction continue, la règle de dépliage[1] de rang n :

$$(\text{unfold}_n)\ \frac{\begin{array}{l}\Box P(\varepsilon)\\ \Box P(f(\varepsilon))\\ \quad\ldots\\ \Box P(f^{n-1}(\varepsilon))\\ \forall x \in D^*.\Box P(x) \wedge \Box P(f(x)) \wedge \ldots \wedge \Box P(f^{n-1}(x)) \Rightarrow \Box P(f^n(x))\end{array}}{\Box(P(\mu_x.\ x = f(x))}$$

Règle de dépliage au rang n

L'intérêt principal de cette règle est de faire figurer, dans l'hypothèse du cas général, la propriété sur les n premiers dépliages du système et de demander de l'établir sur le $n+1^{\text{ème}}$: les n premiers éléments de la dynamique du système (les n premières mémoires) sont ainsi présentes dans les hypothèses. C'est une amélioration par rapport à la règle d'induction continue de base, qui ne faisait figurer dans l'hypothèse que la toute première étape. En fait, la règle de dépliage est une généralisation de la règle de continuité : le dépliage au rang 1 est exactement la règle de continuité.

Exemple de Fibonacci Dans la section 3.5.1, nous avons utilisé l'exemple 3.3 de la suite de Fibonacci pour illustrer les limites de la règle de continuité. Profitons-en maintenant pour montrer l'utilité de la nouvelle règle.

En effet, en appliquant la règle (unfold_2) sur le même exemple, nous obtenons trois buts :

1. Le cas de base « $\Box P(f(\varepsilon))$ »

   $$\frac{}{\Box(\varepsilon > 0)}$$

   qui est trivialement vrai.

2. Le cas de base « $\Box P(f(\varepsilon))$ »

   $$\frac{}{\Box(1.(\varepsilon + \varepsilon) > 0)}$$

   qui donne une seule obligation de preuve scalaire pour l'utilisateur

   $$(\text{OP}_0)\ \frac{}{1 > 0}$$

[1] « unfolding » en anglais

3. Le cas général « $\forall x \in D^*.\Box P(x) \wedge \Box P(f(x)) \Rightarrow \Box P(f^2(x))$ » qui donne, après induction, skolémisation et simplification

$$\frac{\Box(f > 0) \qquad \Box(1.(f+g) > 0) \qquad \Box(1.(tl(f)+tl(g)) > 0 \Rightarrow \Box(1.(1.(tl(f)+tl(g))+0.tl(f)) > 0)}{\Box(1.(1.(f+g)+0.f) > 0)}$$

En appliquant les règles d'avancement, on obtient au total trois obligations de preuve scalaires, avant d'arriver au cas inductif :

$$(\mathrm{OP}_1)\ \frac{1 > 0}{1 > 0} \qquad (\mathrm{OP}_2)\ \frac{1 > 0}{1 + 0 > 0} \qquad (\mathrm{OP}_3)\ \frac{hd(f) > 0 \qquad hd(f) + hd(g) > 0}{hd(f) + hd(g) + hd(f) > 0}$$

Nous pouvons constater que toutes les obligations de preuve sont (facilement) prouvables, si bien que l'utilisateur n'aura pas de difficulté à établir la propriété voulue.

Utilisation pratique Nous venons de voir que la règle de dépliage permet en effet de prouver davantage de propriétés que la simple règle d'induction continue. Mais son utilisation pratique soulève la question du rang de la règle à appliquer : dans l'exemple de Fibonacci, nous avons fixé $n = 2$ sans aucune explication. Y a-t-il un procédé pour déterminer *a priori* ce paramètre, dans le cas d'un système donné ? Peut-il y avoir un tel procédé ? Nous allons répondre à ces questions dans la section suivante.

5.2 Combien de dépliages ?

La règle de dépliage proposée à la section précédente est basée sur l'induction. Comme telle, elle est correcte, mais elle ne saurait être complète : certaines propriétés vraies ne peuvent pas être prouvées par cette méthode. Aussi, nous n'allons pas chercher *le* bon nombre de dépliages à appliquer, car il pourrait très bien ne pas exister ; nous nous proposons de trouver un nombre de dépliages *optimum*, c'est-à-dire ayant de bonnes chances de marcher, tout en n'étant pas trop élevé.

5.2.1 Formalisation

Le raisonnement que nous allons présenter s'applique à une classe de systèmes plus large que les systèmes écrits en Lustre. C'est pourquoi nous nous placerons dans ce cadre plus général, ce qui nous permettra entre autres d'utiliser des notations plus simples.

Systèmes Nous allons considérer des systèmes fermés, c'est-à-dire sans entrées. Ainsi, nous appellerons « système » l'ensemble d'un programme *et* de l'environnement avec lequel il interagit. Cette inclusion de l'environnement dans le modèle n'est pas superflue, car les propriétés les plus intéressantes des systèmes réactifs ne se vérifient que dans une boucle d'interaction avec un environnement donné.

Formellement, posons

- S l'espace des états. S est le produit cartésien de sous-espaces associés aux m variables du système : $S = \prod_{i=1}^{m} S_i$.
- I_0 l'ensemble des états initiaux potentiels : $\emptyset \subset I_0 \subseteq S$.
- T la relation de transition : $T \subseteq (S \times S)$. Nous allons supposer que T est *totale*, c'est-à-dire que[2] $dom(T) = S$. Ainsi, nous ne nous intéresserons pas aux dead-locks.

[2] $dom(T) = \{x \in S \mid \exists y \in S.T(x,y)\}$

Dans ce cadre, l'ensemble des états accessibles du système (S, I_0, T) est I_∞, le plus petit point fixe vérifiant

$$I_\infty = I_0 \cup T(I_\infty)$$

où $T(X) = \{y \mid \exists x \in X.\, T(x, y)\}$, pour $X \subseteq S$. Une manière équivalente de définir I_∞ est

$$I_\infty = \inf\{X \mid I_0 \cup T(X) \subseteq X\} = \bigcap\{X \mid I_0 \cup T(X) \subseteq X\} \tag{5.8}$$

Propriétés et induction La propriété que nous voulons démontrer est donnée par un prédicat P sur S : $P \subseteq S$. P est un invariant de (S, I_0, T) si

$$I_\infty \subseteq P \tag{5.9}$$

Le principe d'induction affirme que

$$I_0 \cup T(P) \subseteq P \tag{5.10}$$

est suffisant pour prouver (5.9). Dans la suite, nous allons nous référer à l'ensemble du système et de la propriété par la notation raccourcie (S, I_0, T, P).

L'induction et l'image inverse L'image inverse de la relation totale $T \subseteq S \times S$ est définie par

$$T^-(Y) = \sup\{X \mid T(X) \subseteq Y\} = \bigcup\{X \mid T(X) \subseteq Y\}$$

On peut facilement montrer que les images et les images inverses forment des connexions de Galois :

$$\begin{array}{rcl} Id & \subseteq & T^- \circ T \\ T \circ T^- & = & Id \end{array} \tag{5.11}$$

où Id est l'identité sur S.

Nous nous intéressons aux images inverses, parce que nous pouvons établir un parallèle avec les dépliages de la section 5.1. En fait, les dépliages de μ_f, point fixe de l'équation $x = f(x)$, correspondent aux images inverses de μ_f :

$$\begin{array}{rcl} f^-(\mu_f) & = & \sup\{s \mid f(s) \preceq \mu_f\} \\ & = & \mu_f \\ & = & f(\mu_f) \end{array}$$

Le parallèle entre les dépliages et les images inverses ne s'arrête pas là. En fait, on retrouve l'équivalence logique de la preuve de base et de la preuve dépliée dans le cas des systèmes (S, I_0, T, P) :

- si $I_\infty \subseteq P$
- · comme $I_\infty = I_0 \cup T(I_\infty)$, on a également $T(I_\infty) \subseteq P$
- · ce qui donne $I_\infty \subseteq T^-(P)$ en vertu de (5.11)
- ▷ et donc $I_\infty \subseteq P \cap T^-(P)$
- inversement $I_\infty \subseteq P \cap T^-(P)$ donne $I_\infty \subseteq P$

Prouver (S, I_0, T, P) est donc logiquement équivalent à prouver $(S, I_0, T, P \cap T^-(P))$. En fait, un court raisonnement par récurrence montrerait que le problème est équivalent à

$$(S, I_0, T, \bigcap_{i=0}^{n-1} T^{-i}(P))$$

pour tout $n > 0$.

La question du nombre de dépliages optimum se réduit donc au choix d'une « bonne » valeur pour le paramètre n dans la formule ci-dessus. Pour trouver cette valeur, nous allons maintenant caractériser plus précisément les systèmes où (i) P est un invariant (5.9 est vrai) **et** (ii) ce fait peut être prouvé par une application du principe d'induction (5.10) sur une instance dépliée n fois.

5.2.2 Caractérisation des invariants prouvables par n dépliages

Pour trouver la caractérisation recherchée, nous allons procéder en plusieurs étapes, qui nous permettront de construire un raisonnement par équivalences. Dans un premier temps, nous allons démontrer que certains changements de variables conservent les preuves inductives. Ensuite, nous allons mettre à profit cette propriété pour établir des changements de variables qui donnent une caractérisation exacte des propriétés prouvables par induction avec 1, 2 et n dépliages. Finalement, nous en déduirons la règle du nombre de dépliages optimum.

a) Conservation de la preuve inductive par abstraction

Étant donnés deux ensembles S et S', un changement de variables est une fonction $G : S \to S'$. On peut naturellement étendre une telle fonction aux ensembles $(G : \mathcal{P}(S) \to \mathcal{P}(S'))$, ce qui permet de définir son image inverse G^- :

$$G(X) = \{y \in S' \mid \exists x \in S.G(x, y)\} \qquad \text{et} \qquad G^-(Y) = \bigcup\{X \subseteq S \mid G(X) \subseteq Y\}$$

Le changement de variables G génère un nouveau système $(S', I'_0, T', P') = G(S, I_0, T, P)$ tel que

$$I'_0 = G(I_0) \quad , \quad P' = G(P) \qquad \text{et} \qquad T' = G \circ T \circ G^-$$

En général, un changement de variables ne préserve pas les propriétés. Cependant, on trouve dans [30, 15] le théorème suivant :

Théorème 1 (Théorème de l'abstraction) *Soit P vérifiant*

$$P = G^- \circ G(P) \tag{5.12}$$

Si P' est un invariant de (S', I'_0, T', P'), c'est-à-dire si $I'_\infty \subseteq P'$, alors P est un invariant de (S, I_0, T, P) : $I_\infty \subseteq P$.

De plus, en utilisant (5.11) et (5.12), on peut facilement démontrer que pour tout ensemble $X \subseteq S$,

$$G(X) \subseteq G(P) \quad \Leftrightarrow \quad X \subseteq P \tag{5.13}$$
$$\text{and } T'(G(P)) \subseteq G(P) \quad \Leftrightarrow \quad T(P) \subseteq P \tag{5.14}$$

Ces propriétés nous permettent d'énoncer la proposition suivante :

Théorème 2 (Conservation de preuves inductives) *Soit P tel que $P = G^- \circ G(P)$. Alors, il existe une preuve inductive de (S, I_0, T, P) si et seulement si il existe une preuve inductive de (S', I'_0, T', P').*

En effet,

	il existe une preuve inductive de (S, I_0, T, P)	
$\equiv$	$I_0 \subseteq P$ et $T(P) \subseteq P$	
$\equiv$	$G(I_0) \subseteq G(P)$ et $T(P) \subseteq P$	en appliquant (5.13) à I_0
$\equiv$	$G(I_0) \subseteq G(P)$ et $T'(G(P)) \subseteq G(P)$	en appliquant (5.14)
$\equiv$	$I'_0 \subseteq P'$ et $T'(P') \subseteq P'$	
$\equiv$	il existe une preuve inductive de (S', I'_0, T', P')	

□

b) Abstraction associée à un dépliage

Étant donné un système (S, I_0, T, P) et le domaine booléen $\mathbb{B}$, nous définissons le changement de variables G_1 tel que

$$\begin{array}{rcl} G_1 & : & S \rightarrow \mathbb{B} \\ G_1(x) & = & (x \in P) \end{array}$$

Or $\sup\{X \subseteq S \mid G_1(X) \subseteq G_1(P)\} = P$, donc $G_1^- \circ G_1(P) = P$. Par conséquent, P et G_1 vérifient l'hypothèse du théorème 2.

De plus, nous avons une équivalence intéressante dans le nouveau système :

Lemme 1 *Dans $G_1(S, I_0, T, P) = (\mathbb{B}, I'_0, T', P')$, P' est un invariant si et seulement si cela est prouvable par induction.*

En effet, d'une part, l'induction est une règle correcte, si bien que

$$I'_0 \subseteq P' \text{ et } T'(P') \subseteq P' \quad \text{entraîne} \quad I'_\infty = \inf\{X \mid I'_0 \cup T'(X) \subseteq X\} \subseteq P'$$

et, d'autre part, si $I'_\infty \subseteq P'$ alors

- comme $I'_\infty = I'_0 \cup T'(I'_\infty)$, on a $\underline{I'_0 \subseteq P'}$
- ensuite, vu que $\emptyset \subset I_0$, alors nécessairement $\emptyset \subset I'_0$
  - ▷ d'où $\emptyset \subset I'_0 \subseteq P' \subseteq \{\mathtt{t}\}$ et donc $I'_0 = \{\mathtt{t}\} = P'$
- comme $I'_\infty = I'_0 \cup T'(I'_\infty) = I'_0 \cup T'(I'_0 \cup T'(I'_\infty))$
  et $T'(I'_0 \cup T'(I'_\infty)) = T'(I'_0) \cup T' \circ T'(I'_\infty)$
  on a aussi $T'(I'_0) \subseteq P'$
  - ▷ combiné avec $I'_0 = P'$, on conclut que $\underline{T'(P') \subseteq P'}$

□

Ce lemme nous permet de donner la caractérisation suivante :

Théorème 3 *Dans (S, I_0, T, P), P est prouvable par induction si et seulement si P' est un invariant de $G_1(S, I_0, T, P)$.*

La justification est presque immédiate grâce aux propositions précédentes :

$$\begin{array}{rll} & P \text{ peut être prouvée par induction} & \\ \equiv & I_0 \subseteq P \text{ et } T(P) \subseteq P & \\ \equiv & I'_0 \subseteq P' \text{ et } T'(P') \subseteq P' & \text{(théorème 2)} \\ \equiv & I'_\infty \subseteq P' & \text{(lemme 1)} \end{array}$$

□

c) Abstraction associée à deux dépliages

Soit le changement de variables G_2 tel que

$$\begin{array}{rcl} G_2 & : & S \rightarrow \mathbb{B} \times \mathbb{B} \\ G_2(x) & = & (x \in P, x \in T^-(P)) \end{array}$$

Il est facile de montrer que G_2 et $P \cap T^-(P)$ vérifient l'hypothèse du théorème 2 : $G_2^- \circ G_2(P \cap T^-(P)) = P \cap T^-(P)$. Comme précédemment, nous pouvons établir une équivalence entre la valeur de vérité et la possibilité de prouver une propriété dans le nouveau système :

Lemme 2 *Dans $G_2(S, I_0, T, P \cap T^-(P))$, la propriété $P' \cap T'^-(P)$ est un invariant si et seulement si elle est prouvable par induction.*

En effet,

– l'induction est correcte, si bien que

$$I'_0 \subseteq P' \cap T'^-(P') \quad \text{et} \quad T'(P' \cap T'^-(P')) \subseteq P' \cap T'^-(P') \quad \text{implique} \quad I'_\infty \subseteq P' \cap T'^-(P')$$

– inversement, si $I'_\infty \subseteq P' \cap T'^-(P')$, alors

- comme $I'_\infty = I'_0 \cup T'(I'_\infty)$ on déduit $\underline{I'_0 \subseteq P' \cap T'^-(P')}$
- puis $\emptyset \subset I_0$ donne $\emptyset \subset I'_0$
- ▷ et par conséquent $\emptyset \subset I'_0 \subseteq P' \cap T'^-(P') \subseteq \{(\mathtt{t}, \mathtt{t})\}$
  d'où $I'_0 = P' \cap T'^-(P') = \{(\mathtt{t}, \mathtt{t})\}$
- par ailleurs $I'_\infty = I'_0 \cup T'(I'_\infty) = I'_0 \cup T'(I'_0 \cup T'(I'_\infty))$
  et on peut décomposer $T'(I'_0 \cup T'(I'_\infty)) = T'(I'_0) \cup T' \circ T'(I'_\infty)$
  si bien que $T'(I'_0) \subseteq P' \cap T'^-(P')$
- ▷ et on conclut avec $\underline{T'(P' \cap T'^-(P')) \subseteq P' \cap T'^-(P')}$

□

De ce lemme, nous tirons une nouvelle caractérisation :

Théorème 4 *Dans (S, I_0, T, P), $P \cap T^-(P)$ est prouvable par induction si et seulement si $P' \cap T'^-(P')$ est un invariant de $G_2(S, I_0, T, P)$.*

Preuve :

$$\begin{array}{lll} & P \cap T^-(P) \text{ peut être prouvée par induction} & \\ \equiv & I_0 \subseteq P \cap T^-(P) \text{ et } T(P \cap T^-(P)) \subseteq P \cap T^-(P) & \\ \equiv & I'_0 \subseteq P' \cap T'^-(P') \text{ and } T'(P' \cap T'^-(P')) \subseteq P' \cap T'^-(P') & \text{(théorème 2)} \\ \equiv & I'_\infty \subseteq P' \cap T'^-(P') & \text{(lemme 2)} \end{array}$$

□

d) Généralisation et énoncé de la règle

Par induction, nous généralisons les théorèmes 3 et 4 en :

Théorème 5 (Caractérisation des dépliages) *Étant donné le changement de variables*

$$\begin{array}{rcl} G_i & : & S \to \mathbb{B}^i \\ G_i(x) & = & (x \in P, \ldots, x \in T^{-(i-1)}(P)) \end{array}$$

la propriété $\bigcap_{j=0}^{i-1} T^{-j}(P)$ peut être prouvée par induction si et seulement si $G_i(S, I_0, T, P)$ est vrai.

Nous venons d'établir une équivalence entre d'un côté la possibilité de prouver par induction une propriété dépliée i fois ($i > 0$) et de l'autre côté la valeur de vérité de cette propriété dans l'abstraction donnée par le changement de variables G_i.

Or nous avons supposé dès le début que S, l'ensemble des états du système, était produit cartésien de m sous-espaces associés aux variables du système. Ainsi, il faudrait effectuer au moins m dépliages, pour que chaque dimension du système soit représentée par une dimension dans l'abstraction associée. En effet, si $i < m$, alors il n'existe pas d'injection de $S = \prod_{j=1}^{m} S_j$ dans $\mathbb{B}^i$, si bien que prouver une

propriété P dépliée seulement i fois ($i < m$) revient à prouver cette propriété pour toute une classe de systèmes, classe caractérisée par son image par G_i. Mais nous voulons au contraire obtenir une preuve spécifique au système donné, plutôt qu'à tous les systèmes d'une classe : c'est pourquoi il faut effectuer au moins m dépliages. De l'autre côté, chaque nouveau dépliage augmente la taille de la preuve à faire, si bien qu'en pratique, il faut utiliser le moins de dépliages possibles. Étant données ces deux contraintes, nous pouvons énoncer la règle du nombre de dépliages optimum :

Règle du nombre de dépliages Dans un système fermé ayant m variables d'états, le choix optimum pour prouver qu'une propriété P est un invariant du système est celui d'effectuer m dépliages.

Il est bien connu que la suite de Fibonacci est linéaire dans $\mathbb{R}^2$, c'est-à-dire qu'elle a deux variables d'état. C'est pourquoi le dépliage à l'ordre 1 (la règle de continuité de base) n'a pas été suffisant, tandis que le dépliage à l'ordre 2 a permis de prouver la propriété recherchée (cf. exemples page 36 et 57).

5.3 Conclusion

Dans la première partie de ce chapitre, nous avons proposé une règle de dépliages en bloc, destinée à remplacer et à généraliser la règle de continuité. Nous avons prouvé que cette nouvelle règle était correcte et qu'en pratique, elle permettait de prouver davantage de propriétés que l'induction continue de base.

Dans un deuxième temps, nous avons proposé une règle pour déterminer *a priori* le nombre de dépliages à appliquer dans le cas d'un système donné. Nous avons déduit ce nombre *optimum* d'une caratérisation des propriétés prouvables par induction en fonction du nombre de dépliages.

Ainsi, nous disposons d'une nouvelle règle, plus expressive que l'induction continue de base et effectivement applicable en pratique, mais sans les inconvénients que nous avons relevés pour la règle d'induction fragmentée (cf. section 4.3) :

- Le nombre de sous-buts générés par la règle de dépliages en bloc est linéaire en le nombre de dépliages effectués. En particulier, le nombre d'équations qui décrivent le système n'influe aucunement sur le nombre de sous-buts.
- Seules les suites finies figurent dans les sous-buts générés, que ce soit dans les hypothèses ou dans les conclusions. Ainsi, nous retrouvons bien l'esprit de l'induction continue de base, qui permet précisément ce passage du domaine des suites infinies vers domaine des suites finies.
- Toutes les équations du système étant dépliées en même temps, leur ordre n'a pas d'importance. C'est bien ce que nous attendons dans un cadre fonctionnel.
- Et finalement, la dynamique du système est fidèlement restituée dans tous les sous-buts. Le fait de déplier apporte des précisions sur le comportement du système : on n'introduit pas de généralisation, comme c'était le cas pour la règle fragmentée.

Pour pouvoir exploiter en pratique la méthode de preuve fondée sur les règles originelles de Cécile Dumas et sur la règle de dépliages en bloc, nous avons développé l'outil Gloups, présenté au chapitre suivant. Gloups nous permettra de compléter la famille des outils de vérification pour Lustre dans le domaine de la preuve interactive.

6 – Outil de preuve Gloups

Cette présentation de l'outil de preuve Gloups procède du général au particulier. D'abord, nous exposons nos motivations pour implémenter une nouvelle version de cet outil et nous proposons un tour du propriétaire de la nouvelle version. Ensuite, nous expliquons l'algorithme de preuve utilisé, ainsi que la forme générale des obligations produites. Puis, nous détaillons deux optimisations importantes qui nous ont permis de repousser les limites de notre logiciel. Pour finir, nous proposons un exemple d'application réel.

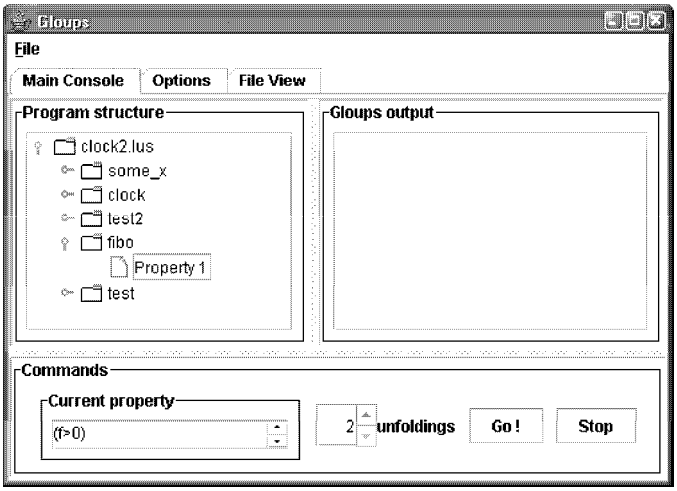


FIG. 6.1 – Interface graphique de Gloups

6.1 Présentation générale

Pour commencer, nous allons retracer l'historique de Gloups et expliquer pourquoi nous avons jugé nécessaire d'en implémenter une nouvelle version. Ensuite, nous donnerons un rapide aperçu de notre logiciel sur un exemple simple.

6.1.1 Motivations

Une première version de Gloups a été implémentée par Cécile Dumas pendant sa thèse. C'est avec cet outil que nous avons effectué nos premières expérimentations, au cours desquelles nous avons découvert

que Gloups permettait de « prouver » des propriétés fausses. Ce constat nous a poussé à reprendre les fondements théoriques de la méthode, et nous avons fini par trouver que la règle d'induction continue fragmentée n'était pas correcte[1].

Après avoir élaboré une correction théorique, exposée au chapitre 5, nous avons également voulu corriger Gloups. Néanmoins, un certain nombre de considérations nous a dissuadé d'effectuer une simple correction du logiciel existant :

- La version 1 de Gloups était écrite en Caml, langage fonctionnel proche de ML, qui nous était peu familier.
- Lors de la recherche initiale des raisons, qui permettait à Gloups de « prouver » des propriétés fausses, nous avons pu constater que le code de l'outil était très peu commenté, compliqué et que le découpage en modules était loin d'être idéal.
- En particulier, la mise en pratique de la règle incorrecte était littéralement « vissée » dans le logiciel, si bien que le remplacement par une autre règle nous apparaissait assez compliqué et de résultat incertain.

Pour toutes ces raisons, nous avons décidé de développer une nouvelle version de Gloups, fondée sur des bases différentes :

- Nous avons choisi d'utiliser la programmation à objets. D'une part, ce paradigme permet de voir le même objet sous plusieurs angles différents et de lui faire jouer plusieurs rôles distincts dans les calculs. Nous allons détailler ce point en section 6.4.1, où nous expliquerons comment nous en avons profité pour réduire l'utilisation de la mémoire. D'autre part, la notion d'héritage permet très naturellement d'implémenter une même fonctionnalité de plusieurs manières différentes, la manière effectivement utilisée étant choisie seulement à l'exécution. Cet aspect nous a permis d'expérimenter très facilement plusieurs approches de certains points cruciaux du logiciel, avant d'en choisir une, ou même, dans certains cas, de laisser le choix à l'utilisateur de l'outil.
- Il nous a paru important d'effectuer un découpage du logiciel en modules fonctionnels, aux rôles bien définis. L'avantage de cette approche est la possibilité d'activer ou non certains modules à l'exécution : ainsi, nous pouvons effectuer des analyses plus ou moins poussées, désactiver les optimisations, ou choisir entre utiliser l'interface graphique ou la ligne de commande.
- Enfin, nous avons souhaité avoir un logiciel portable, afin de pouvoir l'ouvrir à un maximum d'utilisateurs.

Compte tenu de ces remarques, nous avons opté pour le langage Java.

6.1.2 Brève présentation

Avant d'aborder le détail de l'implémentation, nous proposons au lecteur un rapide tour du logiciel, en images. Nous allons nous concentrer sur la version avec l'interface graphique, sachant que la ligne de commande permet d'effectuer la plupart des opérations présentées.

En entrée, Gloups attend un fichier en pseudo-Lustre. En fait, nous avons élargi la syntaxe des nœuds avec une nouvelle construction qui s'ajoute aux équations et aux assertions :

à prouver : `prove` <*expression*> ;

Une clause `prove` correspond à une propriété qu'on aimerait prouver, comme dans l'exemple 6.1.

[1] cf. section 3.5.2

```
node fibo(rien:int)
returns (f:int);
var g:int ;
let
  f = 1 -> pre (f+g) ;
  g = 0 -> pre f ;
  prove (f > 0) ;
tel;
```

Exemple 6.1 Fichier d'entrée de Gloups

Après l'ouverture du fichier (figure 6.2), deux cas peuvent se présenter : ou bien la syntaxe du fichier est incorrecte, au quel cas un surlignage du code source permet de retrouver les derniers lexèmes lus et de déterminer le lexème fautif[2] (figure 6.3), ou bien le fichier est correctement analysé et on affiche la structure arborescente des nœuds et des propriétés à prouver associées (figures 6.4 et 6.1).

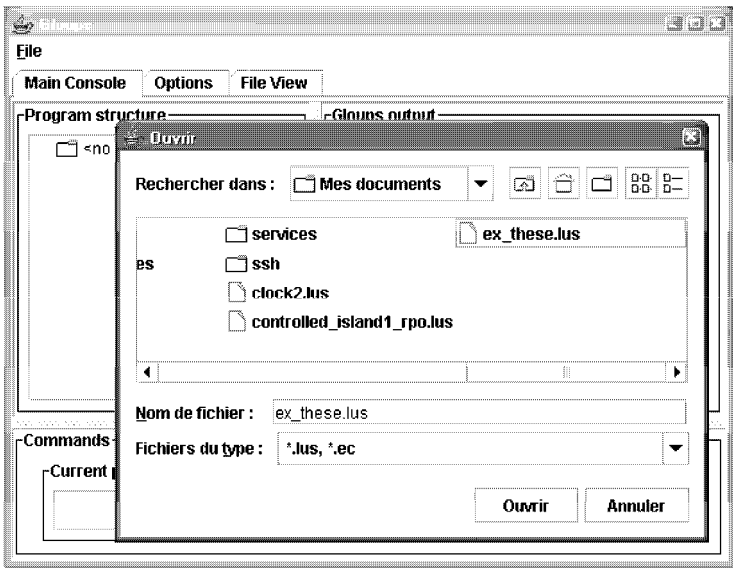


FIG. 6.2 – Ouverture d'un fichier Lustre

On peut remarquer que l'outil suggère le nombre de dépliages à appliquer (deux sur la figure 6.4), mais l'utilisateur peut changer ce paramètre à sa guise. Pour un contrôle encore plus fin du processus de preuve, on dispose également d'un panneau d'options avancées (figure 6.5) que nous détaillerons dans la suite.

Enfin, lorsque tous les réglages sont effectués, le bouton Go ! lance le processus de preuve. Gloups recherche alors une preuve inductive de la propriété voulue, en déchargeant des obligations de preuve scalaires vers PVS.

Dans le cas de l'exemple 6.1, Gloups produit quatre obligations de preuve. Nous reproduisons les fichiers PVS correspondants en annexe C et donnons ici des formules équivalentes :

[2]Dans l'exemple, une erreur de frappe a remplacé l'opérateur -> par -<.

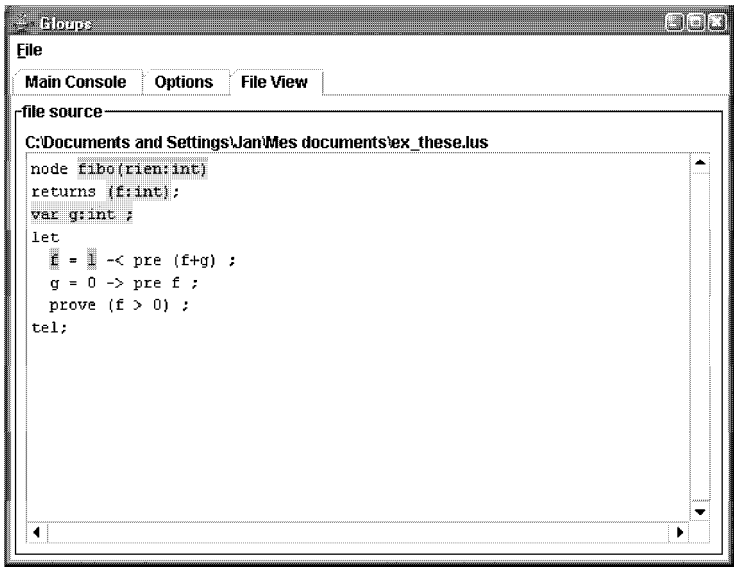


FIG. 6.3 – Exemple d'erreur lexicale

$$\frac{f_{2,0} = 1}{f_{2,0} > 0} \qquad \frac{f_{2,2} = f_{1,1} + g_{1,1} \quad f_{1,1} > 0 \quad g_{1,1} = f_{0,0} \quad f_{0,0} > 0}{f_{2,2} > 0}$$

$$\frac{f_{2,1} = f_{1,0} + g_{1,0} \quad f_{1,0} = 1 \quad g_{1,0} = 0}{f_{2,1} > 0} \qquad \frac{f_{2,3} = f_{1,2} + g_{1,2} \quad f_{1,2} > 0 \quad g_{1,2} = f_{0,1} \quad f_{0,1} > 0}{f_{2,3} > 0}$$

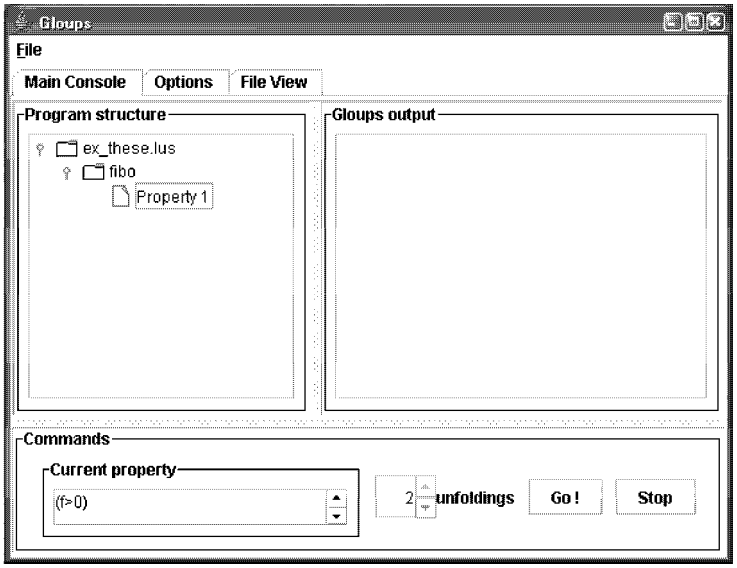


FIG. 6.4 – Structure arborescente d'un fichier

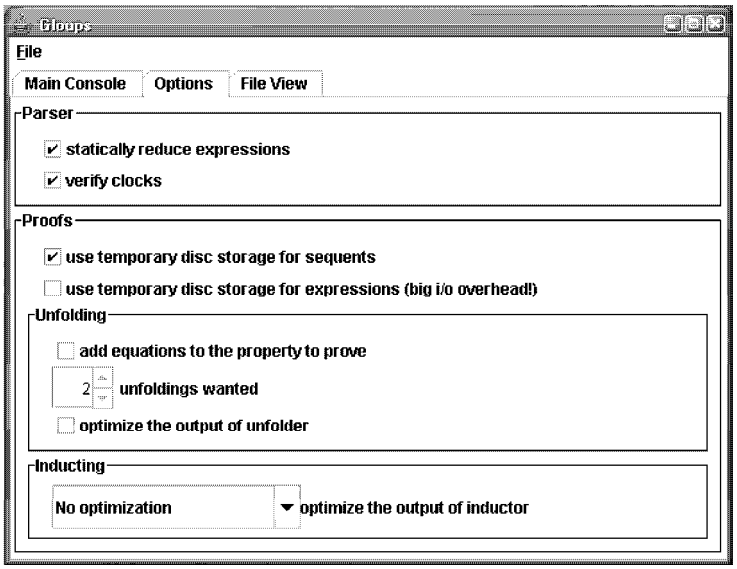


FIG. 6.5 – Panneau d'options

6.2 Algorithme utilisé

Nous avons vu dans notre présentation informelle que l'utilisateur de Gloups commence par ouvrir un fichier Lustre, à l'intérieur duquel il choisit un nœud, puis une propriété à prouver. Après un éventuel réglage d'options, il met en marche le processus de preuve. À partir de ce moment-là, le logiciel agit de manière entièrement automatique. Nous allons maintenant décrire et expliquer l'algorithme de preuve que nous avons implémenté, en nous inspirant de celui que Cécile Dumas avait utilisé dans la première version de Gloups.

6.2.1 Dépliages et généralisations

Dans un premier temps, on élimine les éventuels opérateurs `current` présents dans la propriété à prouver. Pour ce faire, on applique la règle de généralisation[3] (gen) : comme nous l'avons vu à la section 3.4.1, cette règle gère également les opérateurs `when`. Ensuite, c'est à la règle de dépliage en bloc[4] (unfold_n) d'être appliquée : on effectue le nombre de dépliages spécifié par l'utilisateur. Pour finir cette partie, nous ré-appliquons éventuellement la règle de généralisation, car le dépliage a pu introduire de nouveaux `current` dans le but. La figure 6.6 résume cet algorithme.

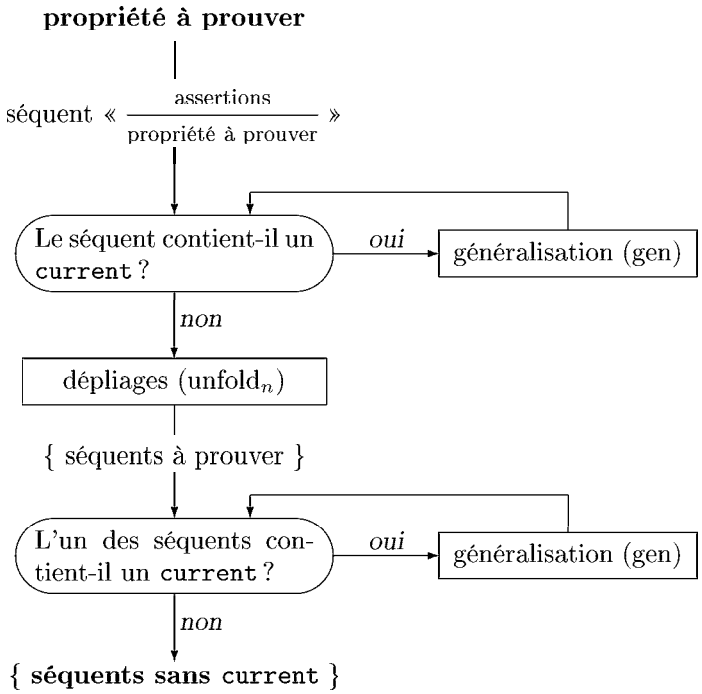


FIG. 6.6 – La phase de dépliages–généralisations

Au terme de cette première phase, la propriété de départ, portant sur des suites infinies, a engendré un ou plusieurs séquents dont tous les éléments sont des suites finies. Nous profitons de ce fait pour représenter les séquents sous une forme skolémisée, car il ne peut pas y avoir d'incertitude sur le typage

[3] généralisation : cf. section 3.4.1, page 35
[4] dépliage en bloc : cf. section 5.1, page 57

à ce stade. De même, nous pouvons omettre la modalité $\Box$ (« à tout instant »), parce qu'elle concerne implicitement toutes les formules.

Pour prendre un exemple, revenons à la suite de Fibonacci. La propriété de départ (écrite sans la modalité $\Box$),

$$f > 0$$

engendre trois séquents par le règle (unfold$_2$) :

$$(\text{seq}_1)\frac{}{\varepsilon > 0} \qquad (\text{seq}_2)\frac{}{1.(\varepsilon + \varepsilon) > 0} \qquad (\text{seq}_3)\frac{f > 0 \qquad 1.(f+g) > 0}{1.(1.(f+g) + 0.f) > 0}$$

Les deux premiers séquents sont des cas d'initialisation, ou « ε-cas ». Le troisième séquent représente le cas général et, comme tel, il est le seul à avoir des hypothèses.

6.2.2 Avancements et induction

Dans la seconde phase, on recherche une preuve inductive (au sens de l'induction structurelle sur les suites finies) du ou des séquents formés précédemment. De manière générale, nous utilisons les règles d'avancement[5], la règle d'induction structurelle[6] et des règles de simplification[7].

Premier séquent Le premier séquent de notre exemple se simplifie en

$$(\text{seq}_1)\frac{}{\varepsilon}$$

ce qui est trivialement vrai. Ce séquent ne donne pas lieu à une obligation de preuve pour PVS.

Deuxième séquent Le deuxième séquent se simplifie en

$$(\text{seq}_2)\frac{}{1.\varepsilon > 0}$$

Même sous cette forme, ce séquent n'est pas trivialement vrai, si bien qu'il faut poursuivre la preuve. Les règles d'avancement nous permettent de séparer le séquent en deux :

$$(\text{seq}_{2'})\frac{}{1 > 0} \qquad\qquad (\text{seq}_{2''})\frac{}{\varepsilon > 0}$$

Le premier sous-but sera déchargé comme première obligation de preuve en PVS (cf. l'annexe C), tandis que le second se simplifiera en une expression trivialement vraie. La seconde branche de la preuve s'arrête donc ici.

Règle d'induction Il reste à prouver le cas général, (seq$_3$), qui demeure inchangé par simplification. Pour trouver une preuve inductive de ce séquent, nous utilisons une règle dérivée de la règle d'induction structurelle. Partant de la forme initiale,

$$(\text{ind})\frac{Q(\varepsilon) \qquad \forall z \in D^+.\ Q(tl\ z) \Rightarrow Q(z)}{\forall y \in D^*.\ Q(y)}$$

[5] avancement : cf. section 3.2, page 31
[6] induction structurelle : cf. section 3.2, page 31
[7] simplification : cf. section B.1.2

nous avons décidé de skolémiser le deuxième sous-but et de ne pas représenter l'hypothèse d'induction. En pratique, nous utilisons donc une règle de cette forme :

$$(\text{ind'})\frac{Q(\varepsilon) \qquad Q(z)}{\forall y \in D^*.\ Q(y)}$$

Le séquent (seq_3) donne lieu, par cette nouvelle règle, à deux nouveaux buts :

$$(\text{seq}_{3\varepsilon})\frac{\varepsilon > 0 \qquad 1.(\varepsilon + \varepsilon) > 0}{1.(1.(\varepsilon + \varepsilon) + 0.\varepsilon) > 0} \qquad\qquad (\text{seq}_{3'})\frac{f > 0 \qquad 1.(f + g) > 0}{1.(1.(f + g) + 0.f) > 0}$$

($\text{seq}_{3\varepsilon}$) devient trivialement vrai par simplification. Reste donc à prouver ($\text{seq}_{3'}$), et on peut remarquer au passage que c'est exactement le même séquent que (seq_3).

Qu'est-ce qui nous permet d'aménager la règle (ind) de la sorte ? En effet, il est assez déconcertant d'« oublier » l'hypothèse d'induction, car si le procédé est *correct*, c'est néanmoins un affaiblissement considérable : comment allons-nous pouvoir conclure ?

La clé de l'explication est la remarque suivante : la formule $Q(z)$ contient implicitement sa propre hypothèse d'induction $Q(tl\ z)$, car il suffit d'appliquer l'opérateur tl à toutes les variables pour retrouver cette hypothèse. Ainsi, lorsque nous allons appliquer les règles d'avancement, nous allons obtenir, outre les obligations de preuve scalaires, une série de séquents dérivés de $Q(z)$. Alors le cas inductif sera atteint, lorsque le séquent courant aura la forme $P(tl\ z)$, tandis que le séquent précédent sera $P(z)$.

Troisième séquent Avant de justifier ce processus plus en détail, illustrons-le sur ($\text{seq}_{3'}$). Un premier avancement génère deux formules : une propriété scalaire[8] et un nouveau séquent

$$\frac{hd(f) > 0 \qquad 1 > 0}{1 > 0} \qquad\qquad (\text{seq}_{3''})\frac{\begin{array}{ll} hd(f) > 0 & 1 > 0 \\ f' > 0 & f + g > 0 \end{array}}{1.(f + g) + 0.f > 0} \quad \text{où } f' = tl(f)$$

On peut remarquer que ($\text{seq}_{3''}$) contient à la fois des hypothèses scalaires (comme $hd(f) > 0$) et des hypothèses sur les suites ($f + g > 0$). Les deux jouent un rôle dans la recherche du cas inductif.

Un deuxième avancement crée une nouvelle obligation de preuve (la deuxième dans l'annexe C) et un nouveau séquent :

$$\frac{\begin{array}{ll} hd(f) > 0 & 1 > 0 \\ hd(f') > 0 & hd(f) + hd(g) > 0 \end{array}}{1 + 0 > 0} \qquad (\text{seq}_{3'''})\frac{\begin{array}{ll} hd(f) > 0 & 1 > 0 \\ hd(f') > 0 & hd(f) + hd(g) > 0 \\ f'' > 0 & f' + g' > 0 \end{array}}{f + g + f > 0} \text{ où } \begin{cases} f' = tl(f) \\ f'' = tl(f') \end{cases}$$

Un troisième avancement produit une obligation de preuve et un séquent *qu'on réécrit en effaçant les hypothèses scalaires les plus anciennes*. Du point de vue logique, il s'agit donc d'un affaiblissement, qui est une règle de déduction correcte :

$$\frac{\begin{array}{ll} hd(f) > 0 & 1 > 0 \\ hd(f') > 0 & hd(f) + hd(g) > 0 \\ hd(f'') > 0 & hd(f') + hd(g') > 0 \end{array}}{hd(f) + hd(g) + hd(f) > 0} \qquad (\text{seq}_{3''''})\frac{\begin{array}{ll} hd(f') > 0 & hd(f) + hd(g) > 0 \\ hd(f'') > 0 & hd(f') + hd(g') > 0 \\ f''' > 0 & f'' + g'' > 0 \end{array}}{f' + g' + f' > 0} \text{ où } \begin{cases} f' = tl(f) \\ f'' = tl(f') \\ f''' = tl(f') \end{cases}$$

[8] Cette propriété scalaire pourrait être déchargée en PVS en tant que obligation de preuve. En fait, on peut se rendre compte que l'obligation ($\text{seq}_{2'}$) subsume cette formule, donc on peut se dispenser de la preuve. Cette remarque n'est pas spécifique à notre exemple, si bien que Gloups en tient compte de manière générale et ne génère pas cette obligation.

Et pour finir, nous appliquons encore une fois le couple avancement–affaiblissement pour aboutir à

$$\frac{\begin{array}{ll} hd(f') > 0 & hd(f) + hd(g) > 0 \\ hd(f'') > 0 & hd(f') + hd(g') > 0 \\ hd(f''') > 0 & hd(f'') + hd(g'') > 0 \end{array}}{hd(f') + hd(g') + hd(f') > 0} \qquad (\text{seq}_{3''''''}) \frac{\begin{array}{ll} hd(f'') > 0 & hd(f') + hd(g') > 0 \\ hd(f''') > 0 & hd(f'') + hd(g'') > 0 \\ f'''' > 0 & f''' + g''' > 0 \end{array}}{f'' + g'' + f'' > 0}$$

Ainsi, nous avons notre quatrième obligation de preuve *et nous avons atteint le cas inductif* : en effet, nous pouvons constater que si nous écrivons le séquent ($\text{seq}_{3''''}$) sous la forme d'une formule qui dépend de f et de g, $P(f, g)$, alors le séquent suivant ($\text{seq}_{3''''''}$) est très précisément $P(f', g') = P(tl(f), tl(g))$.

Pour bien montrer que nous pouvons effectivement conclure par induction, nous allons expliciter l'hypothèse d'induction du séquent ($\text{seq}_{3'}$) :

$$(tl(f) > 0 \wedge 1.(tl(f) + tl(g)) > 0) \Rightarrow 1.(1.(tl(f) + tl(g)) + 0.tl(f)) > 0$$

Avec la notation $f' = tl(f)$, nous retrouvons une forme plus claire

$$(f' > 0 \wedge 1.(f' + g') > 0) \Rightarrow 1.(1.(f' + g') + 0.f') > 0$$

Étant donné que les prémisses $f' > 0$ et $1.(f' + g') > 0)$ sont déductibles des prémisses de ($\text{seq}_{3'}$), l'hypothèse d'induction se réduit à[9]

$$1.(1.(f' + g') + 0.f') > 0$$

Quatre déroulements et affaiblissements suffisent pour extraire

$$f'' + g'' + f'' > 0$$

ce qui est exactement la formule dont nous avons besoin pour conclure dans le séquent ($\text{seq}_{3''''''}$).

Preuve infinie À titre de curiosité, nous pouvons mentionner que dans le cas de de la suite de Fibonacci, il existe une autre approche de la preuve : étant données les règles d'avancement et d'affaiblissement, chaque nouvelle obligation de preuve serait de la forme de la quatrième obligation, avec simplement $tl^n(f)$ à la place de f et $tl^n(g)$ à la place de g. Mais *structurellement*, il s'agirait de la même formule.

On pourrait donc utiliser le théorème de la preuve infinie de Coquand [14] pour affirmer que la quatrième obligation de preuve est *le prototype* de toutes les autres obligations de preuve, si bien qu'il suffirat de prouver le prototype, pour les prouver toutes. Dans cette optique, l'arbre de preuve serait un arbre rationnel, avec la dernière branche qui reboucle sur elle-même.

L'application du théorème de Coquand est conditionnée par l'avancement d'au moins un constructeur dans le rebouclage : cette condition est vérifiée dans notre cas, notamment parce que la formule de départ ne contient pas d'opérateur `when`. En effet, cet opérateur « n'avance pas » dans certains cas, si bien que la méthode de Coquand ne peut pas être appliquée. C'est pourquoi nous utilisons l'induction continue qui est plus générale.

Processus de preuve Nous avons résumé la deuxième phase du processus de preuve sur la figure 6.7. Comme la première phase nous a fourni un ensemble de séquents sans `current`, l'algorithme représenté sur la figure 6.7 s'applique à chacun de ces séquents, l'un après l'autre.

Cette seconde phase est donc une boucle qui enchaîne les simplifications avec les avancements et affaiblissements, jusqu'à trouver le cas inductif. L'algorithme travaille sur le séquent courant cour qu'il transforme et qu'il compare au séquent précédent prec. C'est l'étape d'avancement qui calcule un nouveau séquent courant et relègue l'existant au rang du « précédent ».

[9] Nous avons déjà utilisé ce procédé général dans la section dédiée à la règle d'induction structurelle, page 31.

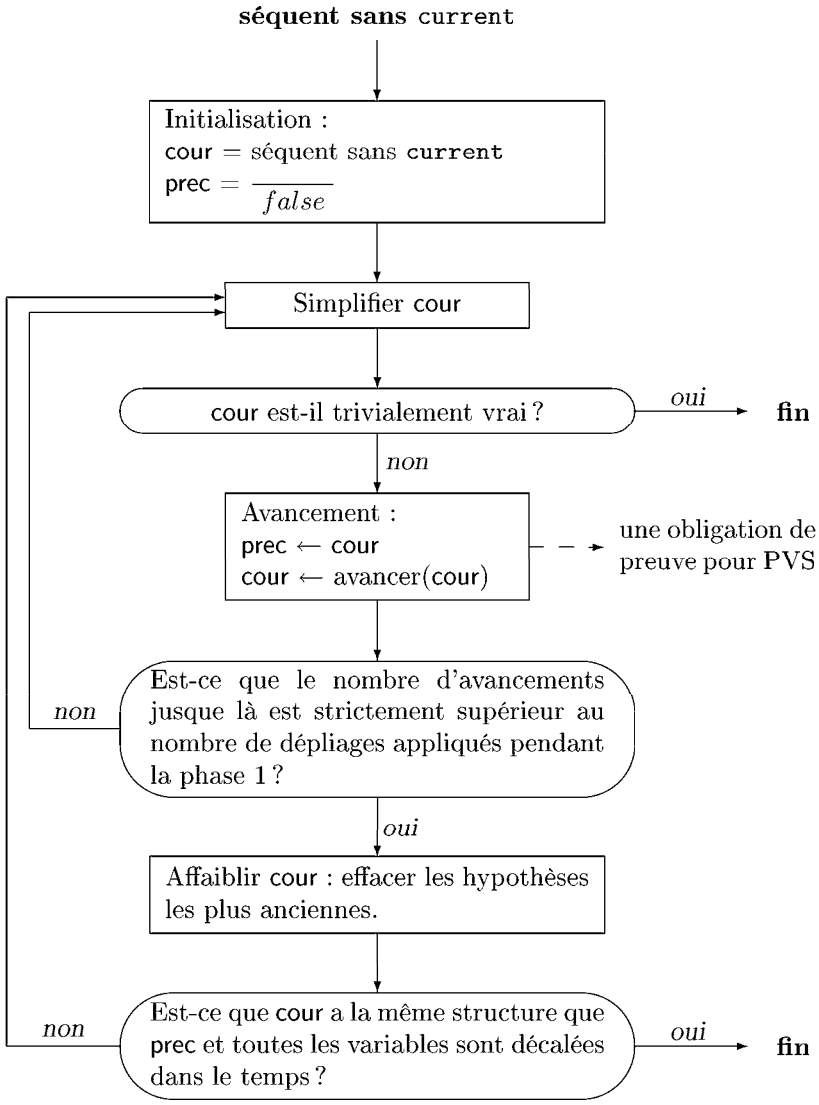


FIG. 6.7 – La phase d'avancements–induction

Début des affaiblissements Un premier point non encore discuté est le critère que nous avons choisi pour appliquer les affaiblissements. En effet, nous laissons avancer le séquent de départ autant de fois qu'il a été déplié pendant la première phase, avant de commencer à effacer les hypothèses les plus anciennes[10].

L'idée directrice de ce choix est de considérer que le rang du dépliage est suffisant pour prouver la propriété de départ : si nous avons déplié n fois, il devrait être suffisant d'avoir des hypothèses sur les n derniers états du système (les n dernières mémoires) pour prouver l'état suivant (le $n+1^{\text{ème}}$).

En pratique, le critère que nous avons choisi assure que le séquent courant commencera par accumuler des hypothèses scalaires jusqu'à en avoir n rangs, puis seulement interviendront les affaiblissements. Les séquents suivants auront donc exactement n rangs d'hypothèses scalaires chacun.

Cette accumulation progressive explique pourquoi nous ne comparons pas les séquents courant et précédent avant le début des affaiblissements : n'ayant pas le même nombre d'hypothèses, leurs structures sont forcément différentes, si bien que la comparaison échouerait de toute manière.

Terminaison Reste la question de l'arrêt de cet algorithme récursif. Nous pouvons résumer l'idée de la preuve ainsi : le programme Lustre sous-jacent est un programme fini, donc la propriété et ses dépliages sont des formules finies. Or les avancements ne font que diminuer la taille des formules, au sens large : faire avancer $1.f$ donne f, faire avancer f donne f'. Après un nombre suffisant d'avancement, le séquent devient donc stationnaire : toute sous-formule ne se réécrit plus qu'en remplaçant $f^{(n)}$ par $f^{(n+1)}$ pour toute variable f. Alors le cas inductif est atteint après au plus n avancements-affaiblissements, n étant le nombre de rangs d'hypothèses scalaires, c'est-à-dire le nombre de dépliages de départ.

Nous avons présenté l'algorithme de preuve de manière aussi abstraite que possible, pour permettre une bonne vue d'ensemble. Dans la suite, nous allons aborder quelques points de détail de son implémentation effective.

6.3 Les fichiers PVS créés

Le lecteur aura pu le constater dans l'annexe C, à chaque obligation de preuve correspond un fichier PVS. De plus, il y a deux autres fichiers pour l'ensemble des obligations : un document de typage des variables et un document maître.

Un obligation de preuve définit une variable booléenne, `TH`k`_value`, qui exprime la valeur de vérité de la $k^{\text{ème}}$ obligation. Ensuite, le lemme correspondant, `TH`k, a pour but de prouver que `TH`k`_value` est vraie. Le document maître contient alors le théorème général, `ALL_OK`, qui demande d'établir toutes les obligations de preuve. L'utilisateur a donc pour tâche de prouver tous les lemmes `TH`k, puis d'en déduire le théorème principal.

Le découpage en plusieurs fichiers s'est avéré nécessaire, car nous avons constaté que Gloups pouvait facilement générer des obligations de preuve de taille excédant la capacité de PVS. Avoir plusieurs fichiers plutôt qu'un seul a donc pour but principal de limiter la taille de chacun de ces fichiers.

Nous allons voir par la suite que les problèmes de taille étaient une de nos préoccupations majeures et que nous avons développé un certain nombre d'optimisations pour y parer.

Nous avons également profité du découpage en plusieurs fichiers pour instaurer un début d'approche générique de la preuve. En effet, on peut constater que les obligations de preuve importent une théorie appelée `Lustre_helpers` : cette théorie est donc sous-jacente à toutes les preuves que l'utilisateur effectue. Dans ce cas précis, il s'agit d'une définition des opérateurs `mod` et `div` et de quelques lemmes de base pour rendre leur manipulation plus aisée.

[10] Dans l'exemple de la suite de Fibonacci, le premier affaiblissement intervient bien après le troisième ($3 > 2$) avancement.

Il serait alors facile pour Gloups de remplacer le nom de la théorie importée par un autre : nous pourrions ainsi générer une obligation de preuve s'appuyant sur une autre définition des opérateurs `mod` et `div`, par exemple pour tenir compte de la réalité sur une plate-forme matérielle donnée[11].

Cependant, on pourrait poursuivre l'idée en proposant d'autres théories, par exemple pour redéfinir les types de base. Par défaut, les entiers en PVS peuvent prendre n'importe quelle valeur : une théorie spécifique pourrait proposer de se limiter aux entiers représentables sur 32 bits. Gloups saurait alors inclure les théories demandées par l'utilisateur : ainsi, l'obligation de preuve générée (toujours la même) pourrait être déclinée selon des critères spécifiques.

6.4 Optimisations

Lors de nos expérimentations pratiques avec Gloups, nous avons vite constaté que la taille des objets à traiter était le facteur limitatif de l'outil. Un temps de calcul faible est certes appréciable, mais pas vital, car notre but n'est pas d'effectuer une vérification à la volée ou en temps réel. Par contre, si les calculs ne peuvent pas se poursuivre, faute d'espace mémoire, la procédure de preuve échoue sans rémission possible : chaque nouvelle exécution de l'outil sur les mêmes données de départ échouerait également.

Or l'occupation de la mémoire peut croître très vite : par exemple chaque dépliage peut facilement faire doubler la taille du séquent. Alors, pour pouvoir traiter des systèmes de tailles intéressantes, nous nous sommes attelés à rechercher des optimisations visant à réduire la complexité spatiale de l'algorithme de preuve.

Nous avons déjà rencontré une telle optimisation dans les paragraphes précédents. En section 6.2.2, nous avons vu un aménagement de la règle d'induction, qui consiste à ne par représenter explicitement l'hypothèse d'induction, ce qui permet de réduire la taille du séquent d'un facteur 2. Nous allons maintenant présenter deux autres optimisations : celle des expressions et celle des séquents.

6.4.1 Représentation optimisée d'expressions

Il est habituel de représenter une formule par une structure abstraite arborescente. Par exemple, l'expression

```
(if (true -> pre b) then x - y else y - x)/ 2.0
```

serait représentée par l'arbre de la figure 6.8.

On conçoit facilement qu'une telle représentation consomme beaucoup d'espace : c'est le prix à payer pour un modèle très général. Or Gloups n'a pas besoin de connaître le détail de chaque expression, car les opérations que l'outil effectue ne concernent que certaines sous-expressions :

- Le dépliage consiste à remplacer une variable par sa définition. Seules les feuilles représentant les variables sont donc transformées par le dépliage.
- La généralisation remplace l'opérateur `current` par un opérateur `default`. Le reste de l'expression demeure inchangé.
- L'avancement concerne principalement les variables (on transforme f en f'). Les opérateurs `pre`, $\rightarrow$ et `fby` peuvent être simplifiés lors de l'avancement (par exemple $tl(a\ fby\ b) = b$). L'opérateur `when` a également un comportement particulier lors de l'avancement, mais le reste des expressions n'est pas concerné.

On peut en déduire qu'il est inutile de représenter toutes les sous-expressions en détail : il suffit de pouvoir distinguer celles qui nécessitent un traitement particulier pendant le déroulement de la preuve, les autres peuvent être « compactées », comme sur la figure 6.9.

[11] En effet, si tout le monde semble en phase pour définir le reste de la division euclidienne entre deux entiers naturels, des divergences d'implémentation arrivent dès qu'un des deux opérandes devient négatif...

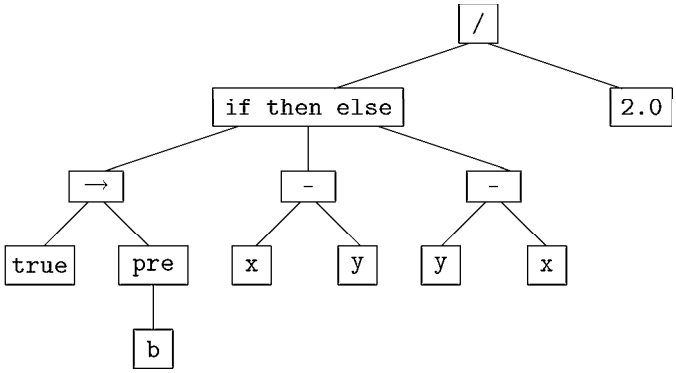


FIG. 6.8 – Exemple d'expression non optimisée

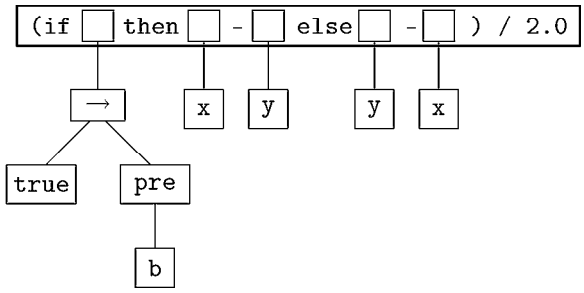


FIG. 6.9 – Expression optimisée

En pratique, nous avons choisi de représenter les sous-expressions invariantes par la chaîne de caractères[12] correspondante, entrecoupée de pointeurs vers les expressions variables. Au fil des avancements, la complexité des formules diminue, à mesure que les opérateurs `pre`, `fby` et → disparaissent. Ainsi, l'expression typique serait une chaîne de caractères pointant vers une série de variables, comme sur la figure 6.10.

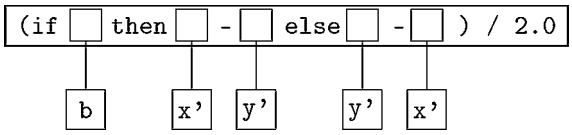


FIG. 6.10 – Expression optimisée, puis avancée

Notre implémentation présente plusieurs avantages : outre un gain d'espace considérable, on constate que le temps de parcours d'une formule optimisée est quasi-linéaire en fonction de sa taille. Ainsi, toutes les opérations (dépliage, avancement, comparaison) ont également profité de la réduction spatiale.

Nous pouvons remarquer que l'optimisation des expressions a été grandement facilitée par l'aspect

[12] Les chaînes de caractères donnent un bon compromis entre la compacité de la représentation et sa lisibilité : on pourrait définir une forme encore plus compacte, mais bien plus difficile à débogger.

« à objets » du langage Java. En effet, c'est grâce à la notion de sous-classe que nous avons pu faire cohabiter les représentations arborescente (pour `pre`, `current` etc.) et linéaire, l'une contenant l'autre et vice-versa.

6.4.2 Optimisations des séquents

Indépendamment de l'optimisation des expressions individuelles, nous avons également mis en place deux optimisations au niveau de séquents entiers. La première concerne la représentation interne et est destinée à réduire l'occupation de la mémoire par Gloups ; la seconde se situe au niveau de la sortie vers PVS et a pour but de diminuer la taille des obligations de preuve déchargées.

Le dépliage optimisé La première idée pour l'optimisation des séquents part d'un constat simple : si nous effectuons le dépliage en réécrivant effectivement les formules, nous risquons de faire grandir leur taille de manière exponentielle. En effet, par exemple dans le système suivant

```
f = 1 -> (pre f + pre g) ;
g = 2 -> (pre f + pre g) ;
```

ou de manière équivalente $\begin{cases} f = 1.(f+g) \\ g = 2.(f+g) \end{cases}$

la propriété $f > 0$ se déplie successivement en

$$\frac{f>0}{1.(f+g)>0}\ ,\ \frac{f>0 \quad 1.(f+g)>0}{1.(1.(f+g)+2.(f+g))>0}\ ,\ \frac{f>0 \quad 1.(f+g)>0 \quad 1.(1.(f+g)+2.(f+g))>0}{1.(1.(1.(f+g)+2.(f+g))+2.(1.(f+g)+2.(f+g)))>0}\ \cdots$$

C'est pourquoi nous avons opté pour un dépliage avec mise en facteur des sous-expressions communes : les équations sont données explicitement et le dépliage est seulement amorcé. La forme précédente peut être retrouvée grâce au principe de substitutivité. Sur notre exemple, on obtient :

$$\frac{f_0>0 \quad f_1=1.(f_0+g_0) \quad g_1=2.(f_0+g_0)}{f_1>0}\ ,\ \frac{\begin{array}{c} f_0>0 \quad f_1=1.(f_0+g_0) \quad g_1=2.(f_0+g_0) \\ f_1>0 \quad f_2=1.(f_1+g_1) \quad g_2=2.(f_1+g_1) \end{array}}{f_2>0}\ ,$$

$$\frac{\begin{array}{c} f_0>0 \quad f_1=1.(f_0+g_0) \quad g_1=2.(f_0+g_0) \\ f_1>0 \quad f_2=1.(f_1+g_1) \quad g_2=2.(f_1+g_1) \\ f_2>0 \quad f_3=1.(f_2+g_2) \quad g_3=2.(f_2+g_2) \end{array}}{f_3>0}\ \cdots$$

La taille des séquents augmente donc linéairement. De plus, cette implémentation du dépliage est plus rapide, car elle demande seulement un « décalage des indices », tandis que le dépliage jusqu'au-boutiste nécessite des copies physiques des expressions.

En Gloups, toutes les variables portent deux indices, par exemple $f_{0,1}$, ce qui se retrouve en PVS sous la forme `f_0_1`. Le premier indice correspond à l'étape du dépliage, tandis que le second apparaît lors des avancements : le premier élément d'un flot a l'indice 0, le second 1 etc. Ainsi, `f_0_1` correspond à $hd(tl(f_0))$.

Le cône d'influence Pour réduire la taille des obligations de preuve générées par Gloups, nous avons implémenté un algorithme d'affaiblissement des séquents déchargés, c'est-à-dire d'effacement de certaines hypothèses, jugées inutiles pour la preuve de l'obligation. Cet algorithme est fondé sur un calcul du cône d'influence[13] des variables du but : on effacera les expressions en dehors du cône, car elles ne pourront

[13] Connu aussi comme algorithme de tranchage (« slicing » en anglais).

pas intervenir dans la preuve. Cet algorithme peut être appliqué à la sortie de la phase de dépliages–généralisations, ou à la fin de la phase d'avancements–induction (ou les deux).

La figure 6.11 présente le fonctionnement de notre algorithme. De manière générale, il s'agit de calculer l'ensemble V, censé contenir toutes les variables ayant un rapport avec le but à prouver. Au départ, on initialise V avec les variables du conséquent, puis on élargit cet ensemble jusqu'à ce qu'un point fixe soit atteint. À ce moment-là, on peut effacer tous les antécédents dont les variables ne figurent pas dans V.

$V \leftarrow$ l'ensemble des variables (indicées) du conséquent
$V' \leftarrow \emptyset$
tant que $(V \neq V')$ **faire**
    $V' \leftarrow V$
    pour tout antécédent A **faire**
        $V'' \leftarrow$ l'ensemble des variables de A
        $c \leftarrow \mathit{cône}(A, V, V'')$
        si c **alors** $V \leftarrow (V \cup V'')$
    fin pour tout
fin tant que
ne garder que les antécédents dont les variables figurent dans V

FIG. 6.11 – Algorithme de tranchage

On peut remarquer que cet algorithme s'arrête forcément, car l'ensemble de toutes les variables d'un séquent est fini : V est majoré par cet ensemble englobant et, de plus, V est croissant. La suite des valeurs successives de V a donc une limite finie, qu'elle atteint au bout d'un nombre fini d'étapes.

Notre algorithme de tranchage fait appel à la fonction *cône*, dont nous n'avons pas encore parlé. Cette fonction a pour rôle de prendre la décision d'élargir ou non l'ensemble V. En fait, nous avons implémenté deux versions différentes de cette fonction, chacune effectuant un tranchage plus ou moins fort. Les deux versions sont représentées sur la figure 6.12. Dans la version de base, un antécédent est pris dans le cône d'influence, dès qu'une de ses variables figure dans l'ensemble V ; le cas échéant, toutes les variables de l'antécédent sont rajoutées à V. La version avancée est plus stricte quant à la condition d'élargissement du cône : pour qu'un antécédent fasse partie du cône, il faut que qu'il soit de type « équation » et que le côté gauche de cette équation figure dans V.

Tranchage normal

fonction $\mathit{cône}(A, V, V')$
retourne booléen b
    $b \leftarrow (V \cap V'') \neq \emptyset$

Tranchage avancé

fonction $\mathit{cône}(A, V, V')$
retourne booléen b
    si A est de la forme
        « variable v = expression e »
    alors $U \leftarrow \{v\}$
    sinon $U \leftarrow \emptyset$
    fin si
    $b \leftarrow (U \cap V) \neq \emptyset$

FIG. 6.12 – Différentes conditions d'élargissement

Sur l'exemple précédent, le tranchage normal appliqué au troisième séquent

$$\frac{\begin{array}{llll} f_0 > 0 & f_1 = 1.(f_0 + g_0) & g_1 = 2.(f_0 + g_0) \\ f_1 > 0 & f_2 = 1.(f_1 + g_1) & g_2 = 2.(f_1 + g_1) \\ f_2 > 0 & f_3 = 1.(f_2 + g_2) & g_3 = 2.(f_2 + g_2) \end{array}}{f_3 > 0}$$

laisse ce dernier inchangé. Par contre, le tranchage avancé permet d'effacer l'hypothèse $g_3 = 2.(f_2 + g_2)$.

On obtient des réductions bien plus importantes à la fin de la phase des avancements–induction : par exemple, nous avons utilisé le tranchage avancé pour produire les obligations de preuve de l'annexe C, ce qui nous a permis de réduire le nombre d'antécédents à au plus cinq, alors que sans le tranchage, il y en aurait jusqu'à dix-huit.

6.4.3 Apport pratique des optimisations

Les optimisations que nous venons de présenter se sont avérées suffisantes pour ce qui concerne Gloups. Notre première implémentation pouvait traiter des systèmes comportant une dizaine d'équations : après optimisation des expressions et des séquents, nous avons su calculer les obligations de preuve pour un exemple avec 170 équations.

Nous pourrions pousser la réduction de l'occupation de mémoire dans Gloups encore plus loin, mais il se trouve que PVS n'est pas capable de gérer les obligations de preuve pour des systèmes aussi grands, même après le tranchage avancé. En effet, l'une des obligations issues de notre exemple à 170 équations est un fichier de 1,2 mégaoctets, ce qui dépasse les capacités de PVS.

6.5 Exemple du synchronisateur d'horloges

Pour terminer la présentation de Gloups, nous allons décrire un exemple réel d'application de notre logiciel. En 1996, Miner et Johnson [36] ont proposé une preuve co-inductive de l'équivalence de deux circuits de synchronisation d'horloges tolérants aux pannes. Grâce à Gloups, nous allons pouvoir effectuer une preuve inductive de la même propriété.

6.5.1 Présentation des circuits

Circuit standard La figure 6.13 présente le circuit standard. Ce circuit reçoit trois signaux booléens, ($\mathbf{f}_1$, **nf** et **R**eset qui ne figure pas sur le schéma) et un signal entier **rd**. Le signal entier **nor** constitue la sortie. Lorsque **R**eset est vrai, tous les autres signaux sont forcés à faux (ou à 0). De plus, **nf** ne peut devenir vrai que si $\mathbf{f}_1$ l'est déjà.

Le circuit fonctionne comme ceci : **rd** est incrémenté de 1 à chaque instant. Lorsque x horloges ont envoyé leurs signaux de synchronisation, $\mathbf{f}_1$ devient vrai. Lorsque $x+a$ horloges ont envoyé leur signal, **nf** devient vrai à son tour, et **nor** contient alors le nouveaux temps de référence, qu'on diffuse aux horloges pour la synchronisation. À la fin du cycle, **R**eset devient vrai et tout recommence. Ce dispositif peut être utilisé pour synchroniser n horloges matérielles (avec $n \geq 2x + a - 1$) et tolère $x - 1$ pannes.

Circuit optimisé Le deuxième circuit est sur la figure 6.14. Il possède les mêmes entrées et sorties que le précédent, mais nous avons renommé la sortie en **opt**. Ce circuit est censé fournir la même fonctionnalité que le circuit standard, en utilisant seulement un additionneur +1 à la place de l'additionneur et du diviseur /2. Ainsi, ce nouveaux circuit est optimisé en termes du nombre de composants.

En fait, ce second circuit calcule la sortie **opt** progressivement, entre le front montant de $\mathbf{f}_1$ et celui de **nf**, tandis que le circuit standard effectue tous les calculs à la fin, sur le front montant de **nf**.

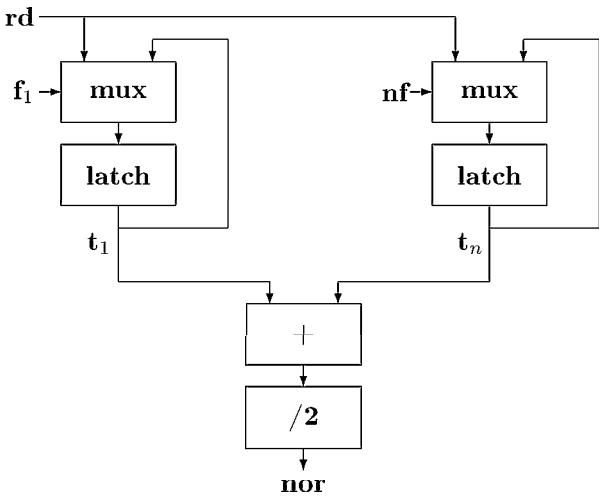


FIG. 6.13 – Le circuit standard

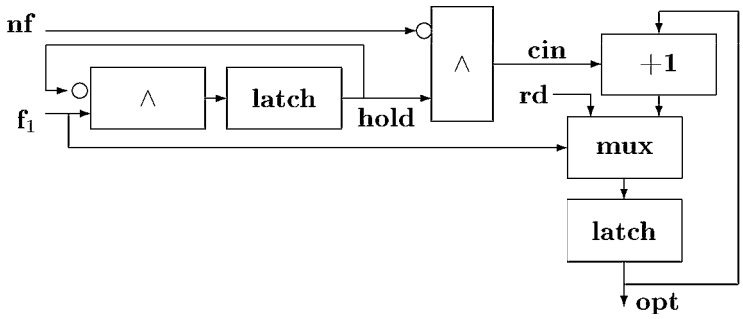


FIG. 6.14 – Circuit optimisé

6.5.2 La preuve de l'équivalence

Miner et Johnson commencent par modéliser les deux circuits en PVS, à l'aide de suites infinies de booléens et d'entiers, puis ils proposent une relation entre les deux circuits et prouvent que cette relation est une bisimulation. Ils en déduisent alors que les deux sorties **nor** et **opt** sont égales, si les entrées le sont.

Notre modélisation du problème utilise naturellement Lustre. Le nœud `clock` suivant contient
- la définition du flot **rd**, commune aux deux circuits
- la définition des flots du circuit standard : **nor**, $\mathbf{t}_1$ et $\mathbf{t}_n$
- les flots du circuit optimisé : **h**, **cin**, **opt**
- les hypothèses sur l'environnement dans les clauses `assert` :
  1. on suppose que les circuits sont **R**éinitialisés au premier instant
  2. **nf** devient vrai après $\mathbf{f}_1$, conformément à ce que nous avons expliqué dans la description des circuits

3. $\mathbf{f}_1$ est à faux au premier instant
4. on suppose qu'une fois le signal **nf** devient vrai, il reste à vrai, jusqu'à la prochaine **R**éinitialisation

– la propriété à prouver : **nor=opt**

```
node clock(R,f1,nf : bool) returns (nor, opt : int);
var h, cin : bool;  t1, tn, rd : int ;
let
  rd = 0 -> if R then 0 else pre rd + 1 ;

  nor = (t1 + tn) div 2 ;
  t1 = 0 -> if R then 0 else (if f1 then pre t1 else rd) ;
  tn = 0 -> if R then 0 else (if nf then pre tn else rd) ;

  h = false -> (not R) and f1 and (not pre h) ;
  cin = false -> (pre h) and not nf ;
  opt = 0 -> if R then 0 else
        (if f1 then (if cin then 1 else 0) + pre opt else rd);

  assert R -> true ;
  assert nf => f1 ;
  assert not f1 -> true ;
  assert true -> R or (pre nf => nf) ;

  prove nor = opt ;
tel ;
```

Nous fournissons le nœud `clock` à Gloups, qui calcule un ensemble de quatre obligations de preuve. Ces obligations donnent lieu à des preuves faciles en PVS : il suffit d'effectuer des disjonctions de cas sur les variables booléennes du problème (**R**, $\mathbf{f}_1$ et **nf**) et laisser PVS conclure.

Ainsi, notre outil de preuve nous a permis d'alléger considérablement le travail de preuve : d'une part, la modélisation des circuits électroniques est directe en Lustre et, d'autre part, les obligations de preuve scalaires, même si volumineuses, ne constituent pas un défi. Comparée à la modélisation ad hoc de Miner et Johnson, suivie de la recherche d'une bisimulation (qui est plus compliquée que le simple **nor=opt**), notre approche est donc bien plus intéressante sur le plan pratique.

Nous pouvons remarquer que la tactique de co-induction (cofix) implantée par Giménez dans Coq [19] permet également de trouver, dans certains cas, la bonne relation de bisimulation par des dépliages successifs. La preuve se ramène donc automatiquement à des obligations non (co-)inductives ; mais Gloups propose en plus de cela l'extraction des obligations de preuve scalaires et la détection de la terminaison.

6.6 Bilan

L'implémentation d'une version corrigée de Gloups nous a permis de mettre en pratique les notions théoriques que nous avons abordées au cours des chapitres précédents. Grâce à notre outil, nous avons pu mener à bien quelques études de cas et de démontrer ainsi l'intérêt de la méthode théorique.

Avec ses 18000 lignes de code et 16000 lignes de commentaires, Gloups n'est plus un logiciel-jouet : il ne se réduit pas à l'implémentation directe de la méthode de preuve théorique, mais propose également des optimisations de la méthode et de son fonctionnement. Ainsi, on peut ajouter Gloups à l'écurie des outils de vérification développés à Vérimag (nbac, lesar).

Néanmoins, notre logiciel reste encore au stade expérimental : certains choix n'ont pas encore été définitivement arrêtés et certaines constructions de Lustre mériteraient un meilleur traitement. En fait, arrivé à un certain stade, nous avons décidé d'investir nos efforts dans une nouvelle direction, qui se situe plutôt du côté de l'utilisation des outils de vérification que de leur développement. Nos recherches dans ce domaine constituent le contenu de la partie suivante.

Troisième partie

Le raffinement

7 – Introduction et motivations

Dans ce chapitre, nous exposons les motivations des recherches présentées dans la troisième partie de ce mémoire. Après avoir souligné l'intérêt du raffinement en général, nous présentons un cadre bien établi de développement par raffinements, la méthode B. Cette présentation nous conduit à une première manière d'introduire le raffinement en Lustre, manière qui s'avère peu pratique. Nous terminons en annonçant une autre technique de raffinement pour Lustre, qui sera développée dans les chapitres suivants.

7.1 Le raffinement en général

Au chapitre 1, nous avons déjà donné un aperçu de ce qu'était le raffinement en informatique. Néanmoins, pour bien expliquer les motivations qui nous ont poussé à nous intéresser à ce domaine, nous proposons au lecteur une description plus détaillée.

Une méthode de développement correct La raison d'être du raffinement est de proposer une méthode de développement de programmes. Partant des spécifications initiales, que l'on raffine en des spécifications de plus en plus détaillées, on arrive finalement au code compilable : le fait que chaque nouvelle étape raffine la précédente assure que celle-là est correcte par rapport à celle-ci, si bien que le programme final constitue, par construction, une implémentation correcte des spécifications de départ.

Dans cette première perspective, le raffinement est donc une manière maîtrisée de dériver un logiciel à partir de ses spécifications.

Une méthode prouvée Pour établir qu'un module raffine un autre, il faut faire des preuves mathématiques. *Prouver* le raffinement a des avantages pratiques indéniables : outre la certitude de ne pas avoir introduit d'erreur dans le programme, on a la possibilité de reproduire la preuve du raffinement. Cette propriété est importante, car les logiciels de certains domaines d'application doivent être certifiés, avant de pouvoir être utilisés. La certification est alors facilitée par les preuves de correction établies pendant le développement : tandis que les tests ne peuvent pas être exhaustifs, voire ne sont pas reproductibles, les preuves, elles, peuvent être vérifiées à nouveau.

En contrepartie, les preuves de raffinement constituent un surcroît de travail lors du développement. De plus, savoir prouver la correction d'un raffinement fait appel à des compétences plus poussées que la programmation. Ainsi, le choix du raffinement comme méthode de développement est en pratique réservée aux projets où le degré de certification exigé ne permet pas de procéder « à moindre frais »[1].

Les exemples d'utilisation de Lustre que nous avons cités au chapitre 1 (Framatome [8], métro de Hong-Kong [28], Airbus [12]) sont autant de développements où la vérification des logiciels est requise et

[1] La norme de sécurité informatique ITSEC définit sept niveaux de certification. À partir du niveau quatre, le logiciel doit être conçu de manière méthodique, et à partir du niveau six, la conception doit être formelle. Par exemple, les cartes à puce sécurisées sont de niveau sept.

où le raffinement aurait pu être utilisé, du moins en théorie, car il est vrai que spécifier formellement un pilote d'avion est une tâche ardue.

Une méthode formelle Pour établir des preuves de raffinement, il faut bien sûr un cadre formel, qui doit être suffisamment général pour pouvoir exprimer les spécifications de haut niveau, les programmes implémentables, ainsi que toutes les étapes intermédiaires. En effet, c'est grâce à ce formalisme unique que le logiciel final peut être prouvé correct par rapport à ses spécifications.

Un cadre formel aussi général a donc l'avantage de pouvoir englober tout le cycle de développement, mais constitue également un inconvénient, car il peut exprimer des constructions plus abstraites qu'un langage de programmation classique, si bien qu'il faut une formation spécifique pour pouvoir développer par raffinements.

Le côté abstrait du formalisme va donc de pair avec la nécessité de savoir faire des preuves mathématiques : les deux exigences contribuent à augmenter le coût d'un développement par raffinements. Cependant, nous allons voir dans la section 7.3.2 que nous pouvons diminuer, voire supprimer ces inconvénients dans le cas de Lustre.

Pour illustrer notre présentation du raffinement, nous allons maintenant introduire l'une des méthodes les plus connues (et les plus utilisées), la méthode B.

7.2 La méthode B

La méthode B [3] s'apparente aux méthodes Z [11] et VDM [26] en ce qui concerne sa conception du raffinement, mais propose en plus de nombreuses constructions issues du génie logiciel. Grâce à l'Atelier B, l'outil qui fournit un environnement de développement, la méthode B constitue un système complet pour le développement de logiciels corrects.

7.2.1 Machines abstraites

L'unité de base de tout système en B est la machine abstraite. C'est un ensemble de déclarations qui définissent un état et des comportements possibles et qui mettent en place une structure de génie logiciel. La forme générale d'une machine abstraite se trouve sur la figure 7.1.

Variables L'état interne d'une machine abstraite est défini par un ensemble de variables typées. Ces variables peuvent être initialisées (de manière déterministe ou non-déterministe). En général, les variables sont encapsulées dans la machine et invisibles de l'extérieur.

Invariants On peut déclarer des propriétés d'invariance sur les variables. L'invariant d'une machine abstraite fournit en fait une abstraction de son état réel.

Lorsqu'un invariant est spécifié, on doit vérifier que l'initialisation (ou la non-initialisation) des variables établit bien cet invariant. C'est une première obligation de preuve de la cohérence de la machine.

Opérations Les variables d'une machine abstraite peuvent devenir visibles[2] de l'extérieur, mais leur valeur ne peut être affectée que de l'intérieur. Outre l'initialisation déjà citée, les variables peuvent être changées par les opérations de la machine.

Une opération est une procédure (ou une fonction) au sens large du terme : c'est une spécification qui relie l'ensemble des entrées admissibles, définies par une pré-condition, à l'ensemble des sorties possibles.

[2]Par exemple, lors de l'inclusion d'une machine dans une autre, les variables de la machine incluse deviennent lisibles dans les opérations de la machine englobante.

De manière générale, une opération est donc une relation partielle et non-déterministe. La méthode B prévoit des obligations de preuve pour vérifier que chaque opération, lorsqu'elle est appelée dans sa précondition, préserve bien l'invariant de la machine.

Ainsi, les opérations constituent les points d'entrée de la machine abstraite : ce sont elles qui, visibles de l'extérieur, permettent d'interroger ou d'affecter l'état de la machine. L'interaction entre deux machines s'effectue généralement par le biais des opérations : l'appel fonctionnel (bloquant) est *le* moyen de communication.

Définitions pratiques Chaque machine abstraite peut définir de nouveaux ensembles (de nouveaux types, en fait), des constantes et des notations raccourcies (des macros). Ces constructions ne sont pas forcément indispensables, mais rajoutent un degré de confort au développement en B.

Structuration du logiciel Les éléments précédents décrivent l'état et le comportement d'une machine abstraite. Cependant, les systèmes réels sont trop grands pour être décrits par une seule machine : c'est pourquoi la méthode B prévoit des mécanismes de structuration, qui facilitent le découpage d'un logiciel en plusieurs machines.

Parmi ces procédés, on trouve l'inclusion (la réutilisation textuelle du code d'une autre machine) et l'utilisation des informations statiques d'une autre machine (constantes, ensembles). De plus, une machine abstraite peut avoir un certain nombre de paramètres, si bien qu'elle devient générique par rapport à ces entités.

`MACHINE` *nom* (*paramètres*)	les paramètres sont optionnels
`CONSTRAINTS` *cr*	contraintes sur les paramètres formels
`INCLUDES` *incl*	liste des mchines incluses
`USES` *u*	liste des machines dont celle-ci partage l'information statique
`SETS` *s*	définitions d'ensembles
`CONSTANTS` *cst*	définitions de constantes
`DEFINITIONS` *def*	macros
`VARIABLES` *v*	variables typées
`INITIALIZATION` *init*	initialisation des variables
`INVARIANT` *inv*	invariant sur les variables
`ASSERTIONS` *a*	propriétés déductibles de l'invariant
`OPERATIONS`	Opérations
$op_1 = \ldots$	
$op_2 = \ldots$	
`PROMOTES` *p*	opérations incluses réexportées

FIG. 7.1 – Une machine abstraite en général

Les machines abstraites mettent en place une manière de décrire les systèmes informatiques, qui est à la fois expressive au niveau logique (on peut décrire des comportements non-déterministes, on peut spécifier des invariants,...) et rigoureuse au niveau du génie logiciel (la visibilité des différents éléments est imposée, on dispose de plusieurs mécanismes de réutilisation du code etc.). Cependant, tout ceci ne constitue que la cadre formel pour le vrai but de la méthode B, le développement par raffinements.

7.2.2 Raffinements

Une machine abstraite est, comme son nom l'indique, abstraite. Elle donne une spécification de haut niveau, en décrivant le comportement désiré, mais pas la manière de l'obtenir[3]. Le rôle du raffinement est alors de réécrire ces spécifications en rajoutant au fur et à mesure des détails supplémentaires, jusqu'à arriver à une forme implémentable.

```
MACHINE Alea (max)
CONSTRAINTS max ∈ 1..MAXINT
SETS espace_choix==(0..max-1)
OPERATIONS
  v←alea =
    ANY x WHERE x ∈ espace_choix
    THEN v :=x
    END
```

FIG. 7.2 – Spécification d'un générateur pseudo-aléatoire

Raffinement de machine abstraite De manière générale, le raffinement d'une machine abstraite est une autre machine abstraite, comme celle de la figure 7.3. En particulier, toutes les vérifications de cohérence que nous avons vues précédemment concernent la machine raffinée.

```
REFINEMENT Alea1 (max)
REFINES Alea
VARIABLES prochain
INVARIANT prochain ∈ espace_choix
INITIALISATION prochain :∈ espace_choix
OPERATIONS
  v←alea =
    ANY x WHERE x ∈ espace_choix
    THEN prochain :=x || v :=prochain
    END
```

FIG. 7.3 – Raffinement du générateur pseudo-aléatoire

Cependant, cette nouvelle machine est dépendante de la précédente, car il faut qu'un utilisateur extérieur ne puisse pas distinguer une machine et son raffinement. C'est l'essence même de la méthode : si tous les comportements du raffinement sont conformes aux spécifications abstraites (c'est-à-dire si la machine abstraite pourrait produire les mêmes traces), alors la machine raffinée en est un cas particulier correct.

Pour assurer que les comportements des deux machines ne peuvent pas être distingués, il faut vérifier un certain nombre de contraintes. D'abord, question de typage, les opérations doivent garder le même profil tout au long du raffinement, parce qu'elles sont visibles de l'extérieur. Par contre, les variables peuvent être remplacées par d'autres, à condition d'expliciter le lien entre les variables abstraites et concrètes (l'« invariant de liaison »).

[3]Par exemple, la figure 7.2 présente une spécification d'un générateur de nombres pseudo-aléatoires. Cette spécification reste vague quant au processus du choix : l'opération *alea* se contente de choisir un nombre de manière non déterministe.

Pour finir, il existe des obligations de preuve du raffinement, qui assurent que l'initialisation de la machine raffinée établit aussi l'invariant abstrait et que les opérations raffinées le préservent, en plus de leur propre invariant. Le cas échéant, les préconditions (les ensembles d'entrées admissibles) des opérations raffinées doivent être plus larges que celles des opérations abstraites : la machine raffinée doit accepter au moins autant d'entrées que sa spécification.[4]

Implémentation Les différents raffinements peuvent s'enchaîner, car la relation induite par la théorie est transitive. Ainsi, une machine M_1 peut être raffinée en M_2, qui donne à son tour un raffinement M_3, qui est raffiné en M_4, et ainsi de suite jusqu'à M_k. Par transitivité, M_k sera alors un raffinement de M_1.

Au cours de ces différents raffinements, on introduit des détails concernant la représentation des données et le code des opérations. Le but est d'obtenir à la fin une machine particulière, appelée « implémentation », qui ne contient que des déclarations directement implémentables. Cette dernière machine doit être décrite dans un langage restreint : les exécutions parallèles, les affectations non-déterministes et les ensembles infinis ne peuvent pas y figurer ; c'est en fait un langage de programmation séquentielle, avec uniquement des types finis[5].

On peut noter que le fait de produire une implémentation constitue la preuve de la faisabilité des spécifications de départ, c'est-à-dire du fait qu'au moins un programme déterministe corresponde à ces spécifications. En effet, la méthode B ne prévoit pas d'obligation de preuve pour vérifier la faisabilité plus tôt dans le cycle de développement, car une telle preuve d'existence pourrait être difficile et reviendrait à exhiber une implémentation : on ferait le même travail deux fois.

```
IMPLEMENTATION Alea2 (max)
REFINES Alea1
CONCRETE_VARIABLES prochain
INITIALISATION prochain = 0
OPERATIONS
  v←alea =
    v := prochain ;
    IF prochain < max-1
    THEN prochain := prochain+1
    ELSE prochain := 0
    END
```

FIG. 7.4 – Une implémentation possible du générateur pseudo-aléatoire

Compositionnalité du raffinement Jusqu'alors, nous n'avons parlé que du raffinement vertical où une machine abstraite donnée est raffinée en une autre, qui est raffinée à son tour etc. Cependant, un système peut être formé de plusieurs machines abstraites, qui communiquent entre elles. Une propriété importante du raffinement en B est alors son aspect compositionnel : si une machine M_1 est raffinée par M_2 et si N_1 est raffinée par N_2, alors l'ensemble $M_1||M_2$ est raffiné par $M_2||N_1$, $M_1||N_2$ et par $M_2||N_2$.

C'est grâce à la compositionnalité du raffinement que la méthode B peut être utilisée dans les projets de grande taille : si une spécification est séparée en plusieurs modules, ceux-ci peuvent être développés

[4] Dans notre exemple, nous avons introduit la variable *prochain* qui contient le prochain nombre pseudo-aléatoire. Vu qu'il n'y avait pas d'invariant dans la machine *Alea*, les vérifications afférentes au raffinement se bornent à s'assurer que la fonction *alea* a toujours le même profil.

[5] La figure 7.4 donne une implémentation du générateur pseudo-aléatoire. Cette implémentation n'est pas très intéressante en pratique, car elle produit les nombres dans l'ordre, mais cela reste une implémentation correcte des spécifications de départ.

par raffinements indépendamment les uns des autres, et cependant les implémentations finales vérifieront les spécifications de départ.

Ainsi, la méthode B offre un système complet de développement de logiciels : c'est une méthode formelle qui permet de garantir la correction des systèmes tout au long du développement, et c'est aussi une méthode de génie logiciel qui met en pratique des concepts de bonne programmation (modularité, réutilisation, séparation entre les spécifications et les implémentations).

7.3 La méthode B et Lustre

Après ce rapide tour d'horizon, on peut se demander quelle est le rapport entre la méthode B et notre langage Lustre. En fait, nous avons vu que Lustre/Scade était utilisé dans le développement de système critiques, soumis à la certification. Or procéder par raffinements est une bonne manière de créer des systèmes vérifiés : il pourrait donc être intéressant de disposer d'une telle technique pour Lustre.

7.3.1 Lustre en B

Nous l'avons vu, la méthode B est une méthode de développement par raffinements très complète. Ainsi, la première idée pour introduire le raffinement en Lustre est de traduire Lustre en B : de cette manière, on pourrait récupérer toute la puissance théorique et l'instrumentation pratique de B. En effet, pourquoi (ré)inventerait-on un raffinement pour Lustre, si on en dispose déjà en B ?

Une traduction de Lustre vers B a été proposée par Cécile Dumas[6] dans sa thèse [17]. Nous nous sommes également essayé à cet exercice lors de nos recherches préliminaires, pour aboutir à une traduction légèrement plus modulaire, mais tout aussi inadaptée sur le plan pratique. Il est en effet possible de traduire Lustre en B, car B est au moins aussi expressive, mais le prix à payer reste trop élevé.

Granularité des nœuds Puisque la machine abstraite est l'unité de base en B et que le nœud est la base de Lustre, le seul choix cohérent de traduction est de faire correspondre une machine à un nœud.

En effet, avec ce choix, on pourra regrouper en cas de besoin plusieurs nœuds au sein d'une même machine abstraite, en bénéficiant du mécanisme d'inclusion. Il serait donc inutile de traduire directement plusieurs nœuds en une machine.

Le choix dual, qui consisterait à traduire un nœud par plusieurs machines abstraites, doit être écarté, car un nœud est une unité synchrone : le synchronisme ne pourrait pas être maintenu, si le nœud était représenté par plusieurs machines[7].

Typage des opérations Un nœud correspond donc à une machine abstraite. Avant de réfléchir au détail de la traduction, il faut choisir entre deux représentations radicalement différentes des nœuds :

- Étant donné qu'un nœud est une fonction de suites (cf. section 2.2.2), le premier choix est celui de faire correspondre à un nœud une machine abstraite, dont les opérations portent sur les suites. En effet, la méthode B est suffisamment expressive pour qu'on puisse définir le type « suite » et même « suite infinie » ; il est également possible d'exprimer la sémantique des opérateurs Lustre, telle qu'elle figure dans l'annexe A.
  Le seul problème, mais de taille, est celui de l'implémentation : comme les opérations doivent garder leur profil tout au long du développement, l'implémentation devrait travailler, elle aussi,

[6] Cécile Dumas a proposé cette traduction pour vérifier des programmes Lustre, non pour les raffiner. Le but (commercial) de sa traduction était de participer à la création d'un outil commun B-Scade.

[7] Ce que nous voulons dire, c'est qu'il faut une seule machine de « haut niveau » pour représenter un nœud. En effet, cette machine pourrait faire appel à d'autres machines qu'elle contrôlerait.

avec des suites. Or, si Lustre spécifie les nœuds en termes de suites, l'implémentation effective des programmes Lustre ne travaille qu'avec des valeurs scalaires : ainsi, garder les suites jusqu'au bout en B irait à l'encontre de ce que nous voulons faire.
- La seconde possibilité est celle de travailler avec des scalaires au lieu de suites. Avec ce choix, traduire Lustre en B revient tout simplement à le compiler : il faut expliciter les mémoires cachées dans les `pre` sous forme de nouvelles variables, il faut déterminer un ordre pour calcul des équations du nœud etc. Avec ce choix, on perd donc toute l'élégance de Lustre et son aspect flot-de-données, mais la traduction demeure possible.

Granularité des opérations La question suivante concerne le nombre d'opérations dans la machine abstraite correspondante à un nœud donné. L'idéal serait d'en avoir une seule, qui produirait les sorties à partir de l'état mémorisé et des entrées courantes. Mais nous avons vu que les flots en Lustre peuvent être sur différentes horloges, si bien que le nombre de valeurs en entrée et en sortie d'un nœud peut varier d'un instant à l'autre : à certains moments, certains flots « n'existent pas ».

Il semble donc nécessaire de définir au contraire une opération par flot d'entrée (« lire la valeur courante, si elle existe ») et une par flot de sortie (« donner la sortie calculée »). Une telle disposition nécessiterait un mécanisme supplémentaire qui empêcherait par exemple de lire une sortie avant que toutes les entrées soient positionnées, car ne pas positionner une entrée reviendrait à changer l'arité du nœud à la volée et à perdre le déterminisme des calculs.

Synchronisme Un autre problème de la traduction est le traitement du caractère synchrone de Lustre. Il est en effet possible d'assurer que les flots avancent à la vitesse qui leur est donnée par leur horloge, mais cela nécessite la mise en place de garde-fous supplémentaires, qui alourdissent la traduction.

Pour conclure, nous pouvons affirmer qu'il est effectivement possible de traduire Lustre en B, mais que la traduction présente un certain nombre d'inconvénients : d'une part, elle dénature les programmes Lustre en les compilant et d'autre part, elle met en place des mécanismes supplémentaires qui entravent sa lisibilité. De plus, il n'est pas sûr que raffiner une traduction de Lustre en B aboutit à un système synchrone.

Tous ces arguments plaident contre la traduction de Lustre en B, suivie du raffinement. En effet, les utilisateurs de Lustre/Scade sont généralement automaticiens ou électroniciens, mais peu possèdent une véritable formation en informatique. L'une des raisons pour lesquelles ils utilisent Lustre/Scade est précisément la simplicité du langage et sa parenté avec les circuits électroniques. Il est difficilement imaginable que ces mêmes utilisateurs consentent à abandonner leur outil simple, mais efficace pour un formalisme beaucoup plus complexe (B) où leurs systèmes sont exprimés de manière compliquée (traduction de Lustre).

7.3.2 Lustre sans B

C'est pour toutes les raisons évoquées dans les lignes précédentes que nous avons décidé d'abandonner la piste de la traduction de Lustre vers B, au profit de la recherche d'un raffinement propre à Lustre, raffinement qui serait inspiré de celui de B, mais qui en serait indépendant.

Nous avons vu au début de ce chapitre que l'une des caractéristiques importantes d'un développement par raffinements était le fait d'utiliser le même formalisme pour décrire toutes les étapes depuis les premières spécifications jusqu'à l'implémentation. La méthode B utilise un tel formalisme, spécialement créé pour elle. Or, en Lustre, nous en avons déjà un : il suffit d'utiliser le langage lui-même. En effet, les propriétés invariantes sur les programmes Lustre peuvent être exprimées sous forme de nœuds particuliers, appelés observateurs synchrones [21].

Un autre point important est la possibilité d'effectuer des preuves, car le raffinement produit des obligations qui doivent être vérifiées. Or nous disposons pour Lustre d'un éventail d'outils de vérification, parmi lesquels on trouve Lesar, nbac [25] et Gloups (cf. chapitre 6).

Nous avons donc toutes les prémisses pour établir un calcul de raffinement adapté à Lustre. Nous présentons le résultat de nos recherches dans ce domaine dans les chapitres qui suivent.

8 – Raffinement pour Lustre

Nous établissons le calcul de raffinement pour Lustre en deux phases : d'abord, nous instaurons un cadre général correct, puis à l'intérieur de ce cadre, nous délimitons une partie adaptée à la fois au langage et à nos moyens de preuve pratiques. Nous terminons par une discussion des points les plus importants du raffinement proposé.

Pour établir un raffinement pour le langage Lustre, nous allons procéder en deux étapes. D'abord, nous allons formaliser le problème et donner un premier raffinement assez général, raffinement qui s'appliquera à une large classe de systèmes, dont ceux qui peuvent être décrits en Lustre. Ce premier raffinement constituera un cadre de travail correct pour la suite : dans un deuxième temps, nous resserrerons ce cadre pour l'adapter à Lustre, ce qui nous donnera en fin de compte un calcul de raffinement plus précis et plus intéressant du point de vue pratique.

Nous avons publié les résultats théoriques de ce chapitre, et notamment le raffinement du temps à la conférence SLAP'05 [35].

8.1 Cadre général

Dans cette première partie, nous allons créer un raffinement pour des systèmes réactifs non-déterministes. Avant d'énoncer le raffinement lui-même et ses propriétés de correction, nous allons d'abord définir quels sont précisément les systèmes auxquels nous nous intéressons.

8.1.1 Systèmes réactifs non-déterministes

Relations De manière générale, les systèmes que nous allons considérer seront des *relations* entre les entrées et les sorties. En effet, choisir des relations plutôt que des fonctions est une manière commode d'introduire le non-déterminisme dans la modélisation. Ainsi, si les entrées possèdent un certain type T et les sorties un type T', alors un système S sera $S \subseteq T \times T'$. Schématiquement, nous allons noter :

$$T \quad \overline{\qquad S \qquad} \quad T'$$

Pour faire simple, nous allons confondre les sorties et les variables locales, en supposant que le type T' englobe les deux éléments. En effet, du point de vue des calculs, ces deux catégories jouent le même rôle ; le problème de visibilité sera abordé à un autre niveau.

Nous pouvons remarquer que lorsque nous allons nous restreindre aux systèmes décrits en Lustre à la section 8.2, la relation S sera toujours totale, car elle sera exprimée par un nœud Lustre. Or, les spécifications sont généralement des relations partielles : il nous faut donc un moyen de les décrire, même en Lustre. C'est pourquoi nous introduisons également des pré- et postconditions.

Précondition et postcondition Soit P un prédicat sur les entrées ($P \subseteq T$) et Q un prédicat sur les entrées et les sorties ($Q \subseteq T \times T'$). On dira que P est la précondition du système S et que Q est sa postcondition. Ces deux éléments viennent enrichir le schéma précédent :

$$P : \quad T \quad \overset{S}{\text{———}} \quad T' \quad : Q$$

Informellement, le rôle de P est de restreindre l'ensemble des entrées admises de S : le système est censé réagir à toutes les entrées qui vérifient la précondition, mais son comportement n'est pas défini pour les autres entrées.

La postcondition Q établit une abstraction des calculs du système : les sorties d'un système correct doivent vérifier sa postcondition, dès lors que l'appel a été fait dans le précondition.

À la différence de S, la postcondition ne doit pas faire intervenir les variables locales, mais seulement les entrées et les « vraies » sorties. C'est donc la postcondition qui gère la visibilité.

Correction et réactivité Pour formaliser le rôle de la précondition et de la postcondition, nous allons définir deux (méta-)propriétés fondamentales de nos systèmes, la correction et la réactivité.

S est **correct** par rapport à P et Q si et seulement si $[\text{COR}_\text{S}]\ \forall (x, y) \in T \times T'.\ P(x) \wedge S(x, y) \Rightarrow Q(x, y)$ c'est-à-dire si la précondition et la relation impliquent la postcondition
S est **réactif** par rapport à P si et seulement si $[\text{REA}_\text{S}]\ \forall x \in T.\ P(x) \Rightarrow \exists y \in T'.\ S(x, y)$ c'est-à-dire si à toute entrée dans la précondition correspond une sortie

FIG. 8.1 – Les deux propriétés fondamentales des systèmes

Le raffinement général que nous allons proposer maintenant est centré autour de ces deux notions.

8.1.2 Un premier raffinement

Notre but est de définir un raffinement qui assure que si le système S est raffiné par S', on devrait pouvoir utiliser S' à la place de S, tout en conservant la correction et la réactivité de S.

Deux systèmes Dans un souci de généralité, nous supposerons que les deux systèmes S et S' n'ont pas nécessairement le même type : $S \subseteq T \times T'$ et $S' \subseteq U \times U'$. Pour relier les deux systèmes, nous supposerons l'existence de deux relations de liaison, σ et τ : $\sigma \subseteq T \times U$ et $\tau \subseteq T' \times U'$. De plus, les deux systèmes ont chacun leur propre précondition et postcondition. Schématiquement, la situation donc est celle-ci :

$$\begin{array}{rlcccl} \text{système abstrait } \{ & P : & T & \overset{S}{\text{———}} & T' & : Q \\ \text{changements de variables } \{ & & \sigma \big| & & \tau \big| & \\ \text{système concret } \{ & P' : & U & \overset{S'}{\text{———}} & U' & : Q' \end{array}$$

Conditions de raffinement Pour définir les conditions du raffinement, nous nous sommes inspirés de l'esprit de la méthode B [3] et de notre travail plus ancien [32], que nous avons généralisé. Nous dirons que le système S' raffine le système S (noté $S \sqsubseteq S'$) si les trois conditions de la figure 8.2 sont vérifiées.

$[\mathrm{COR}_{S \to S'}]$	$\forall (t, t', u, u') \in T \times T' \times U \times U'.\ P(t) \wedge \sigma(t, u) \wedge S'(u, u') \wedge \tau(t', u') \Rightarrow S(t, t')$
$[\mathrm{REA}_{S \to S'}]$	$\forall t \in T.\ P(t) \Rightarrow (\exists u \in U.\ \sigma(t, u)) \wedge (\forall u \in U.\ \sigma(t, u) \Rightarrow P'(u))$
$[\mathrm{REA}'_{S \to S'}]$	$\forall (u, u') \in U \times U'.\ P'(u) \wedge S'(u, u') \Rightarrow \exists t' \in T'.\ \tau(t', u')$

FIG. 8.2 – Conditions de raffinement

Ces conditions de raffinement expriment formellement notre vision du rôle d'un raffinement :

- $[\mathrm{COR}_{S \to S'}]$ assure la préservation de la correction. Cette propriété impose que tout calcul que le système S' fait sur une entrée admissible de S (c'est-à-dire une entrée vérifiant P) aboutit à une valeur de sortie qui aurait pu être produite par S. Ainsi, l'utilisateur ne pourra pas distinguer si c'est le système S ou S' qui a fait le calcul.
- $[\mathrm{REA}_{S \to S'}]$ assure, pour une part, la réactivité de S'. Cette propriété impose que toute entrée admissible de S reste admissible dans S'. La précondition peut donc être élargie pendant le raffinement, mais pas restreinte.
- $[\mathrm{REA}'_{S \to S'}]$ est la seconde partie de la réactivité : tout calcul admissible de S' aboutit à une sortie qui a une contre-partie dans S.

Propriétés du raffinement Nous avons prouvé[1] que si les trois *obligations de preuve* $[\mathrm{COR}_{S \to S'}]$, $[\mathrm{REA}_{S \to S'}]$ et $[\mathrm{REA}'_{S \to S'}]$ sont vraies, alors S' préserve la correction et la non-réactivité de S. Plus précisément, nous avons prouvé que

$$\frac{[\mathrm{COR}_{S \to S'}]}{\begin{array}{l}\vdash \forall Q \subseteq T \times T'.\ (\forall (t, t') \in T \times T'.\ P(t) \wedge S(t, t') \Rightarrow Q(t, t')) \Longrightarrow \\ \quad (\forall (t, t', u, u') \in T \times T' \times U \times U'.P(t) \wedge \sigma(t, u) \wedge S'(u, u') \wedge \tau(t', u') \Rightarrow Q(t, t'))\end{array}}$$

et

$$\frac{[\mathrm{COR}_{S \to S'}] \wedge [\mathrm{REA}_{S \to S'}] \wedge [\mathrm{REA}'_{S \to S'}]}{\vdash [S' \text{ est réactif}] \Rightarrow [S \text{ est réactif}]}$$

Nous obtenons donc bien le comportement voulu pour la correction : si S vérifie une postcondition Q, alors S' la vérifie aussi. Ainsi, les deux systèmes ne sont pas distinguables de l'extérieur. La postcondition complète de S' est alors « $Q \wedge Q'$ », si on fait abstraction du typage : la postcondition est donc restreinte par le raffinement.

Le résultat pour la réactivité peut sembler plus surprenant : si S' est réactif, S l'est aussi. Or, on aurait naturellement tendance à souhaiter l'implication inverse. En fait, on retrouve ici un problème semblable à celui de la faisabilité en B : pour prouver qu'une spécification abstraite (S) est réactive, on en exhibera une implémentation concrète (S'). Cette implémentation sera écrite en Lustre, donc elle sera automatiquement réactive.

En fait, notre raffinement assure qu'une spécification non-réactive ne pourra jamais être raffinée en une implémentation réactive ; par contre une spécification réactive pourra donner lieu à une implémentation.

[1] Nous avons retranscrit en PVS et prouvé les propriétés mentionnées dans cette partie.

8.1.3 Transitivité du raffinement

Le raffinement défini dans la section précédente possède une autre propriété importante, la transitivité. En effet,

$$S \sqsubseteq S' \wedge S' \sqsubseteq S'' \Rightarrow S \sqsubseteq S''$$

Ainsi, nous allons pouvoir développer des systèmes en plusieurs étapes, faisant du raffinement incrémental : si chaque nouvelle étape raffine la précédente, alors par transitivité, la dernière étape (l'implémentation) raffinera bien les spécifications de départ.

Plus précisément, soit trois systèmes $S \subseteq T \times T'$, $S' \subseteq U \times U'$ et $S'' \subseteq V \times V'$, avec leurs préconditions et postcondition. Soit également quatre relations de changement de variables, comme indiqué sur le schéma suivant :

$$\begin{array}{lccccc}
P : & T & \xrightarrow{\quad S \quad} & T' & : Q \\
 & \sigma \downarrow & & \tau \downarrow & \\
P' : & U & \xrightarrow{\quad S' \quad} & U' & : Q' \\
 & \rho \downarrow & & \varphi \downarrow & \\
P'' : & V & \xrightarrow{\quad S'' \quad} & V' & : Q''
\end{array}$$

Alors on peut prouver que

$$\frac{[\mathrm{COR}_{S \to S'}] \wedge [\mathrm{REA}_{S \to S'}] \wedge [\mathrm{COR}_{S' \to S''}]}{\begin{array}{l}\vdash \forall Q \subseteq T \times T'.\ (\forall t, t' \in T \times T'.\ P(t) \wedge S(t,t') \Rightarrow Q(t,t')) \Longrightarrow \\ \quad (\forall (t,t',u,u',v,v') \in T \times T' \times U \times U' \times V \times V'. \\ \quad\quad P(t) \wedge \sigma(t,u) \wedge \rho(u,v) \wedge S''(v,v') \wedge \varphi(u',v') \wedge \tau(t',u') \Rightarrow Q(t,t'))\end{array}}$$

et

$$\frac{\begin{array}{l}[\mathrm{COR}_{S \to S'}] \wedge [\mathrm{REA}_{S \to S'}] \wedge [\mathrm{REA}'_{S \to S'}] \wedge \\ [\mathrm{COR}_{S' \to S''}] \wedge [\mathrm{REA}_{S' \to S''}] \wedge [\mathrm{REA}'_{S' \to S''}]\end{array}}{\vdash [\mathrm{S}'' \text{ est réactive}] \Rightarrow [\mathrm{S} \text{ est réactive}]}$$

ce qui établit bien la transitivité.

8.1.4 Bilan du premier raffinement

Nous venons de proposer un cadre pour le raffinement, cadre qui présente des analogies directes avec la méthode B, comme décrit sur la figure 8.3.

Pour motiver la suite du chapitre, nous pouvons formuler plusieurs remarques sur le raffinement proposé dans cette section :

- Les programmes Lustre font bien partie des systèmes que nous avons introduits à la section 8.1.1. En effet, un programme Lustre est une fonction entre des flots d'entrée et des flots de sortie (et des variables locales) ; en particulier, on peut considérer un tel programme comme une relation sur les flots.
- Les préconditions et les postconditions peuvent être également exprimées en Lustre, grâce aux observateurs synchrones [21].

méthode B	**notre raffinement**
machine abstraite à une opération	relation S avec sa précondition P et postcondition Q
variables	sorties et variables locales (décrites par T')
opération	relation S
entrées de l'opération	entrées (décrites par T)
sorties de l'opération	sorties
invariant	postcondition Q
précondition	précondition P
invariant de liaison	τ[2]

FIG. 8.3 – Analogies de notre raffinement avec la méthode B

- Le raffinement que nous avons proposé est un peu trop général : par exemple, il serait tout à fait possible de raffiner un système S par S' réduit à la fonction identité ; tous les calculs seraient alors dans les relations de changement de variables σ et τ. Or cela ne correspond pas à l'idée que nous avons d'un développement par raffinements : nous allons donc devoir imposer des restrictions supplémentaires.
- Dans un autre ordre d'idées, nous pouvons remarquer que les obligations de preuve du raffinement ($[\mathrm{COR}_{S\to S'}]$, $[\mathrm{REA}_{S\to S'}]$ et $[\mathrm{REA}'_{S\to S'}]$) comportent aussi des quantifications existentielles. Cela pose un double problème : nous ne savons pas les traduire en Lustre et nous n'avons pas d'outils à notre disposition pour les prouver.

8.2 Raffinement adapté

Dans cette partie, nous allons étudier plusieurs adaptations du raffinement proposé dans la partie précédente. Ces adaptations visent d'une part à mieux cibler le langage Lustre et d'autre part à lever certaines difficultés pratiques.

Le but que nous poursuivons est d'établir un raffinement *pour* Lustre, qui soit exprimé *en* Lustre : ainsi, le langage fournira son propre formalisme et tous les outils de preuve dont nous disposons pourront être utilisés pour prouver les obligations de raffinement.

Dans l'ordre, nous allons introduire les contraintes suivantes :
- Une contrainte de sous-typage des domaines (T, U) et des co-domaines (T', U') règlera une difficulté pratique.
- Ensuite, l'introduction d'une modélisation des flots conduira à la possibilité d'un raffinement du temps.
- Finalement, nous allons parachever le raffinement temporel en adaptant les relations de changement de variables σ et τ.

8.2.1 Sous-typage des domaines et co-domaines

Un problème pratique Nous avons déjà mentionné la question de la quantification existentielle dans les obligations de preuve $[\mathrm{REA}_{S\to S'}]$ et $[\mathrm{REA}'_{S\to S'}]$:

$$[\mathrm{REA}_{S\to S'}] \quad \forall t \in T.\ P(t) \Rightarrow (\exists \mathbf{u} \in \mathbf{U}.\ \sigma(\mathbf{t}, \mathbf{u})) \wedge (\forall u \in U.\ \sigma(t, u) \Rightarrow P'(u))$$
$$[\mathrm{REA}'_{S\to S'}] \quad \forall (u, u') \in U \times U'.\ P'(u) \wedge S'(u, u') \Rightarrow \exists \mathbf{t}' \in \mathbf{T}'.\ \tau(\mathbf{t}', \mathbf{u}')$$

[2] L'invariant de liaison en B concerne les variables de la machine abstraite : ainsi, seul τ correspond à cette notion. σ correspondrait à l'élargissement de la précondition.

Cette quantification constitue un problème d'ordre pratique, car une formule quantifiée existentiellement ne peut pas être exprimée avec un observateur synchrone. Notre première tâche sera donc d'éliminer ces quantifications.

Solution pour REA' Considérons la condition $[\mathrm{REA}'_{\mathrm{S}\to\mathrm{S}'}]$: elle exprime le fait que tout résultat u' calculé par le système concret S possède une contre-partie t' dans le domaine abstrait.

$$\begin{array}{lccccl} P: & T & \xrightarrow{\quad S \quad} & T'\,(\mathbf{t}') & & : Q \\ & \sigma\big| & & \tau\big| & \uparrow & \\ P': & U & \xrightarrow{\quad S' \quad} & U'\,(\mathbf{u}') & & : Q' \end{array}$$

Or il existe un moyen structurel d'assurer cette existence, qui est le sous-typage. En effet, si U' est un sous-type de T' (noté $U' \preceq T'$), alors tout élément de U' sera automatiquement élément de T', si bien que la condition $[\mathrm{REA}'_{\mathrm{S}\to\mathrm{S}'}]$ sera toujours vérifiée. C'est pour cette raison que nous ajoutons à notre raffinement la contrainte supplémentaire

$$U' \preceq T'$$

Solution pour REA La partie existentielle de $[\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}]$ est en fait le dual de $[\mathrm{REA}'_{\mathrm{S}\to\mathrm{S}'}]$: on demande que toute entrée abstraite t ait une contre-partie concrète u. Nous pouvons donc utiliser le sous-typage encore une fois pour rendre cette condition toujours vraie. Ici, les rôles des domaines abstrait et concret sont inversés par rapport au cas précédent, si bien que la contrainte supplémentaire est :

$$T \preceq U$$

Conséquences pratiques La conséquence immédiate d'imposer $U' \preceq T'$ et $T \preceq U$ est que la condition $[\mathrm{REA}'_{\mathrm{S}\to\mathrm{S}'}]$ devient inutile (car toujours vraie) et que $[\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}]$ se simplifie en

$$[\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}] \quad \forall t \in T.\ P(t) \Rightarrow (\forall u \in U.\ \sigma(t,u) \Rightarrow P'(u))$$

ou de manière équivalente en

$$[\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}] \quad \forall (t,u) \in T \times U.\ (P(t) \wedge \sigma(t,u)) \Rightarrow P'(u)$$

Les quantifications existentielles ont donc bien disparu.

Cependant, il existe une autre conséquence de nos choix. Nous avons vu au chapitre 2 que les nœuds Lustre pouvaient avoir un nombre quelconque d'entrées et de sorties, chacune d'elles étant un flot. Il est donc légitime de considérer les domaines et les co-domaines de nos systèmes comme des produits cartésiens de plusieurs types :

$$\begin{array}{lccccl} P: & T = \prod_{i=1}^{j} T_i & \xrightarrow{\quad S \quad} & T' = \prod_{i=1}^{k} T'_i & & : Q \\ & \sigma\big| & & \tau\big| & & \\ P': & U = \prod_{i=1}^{l} U_i & \xrightarrow{\quad S' \quad} & U' = \prod_{i=1}^{m} U'_i & & : Q' \end{array}$$

Le sous-typage des produits suit les règles suivantes : pour tous types A et B, $A \times B \preceq A$ et $A \times B \preceq B$, car il existe des projections canoniques de $A \times B$ dans A, respectivement B. De plus, si pour deux types A' et B', $A \preceq A'$ et $B \preceq B'$, grâce à la transitivité du sous-typage, nous obtenons : $A \times B \preceq A'$ et $A \times B \preceq B'$.

Sous-typage des domaines La contrainte $T \preceq U$ se traduit par l'existence d'une fonction injective $\alpha : \{1, \ldots, l\} \to \{1, \ldots, j\}$ telle que $\forall i \in \{1, \ldots, l\}$. $T_{\alpha(i)} \preceq U_i$. En clair, on impose qu'il y ait au moins autant d'entrées abstraites que d'entrées concrètes et que les entrées qui se correspondent (par α) dans les deux domaines soit liées par une contrainte de sous-typage.

Ainsi, lorsqu'un système S est raffiné, il est possible d'« oublier » certaines entrées (elles ne figureront plus dans le système concret S') et toutes celles qui restent auront des types égaux ou plus grands que les entrées abstraites correspondantes. Par exemple, il sera possible de raffiner un système à trois entrées (`bool`, `int`, `bool`) en un système à deux entrées (`bool`, `bool`).

Sous-typage des co-domaines De manière duale, la contrainte des co-domaines, $U' \preceq T'$, se réduit à l'existence d'une injection $\beta : \{1, \ldots, k\} \to \{1, \ldots, m\}$ telle que $\forall i \in \{1, \ldots, k\}$. $U'_{\beta(i)} \preceq T'_i$. Par conséquent, il est possible de définir de nouvelles sorties pour le système concret, à condition de conserver les sorties abstraites. De plus, une sortie « conservée » devra avoir un type concret qui soit sous-type du type abstrait.

Il est clair que ces nouvelles contraintes (l'existence d'une injection, le sous-typage etc.) ne peuvent pas être exprimées en Lustre. Mais ce n'est pas nécessaire, car ce sont des contraintes de typage qui peuvent être vérifiées statiquement sur les programmes Lustre. D'ailleurs, nous allons voir dans la prochaine section que ces contraintes vont encore se simplifier.

8.2.2 Prise en compte des flots

Nous avons établi le premier raffinement en considérant des types généraux, comme par exemple T. Nous venons d'imposer certaines nouvelles contraintes et, pour nous approcher de la réalité de Lustre, nous avons décidé de considérer seulement des produits cartésiens de types, comme $\prod_{i=1}^{j} T_i$. Dans cette section, nous allons contraindre les types encore davantage, pour tenir compte de la nature des variables en Lustre.

Les flots En effet, chaque variable d'un programme Lustre est un flot, c'est-à-dire une suite finie ou infinie d'éléments d'un type scalaire. Soit $\mathbb{N}$ l'ensemble des entiers naturels. Soit $\mathbb{N}_k$ l'ensemble des k premiers entiers naturels où $k \leq \omega$ est un ordinal : $\mathbb{N}_k = \{n \in \mathbb{N} \mid n < k\}$. On appellera Σ l'ensemble de tous les $\mathbb{N}_k$:

$$\Sigma = \bigcup_{k=0}^{\omega} \mathbb{N}_k$$

Si D est un domaine scalaire, alors un flot f est une fonction

$$f : X \to D \text{ où } X \in \Sigma$$

Sous-typage des flots Le sous-typage des flots suit donc les règles de sous-typage des fonctions :

$$\mathbb{N}_i \to D \preceq \mathbb{N}_j \to D' \quad \Leftrightarrow \quad \begin{cases} D \preceq D' \\ \mathbb{N}_j \preceq \mathbb{N}_i \end{cases}$$

- Les domaines scalaires sont ordonnés par le sous-typage comme attendu ($D \preceq D'$), mais nous allons utiliser plus simplement $D = D'$. En effet, le langage Lustre, tel que nous l'avons présenté dans le chapitre 2, ne contient que trois types : les entiers, les réels et les booléens. D'un point de vue purement mathématique, les entiers pourraient être considérés comme un sous-type des réels, mais étant données leurs représentations radicalement différentes dans les ordinateurs, il faut abandonner cette idée en pratique.
  Il est vrai que la version 6 du langage Lustre permet de définir des types structurés, comme des tableaux ou des enregistrements : on pourrait effectuer des « vrais » sous-typages sur ces structures. Néanmoins, le sous-typage des données scalaires est un problème bien étudié [1] et ne nous intéresse pas vraiment ici : nous allons conserver la contrainte $D = D'$.

- Les ensembles d'indices $\mathbb{N}_i$ et $\mathbb{N}_j$ sont ordonnés dans l'ordre inverse de la contrainte de départ : c'est le phénomène de la contra-variance. Étant donné que $\mathbb{N}_i$ et $\mathbb{N}_j$ représentent des ensembles de nombres entiers, $\mathbb{N}_j \preceq \mathbb{N}_i$, c'est-à-dire la contrainte « tout élément de $\mathbb{N}_j$ est élément de $\mathbb{N}_i$ » revient simplement à

  $$j \geq i$$

  En conséquence, le flot $f_2 : \mathbb{N}_j \to D'$ est « plus long » que le flot $f_1 : \mathbb{N}_i \to D$.

Sous-typage des flots dans notre raffinement Remettons à présent la règle de sous-typage des flots que nous venons d'énoncer dans le contexte établi dans la section précédente.

- La contrainte de **sous-typage sur les domaines** se traduit, nous l'avons vu, par l'existence d'une injection $\alpha : \{1, \ldots, l\} \to \{1, \ldots, j\}$ telle que $\forall i \in \{1, \ldots, l\}$. $T_{\alpha(i)} \preceq U_i$. Posons $T_i = \mathbb{N}_{x(i)} \to D_i$ et $U_i = \mathbb{N}_{y(i)} \to D'_i$ où x et y sont les fonctions qui donnent la longueur des flots. Pour chaque i, il nous faut donc avoir $\mathbb{N}_{x(\alpha(i))} \to D_{\alpha}(i) \preceq \mathbb{N}_{y(i)} \to D'_i$, ce qui donne en fin de compte

  $$\begin{cases} D_{\alpha(i)} = D'_i \\ y(i) \geq x(\alpha(i)) \end{cases}$$

  En langage clair, cette dernière formule affirme que les entrées concrètes doivent avoir le même type scalaire que les entrées abstraites correspondantes, mais que les flots concrets peuvent être plus longs que les flots abstraits ou, en d'autres termes, que *les entrées concrètes peuvent avoir une horloge plus rapide que les entrées abstraites correspondantes.*

- Par un raisonnement analogue, nous pouvons décortiquer le **sous-typage des co-domaines**. Celui-ci est défini par l'injection $\beta : \{1, \ldots, k\} \to \{1, \ldots, m\}$ et la condition $\forall i \in \{1, \ldots, k\}$. $U'_{\beta(i)} \preceq T'_i$. En posant $T'_i = \mathbb{N}_{x'(i)} \to \Delta_i$ et $U'_i = \mathbb{N}_{y'(i)} \to \Delta'_i$, la condition pour chaque i se réécrit en $\mathbb{N}_{y'(\beta(i))} \to \Delta'(i) \preceq \mathbb{N}_{x'(i)} \to \Delta_i$, d'où

  $$\begin{cases} \Delta_{\beta(i)} = \Delta'_i \\ x'(i) \geq y'(\beta(i)) \end{cases}$$

  De manière duale, chaque sortie concrète a le même type scalaire que la sortie abstraite correspondante, et le niveau concret est plus lent que le niveau abstrait.

Nous n'allons pas nous apesantir sur les résultats de cette section, car nous allons leur trouver une formulation plus simple à la section suivante.

8.2.3 Les relations de changement de variables

Reprenons le schéma général du raffinement :

$$\begin{array}{lccc} P: & T & \xrightarrow{S} & T' \quad :Q \\ & \sigma\downarrow & & \tau\downarrow \\ P': & U & \xrightarrow{S'} & U' \quad :Q' \end{array}$$

Jusqu'à maintenant, toutes les restrictions que nous avons imposées à ce schéma concernent uniquement les types. En effet, les contraintes sur les types nous ont permis d'éliminer les quantifications existentielles dans les obligations de preuve du raffinement, puis d'introduire la notion des flots de données. Ainsi, nous avons répondu à l'une des remarques formulées à la section 8.1.4. Cependant, une autre remarque reste toujours sans réponse : celle qui concerne les relations σ et τ.

Projections En effet, nous avons soulevé la question de savoir dans quelle mesure les calculs devaient figurer dans ces relations. Or σ et τ représentent dans notre esprit simplement des changements de variables, c'est-à-dire des moyens de liaison entre les types T et U, respectivement T' et U'. Comme telles, ces relations devraient contenir aussi peu de calculs que possible, car elles existent plus par nécessité de cohérence théorique que par choix pratique.

Dans le cas idéal, σ et τ seraient donc des identités. Nous avons vu que cela n'est pas possible dans le cas général, car notre raffinement, même avec les restrictions introduites dans les sections 8.2.1 et 8.2.2, permet d'effacer des entrées et de créer de nouvelles sorties. Ainsi, σ et τ devront être au minimum des projections d'un espace vers l'autre.

On peut remarquer que le concept même de projections contient l'idée d'une orientation possible des deux relations. En fait, d'autres éléments viendront corroborer cette idée.

Transformateurs de données Il est clair que dans un développement par raffinements, le système abstrait représente les spécifications et le système concret l'implémentation. Or l'environnement (les autres systèmes) ne voient que les spécifications : les entrées fournies et les sorties attendues seront celles du système abstrait.

Si le système concret décide de redéfinir ses entrées et/ou sorties par rapport au niveau abstrait, la relation σ devra assurer la transformation des entrées données par l'environnement en des entrées concrètes, puis, après le calcul, la relation τ traduira les sorties concrètes en le format abstrait, attendu par l'environnement. Ainsi, les deux relations auraient tendance à marcher en sens inverses :

$$\begin{array}{lccc} P: & T & \xrightarrow{S} & T' \quad :Q \\ & \downarrow\sigma & & \tau\uparrow \\ P': & U & \xrightarrow{S'} & U' \quad :Q' \end{array}$$

Il est intéressant de noter que le sens du flux de données (descendant pour σ et montant pour τ) correspond au sens des projections dont nous avons parlé plus haut.

Transformateurs de temps Nous avons vu dans la section 8.2.2 que les entrées concrètes avaient la possibilité d'être sur une horloge plus rapide que les entrées abstraites. Du point de vue du système

abstrait, il est impossible d'exprimer ce fait en Lustre, car il faudrait pouvoir définir des horloges plus rapides que l'horloge de base. Or ceci est rendu impossible par le calcul d'horloges.

Ainsi, il nous faut adopter le point de vue du système concret. Pour celui-ci, les entrées abstraites sont plus lentes que ses propres entrées, si bien que la relation σ doit effectuer une opération semblable à l'opérateur `current` pour ramener les flots abstraits (lents) au niveau concret (rapide).

De manière duale, à la sortie du système concret, la relation τ devra effectuer l'opération `when` pour ralentir les flots.

Bilan des changements de variables Lorsque toutes les remarques précédentes sont considérées ensemble, il apparaît que le schéma de départ a un peu changé : les relations σ et τ sont devenues des *fonctions*.

$$\begin{array}{lccccl} P: & T & \xrightarrow{\quad S \quad} & T' & & :Q \\ & \sigma\downarrow & & & \tau\uparrow & \\ P': & U & \xrightarrow{\quad S' \quad} & U' & & :Q' \end{array}$$

Une partie des contraintes sur les changements de variables sont des contraintes de typage qui peuvent être vérifiées statiquement : c'est le cas de la conservation du type scalaire des flots, de l'effacement des entrées et de la création de nouvelles sorties. L'autre partie, celle qui concerne les changements de l'échelle temporelle, peut être exprimée en Lustre.

- Soit un type T et une fonction Lustre $f : Bool \times T \rightarrow Bool$. On suppose que cette fonction préserve les horloges, c'est-à-dire que son profil d'horloge est $f :: \alpha \times \alpha \rightarrow \alpha$. Étant donnée f, on peut définir la fonction clk telle que

$$\begin{array}{rcl} clk & : & T \rightarrow Bool \\ clk(x) & = & ck \\ \text{où } ck & = & true \rightarrow pre(f(ck, current(@, ck, x))) \end{array}$$

  Les contraintes sur les horloges imposent que la fonction clk ait le profil temporel $clk :: ck \rightarrow \alpha$ et que par ailleurs ck soit sur l'horloge α. Ainsi, la sortie de clk est plus rapide que son entrée !

- La fonction clk n'est donc pas une fonction Lustre, car elle ne vérifie pas le calcul d'horloges. Mais à par cela, elle est bien définie. On peut alors définir le changement de variables σ comme

$$\begin{array}{rcl} \sigma & : & T \rightarrow Bool \times T \\ \sigma(x) & = & clk(x), x \end{array}$$

  Ce changement de variables préserve x, mais accelère son horloge, ce qui revient à faire répéter certaines valeurs.

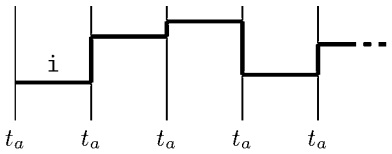
- **Exemple** Soit le nœud `Plus2` qui à chaque instant incrémente son entrée de 2.

```
node Plus2(i : int) returns (s : int);
let
  s = i + 2 ;
tel ;
```

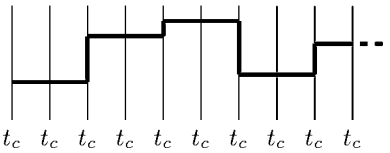
  Pour des raisons d'efficacité, on pourrait vouloir implémenter l'opération « +2 » comme deux opérations « +1 » successives. Ainsi, le nœud qui raffinerait `Plus2` aurait besoin de fonctionner deux fois plus rapidement que `Plus2`. Une telle accélération est possible, en prenant la fonction f suivante :

```
node f(b:bool; x:T) returns (c:bool)
let
  c = true → (not pre c) ;
tel ;
```

Comme f est vraie tous les deux tops d'horloge, elle permet une accélération d'un facteur 2. Les schémas suivants présentent l'idée du fonctionnement :



Ici on présente un exemple d'évolution des valeurs de l'entrée `i` (c'est la ligne épaisse), telle que le nœud `Plus2` la voit. Les changements de la valeur de l'entrée correspondent aux « tops » de l'horloge de base de `Plus2`. Ces instants sont marqués comme t_a, pour « top abstrait ».



La même évolution des valeurs de `i`, vue depuis un nœud qui a profité du changement de variables induit par f_2. L'entrée reste la même, mais l'horloge de base est deux fois plus rapide : ainsi, le nœud raffiné possède assez de temps pour appliquer deux opérations « $+1$ » avant que la sortie soit attendue.

- Ainsi, raffiner un système $S(x, y)$ en $S'((ck', x'), y') = S'(\sigma(x), y)$ peut être décrit, **du point de vue du système** S', par :
  - l'identité sur x, ce qui est une fonction Lustre
  - et une pré-condition supplémentaire

$$Clk\_P'(ck', x') = (ck' = true \rightarrow pre(f(ck', current(@, ck', x'))))$$

  Du point de vue de S', la pré-condition Clk_P' est une expression Lustre normale, car en particulier, elle n'implique pas d'horloge plus rapide que l'horloge de base de S'. Ainsi, nous avons contourné la difficulté de formalisation par un changement de point de vue.

- De manière duale, le changement de variables τ assure le passage d'un monde rapide (concret) vers un monde lent (abstrait) : c'est l'opération `when ck'` qui effectue le changement nécessaire.

Le lecteur trouvera un résumé du raffinement tenant compte de toutes les remarques précédentes dans la section 8.3.1.

8.2.4 Les relations en Lustre

Il reste à expliquer un dernier point, qui est la manière dont nous projetons de décrire les relations S et S' en Lustre. En effet, les programmes Lustre classiques définissent des fonctions, mais il est facile d'élargir le langage aux relations, en utilisant les clauses `assert`. Par exemple, le « nœud » suivant décrit la relation $R \subseteq \mathbb{Z} \times \mathbb{Z}$ telle que $(a, b) \in R \Leftrightarrow a \leq b$ où a est donné et b est attendu en sortie :

```
node R(a : int) returns (b : int);
let
  assert (a <= b) ;
tel;
```

Syntaxiquement, le nœud `R` est correct ; tout ce qui l'empêche d'être un nœud Lustre compilable est l'absence d'une équation pour `b`. Ainsi, l'extension de Lustre aux relations est particulièrement simple.

8.3 Bilan du raffinement

Dans ce chapitre, nous avons établi un cadre pour le raffinement, cadre que nous avons progressivement rétréci pour tenir compte des particularités de Lustre et de nos moyens de preuve. À présent, résumons le raffinement obtenu.

8.3.1 Résumé du raffinement

Vérifications syntaxiques et statiques

1. $\sigma : T \longrightarrow T'$ est une fonction Lustre
2. $\tau : U' \longrightarrow U$ est une fonction Lustre
3. $f : Bool \times T' \longrightarrow Bool$ est une fonction Lustre qui préserve les horloges
4. $Ck\_P'(ck', x') = (ck' = true \rightarrow pre(f(ck', current(@, ck', x'))))$
5. $P'(ck', x') = Ck\_P'(ck', x') \wedge D\_P'(x')$

Obligations de preuve

6. $[\mathrm{COR}_{S \rightarrow S'}]\ \forall (x, y') \in T \times U'. P(x) \wedge Ck\_P'(ck', \sigma(x)) \wedge S'((ck', \sigma(x)), y') \Rightarrow S(x, \tau(y'))$
7. $[\mathrm{REA}_{S \rightarrow S'}]\ \forall x \in T.\ P(x) \Rightarrow D\_P'(\sigma(x))$
8. $[\mathrm{COR}_{S'}]\ \forall (x', y') \in U \times U'.\ P'(x') \wedge S'(x', y') \Rightarrow Q'(x', y')$

On peut remarquer que
- L'obligation de preuve (7) ne fait pas intervenir la précondition sur les horloges Ck_P', qui ne fait qu'exprimer le sur-échantillonnage obtenu par le changement de variables.
- Par contre, une fois que le raffinement est établi, la précondition Ck_P devient partie intégrante du système raffiné, car elle est ajoutée à la précondition P' : c'est ce qu'exprime la condition (5). De manière générale, les préconditions ont donc deux composantes : une pour les horloges (Ck_P) et une pour les données (D_P').

8.3.2 Analogie avec les circuits

Pour donner une image, nous allons illustrer le raffinement proposé sur les circuits électroniques. La situation de départ est schématisée sur la figure 8.4 : le système S reçoit de son environnement un certain nombre d'entrées, dont l'horloge de base ck, et fournit quelques sorties.

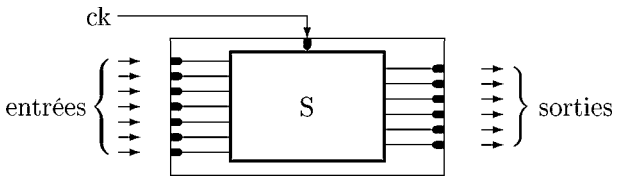


FIG. 8.4 – Schéma d'un circuit

On peut remplacer (raffiner) le circuit S par un circuit S', comme celui de la figure 8.5 :
- Le système S' a sa propre horloge de base ck', qui est éventuellement plus rapide que ck.
- S' peut avoir moins d'entrées et plus de sorties que S.

- Comme les entrées fournies par l'environnement évoluent à la vitesse de ck, on doit les stocker dans des registres pour pouvoir les exploiter dans S', qui évolue plus rapidement. C'est le rôle de σ.
- De même, lorsque S' fournit des sorties, celles-ci doivent être stockées en attendant que l'environnement puisse les lire. Cette opération est assurée par τ.

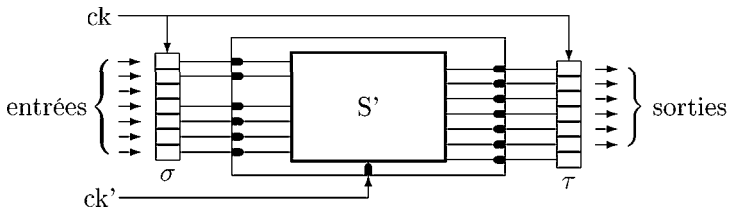


FIG. 8.5 – Schéma du circuit raffiné

8.3.3 De l'aspect temporel du raffinement proposé

La possibilité de raffiner explicitement le temps est certainement l'un des aspects les plus intéressants de notre raffinement. Il existe d'autres approches du raffinement temporel, que nous allons présenter brièvement, pour pouvoir les comparer à notre calcul.

TLA La Logique Temporelle des Actions (TLA [27, 2]) propose la modélisation suivante : tant que l'état du système abstrait ne change pas (le système « bégaie[3] »), tout comportement du système concret constitue un raffinement temporel. Ainsi, en TLA, raffiner un système, c'est ajouter du bégaiement, et pour retrouver le système abstrait, il suffit de le supprimer.

Signal Une autre approche comparable a été adoptée dans le langage Signal [29, 39]. Là où TLA maintient le système abstrait dans un état donné, pour permettre au système concret d'évoluer plus vite, Signal utilise des valeurs spéciales d'absence pour signifier que « rien de nouveau ne se passe » dans le système abstrait.

Lustre La méthode de raffinement du temps que nous avons proposée pour Lustre s'apparente donc davantage à TLA :

- Lorsque le système concret fonctionne plus rapidement que sa spécification, on utilise l'opérateur `current` pour transférer des données abstraites dans le domaine concret : on ajoute du bégaiement.
- Pour transformer les sorties concrètes en sorties abstraites, c'est l'opérateur `when` qui est utilisé. Et, en effet, cet opérateur supprime le bégaiement.

Ainsi, dans son esprit, notre raffinement suit le modèle de TLA. Néanmoins, pour des raisons pratiques, nous ne souhaitons pas atteindre toute l'expressivité de TLA, car cela nous permettrait d'obtenir des raffinements temporels infinis, où le système abstrait resterait bloqué pour toujours. Nous allons discuter ce point plus en détail dans la section suivante.

8.3.4 Le problème de l'équité

L'horloge de raffinement ck', définie au point 4 du résumé (section 8.3.1), a la sémantique suivante : *lorsque ck' est vraie, les systèmes abstrait et concret avancent dans le temps, mais lorsque ck' est fausse,*

[3] *stuttering* en anglais

seul le système concret évolue. Comme nous n'avons pas imposé de contrainte particulière sur les valeurs de la fonction f qui définit ck', il est légitime de soulever la question de ce qui adviendrait *si ck' devenait fausse à un point et le restait pour toujours* ?

Un tel cas correspond à la situation où le système abstrait avance jusqu'à un certain point : si l'horloge ck' a n valeurs vraies, alors le système abstrait passe par ses n premiers états. Ensuite, le $n+1^{\text{ème}}$ instant est raffiné à l'infini par le système concret (ck' devient fausse pour toujours). On peut formuler plusieurs commentaires sur cette situation :

La correction théorique L'obligation de preuve de correction (numéro 6 dans le résumé) assure que toute sortie concrète y' admet une image abstraite, qui est $\tau(y')$. Comme τ est une fonction Lustre, soit elle préserve les horloges, soit elle effectue un échantillonnage. Par conséquent, le flot y' a au moins autant d'éléments que $\tau(y')$.

Dans le cas où ck' devient fausse pour toujours, le raisonnement précédent reste valable : l'obligation de preuve (6) assure que la sortie du système concret sera correcte et en particulier, qu'elle aura la longueur requise pas la spécification abstraite.

De ce point de vue-là, le cas où ck' devient fausse pour toujours correspond donc à une situation où le flot de sortie dépend uniquement d'un nombre fini de valeurs du flot d'entrée. Ainsi, le fait d'arrêter de lire les entrées, dès que la dernière valeur pertinente est connue, constitue un raffinement correct de la spécification.

La preuve de la correction Cependant, une transcription de l'obligation de preuve 6 en Lustre ne peut pas exprimer la globalité de cette obligation. En particulier, dans le cas où ck' devient fausse pour toujours, l'expression de l'obligation 6 en Lustre ne peut pas comparer la totalité des sorties abstraite et concrète, car elles ne sont pas *synchrones* : lorsque ck' devient fausse, le système concret produit une infinité de valeurs avant que le système abstrait en produise une nouvelle.

Par conséquent, l'obligation de preuve écrite en Lustre englobe seulement la partie de sûreté[4] de l'obligation 6, qui est la comparaison des flots de sortie abstrait et concret jusqu'au point où ck' devient fausse pour toujours.

Une condition pour l'équité Pour éviter la situation sus-mentionnée, nous pouvons simplement ajouter une condition suffisante à notre système de raffinement, condition qui empêche tout raffinement temporel infini :

> *Toute valeur $fausse$ de l'horloge ck' peut être immédiatement suivie[5] par au plus un nombre fini de valeurs $fausses$.*

Cette condition, épaulée par le préservation de la longueur des flots que nous avons évoquée dans le paragraphe sur la correction théorique, assure que chaque instant abstrait est raffiné par au plus un nombre fini d'instants concrets. Cependant, nous n'avons pas de moyen pour vérifier en pratique si la condition ci-dessus est vraie pour un programme donné, car les observateurs synchrones ne peuvent pas exprimer les propriétés de vivacité. C'est pourquoi nous allons énoncer une condition un peu plus restrictive :

Le raffinement temporel pour le temps réel On peut dériver une propriété de sûreté de la condition ci-dessus :

> *Étant donné un entier naturel N, une valeur $fausse$ de ck' peut être immédiatement suivie par au plus $N-1$ valeurs $fausses$.*

[4] au sens de « propriété de sûreté »

[5] « immédiatement suivie » signifie « suivie, sans rencontrer de valeur *vraie* »

Connaissant N, il est possible d'écrire un observateur synchrone qui vérifie si cette dernière condition est vraie. On dira que cette condition établit un *raffinement pour le temps réel*, car elle décrit une situation où chaque instant abstrait est subdivisé en au plus $N+1$ instants concrets, si bien que le temps d'exécution maximum est connu.

Le raffinement périodique, tel que celui de l'exemple du nœud `Plus2` à la page 104 est un cas particulier du raffinement pour le temps réel.

Nous terminons ici notre présentation théorique du raffinement pour Lustre. Pour illustrer les possibilités du cadre formel que nous avons établi, nous allons proposer deux exemples détaillés de développement par raffinements au chapitre suivant.

9 – Exemples de raffinement

Pour illustrer le raffinement proposé, nous détaillons dans ce chapitre deux exemples de développement par raffinements. Le premier exemple montre la technique et la mécanisation du raffinement dans le cas d'un seul nœud ; le second exemple fait intervenir plusieurs nœuds.

9.1 Exemple du compteur modulo n

Pour commencer, nous allons illustrer le raffinement sur l'exemple du compteur modulo n.

9.1.1 Spécifications

Le but de cet exemple est de dériver par raffinements un nœud Lustre qui recevrait deux entrées, le flot booléen R et le flot entier n, et qui fournirait en sortie un flot d'entiers c. La sortie doit démarrer à 0, puis augmenter de 1 à chaque instant, jusqu'à atteindre la valeur $n-1$. À ce moment-là, le décompte recommence à 0. L'entrée R peut réinitialiser le compteur à tout moment. Enfin, la valeur de l'entrée n est constante pendant la phase de fonctionnement normal : n peut changer de valeur seulement à la *R*éinitialisation. La figure 9.1 présente un exemple d'exécution.

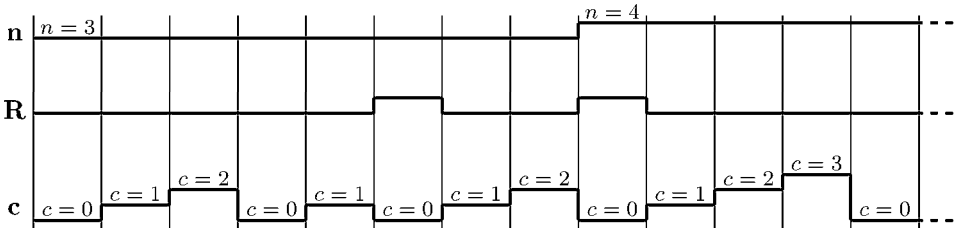


FIG. 9.1 – Chronogramme d'une exécution possible

Ces spécifications peuvent être exprimées formellement sous forme de deux propriétés P et Q qui portent sur les suites R, n et c :

$$\begin{aligned} P &\equiv \forall i > 0.(n_i > 0 \wedge \neg R_i \Rightarrow (n_i = n_{i-1})) \\ Q &\equiv \forall i > 0.[0 \leq c_i < n_i] \wedge [R_i \Rightarrow c_i = 0] \wedge [\neg R_i \Rightarrow (c_i = c_{i-1} + 1) \vee (c_i = 0 \wedge c_{i-1} = n_i - 1)] \end{aligned}$$

La propriété P est la précondition du compteur, Q est sa postcondition. On peut vérifier que la précondition ne concerne que les entrées, tandis que la postcondition lie les entrées et les sorties. En Lustre, nous allons décrire ces deux propriétés par les observateurs synchrones de la figure 9.2.

```
node counter_pre(R:bool ; n:int)
returns (b:bool);
let
 b = true →
      (n > 0) and
      ((not R) => n=(pre n));
tel;
```

```
node counter_post(R:bool ; n:int ; c:int)
returns (b:bool);
let
 b = true →
      (0 <= c) and (c < n) and
      (R => (c = 0)) and
      ((not R) =>
          (c = pre c + 1) or
          ((c = 0) and (pre c = n-1))) ;
tel;
```

FIG. 9.2 – Précondition et postcondition du compteur modulo n

```
node counter(R:bool ; n : int)
returns (c:int);
let
  assert counter_post(R,n,c) ;
  assert (c=0) → true ;
tel;
```

FIG. 9.3 – Spécification du compteur modulo n

Enfin, la spécification du compteur modulo n est présentée sur la figure 9.3. Le nœud `counter` est un nœud Lustre élargi, au sens de la section 8.2.4 : c'est une spécification non-déterministe sous forme de relation. La relation décrite est l'ensemble de tous les flots (R, n, c) qui vérifient les deux conditions énoncées dans le corps du nœud :

1. La première clause `assert` reprend la postcondition du nœud : ainsi, `counter` sera automatiquement correct (il vérifiera sa postcondition).
2. Nous avons ajouté une deuxième clause `assert` pour enrichir le comportement du nœud par rapport à sa postcondition : nous avons imposé l'initialisation de `c` à zéro. Ce choix est cohérent avec le comportement du compteur, tel que nous l'avons décrit informellement.

   Il est légitime de se demander ce qui adviendrait, si on ajoutait une clause `assert` en contradiction avec les autres clauses (par exemple `assert R => (c = 1) ;`). En fait, le nœud `counter` resterait *correct* par rapport à sa postcondition, mais deviendrait *non-réactif*, si bien qu'il serait impossible de le raffiner en une implémentation.

9.1.2 Premier raffinement

Il existe bien sûr une implémentation très simple du compteur modulo n, que nous présentons sur la figure 9.4. Le nœud `counter2` ne spécifie pas ses propres pré/postconditions : on considère qu'il reprend celles du nœud qu'il raffine, `counter`. De plus, `counter2` a exactement le même profil que son ancêtre et ne fait pas de changement de l'échelle du temps. Ainsi, les fonctions de changement de variables σ et τ sont de véritables identités.

Les vérifications statiques sont donc triviales. Quant aux obligations de preuve du raffinement, on peut les exprimer sous forme d'un observateur synchrone, `counter2_rpo`[1]. Ce dernier, donné sur la figure 9.5,

[1] `rpo` pour « refinement proof obligations »

peut être soumis à Gloups, puis prouvé en PVS. On peut donc effectivement prouver que `counter2` est un raffinement de `counter`.

```
node counter2(R:bool ; n : int)
returns (c:int);
refines counter;
let
  c = 0 -> if R then 0 else
            (if (pre c)=(n-1) then 0 else (pre c)+1) ;
tel;
```

FIG. 9.4 – Premier raffinement du compteur modulo n

```
node counter2_rpo(R1,R2:bool; n1,n2:int; c1,c2:int)
returns (ok:bool)
var ref,rea,cor:bool;
let
  ok = ref and rea and cor ;
  ref = (true -> (n1>0 and (not R1 => n1=(pre n1))))                    P
        and (R1=R2 and n1=n2)                                           σ
        and (c2 = (0 -> if R2 then 0 else                               S'
                        (if (pre c2=n2-1) then 0 else (pre c2)+1)))
        and (c1=c2)                                                     τ
        =>
        ((c1=0) -> true) and                                            S
        (true -> ((0 <= c1) and (c1<n1) and (R1 => c1=0)
                 and ((not R1) => (c1=pre c1+1)
                                  or ((pre c1=n1-1) and c1=0)))) ;

   rea = true ;
   cor = true ;
tel;
```

FIG. 9.5 – Obligations de preuve du premier raffinement

Le nœud `counter2_rpo` est produit automatiquement à partir de `counter`, `counter2`, `counter_pre` et `counter_post` par l'outil Flush[2]. Nous avons implémenté cet outil pour mécaniser les transformations de programmes Lustre nécessaires à la création de des obligations de preuve : en fait, ce sont de simples réécritures.

- On peut remarquer que `counter2_rpo` a une seule sortie, le booléen `ok`. Ce booléen est vrai si et seulement si les trois conditions de raffinement (`ref` comme « raffinement », `rea` comme « réactivité » et `cor` comme « correction locale ») sont vraies.
- Le nœud possède six entrées : trois pour le niveau abstrait, indexées par `1`, et trois pour le niveau concret, indexées par `2`. Il n'y a pas de contrainte sur leurs valeurs (pas de clause `assert` sur les

[2]Formal LUStre Helper

entrées), si bien que la propriété `ok` est en fait universellement quantifiée sur `R1`, `R2`, `n1`, `n2`, `c1` et `c2`.

- Flush a décidé de rendre l'obligation de réactivité `rea` triviale. C'est correct, car `counter2` a la même précondition que `counter`, si bien que l'obligation

$$[\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}]\ \forall x \in T.\ P(x) \Rightarrow P(x)$$

est toujours vraie. En plus, il est possible de détecter ce cas de figure statiquement, grâce à l'absence d'une précondition propre à `counter2`.

- De même, l'obligation de correction locale

$$[\mathrm{COR}_{\mathrm{S}'}]\ \forall (x', y') \in U \times U'.\ P'(x') \wedge S'(x', y') \Rightarrow Q'(x', y')$$

est toujours vraie, car $Q' = true$. Flush peut détecter ce cas grâce à l'absence de postcondition propre à `counter2`.

- Ainsi, la seule obligation non triviale est celle du raffinement

$$[\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}]\ \forall (x, y') \in T \times U'.P(x) \wedge Ck\_P'(ck', \sigma(x)) \wedge S'((ck', \sigma(x)), y') \Rightarrow S(x, \tau(y'))$$

Cette obligation se simplifie, car les changements de variables sont des identités et il n'y a pas de raffinement du temps :

$$[\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}]\ \forall (x, x', y, y') \in T \times T' \times U \times U'.\ P(x) \wedge x = x' \wedge S'(x', y') \wedge y = y' \Rightarrow S(x, y)$$

C'est cette dernière forme qui figure dans `counter2_rpo`.

9.1.3 Deuxième raffinement

Le nœud `counter2` est un raffinement correct de `counter` et en plus, c'est un nœud implémentable. Nous pourrions donc nous arrêter ici, car nous avons atteint le but fixé au départ. Néanmoins, nous allons proposer un pas de raffinement supplémentaire, pour illustrer notamment le raffinement temporel.

Puisque le langage Lustre peut être utilisé pour décrire des circuits électroniques, il serait possible de réaliser le circuit correspondant à `counter2`. Cependant, ce nœud ne s'y prête pas bien :

- Les composants matériels possèdent généralement un dispositif particulier pour la réinitialisation, qui est prioritaire sur les autres signaux. Toutefois, l'équation de `c` dans `counter2` place l'opérateur $\rightarrow$, c'est-à-dire le « début du temps », à une place prioritaire par rapport à `R`. Une implémentation directe de `counter2` sous forme de circuit mènerait donc à la création d'une logique supplémentaire (et inutile) qui contrôlerait la priorité de la réinitialisation. Nous aimerions éviter ce surcoût matériel, en rendant `R` prioritaire aux les autres signaux.
- Un autre inconvénient de `counter2` est la multiplicité des calculs arithmétiques. En effet, à chaque cycle d'horloge, le programme calcule `n-1`, compare cette valeur avec `c` et éventuellement incrémente `c`. Ces trois opérations arithmétiques en un cycle nécessiteraient trois UAL[3], ce qui serait trop coûteux en composants. Il serait peut-être possible d'utiliser moins d'UAL, si le cycle était subdivisé en plusieurs phases : ainsi, les opérations arithmétiques s'exécuteraient les unes après les autres sur la même UAL.

C'est pourquoi nous proposons un deuxième raffinement. Le nœud correspondant est présenté sur la figure 9.6 : il est censé marcher deux fois plus vite que `counter2`.

[3] Unité arithmético-logique

```
node counter3(R:bool ; n:int)
returns (c:int);
refines Counter2 by factor 2;
var p:int; cl:bool;
let
  cl = (true → not pre cl);
  c = if R then 0 else
       0 → (if cl then pre c
                  else (if p=n then 0 else p));
  p = if R then 0 else
       (0 → if cl then (pre c)+1 else pre p);
tel;
```

FIG. 9.6 – Deuxième raffinement du compteur modulo n

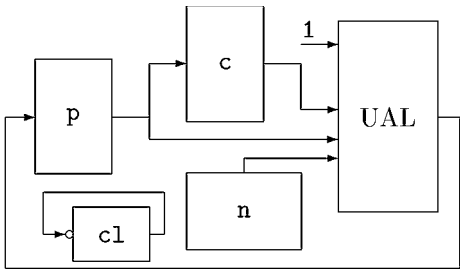


FIG. 9.7 – Circuit correspondant à `counter3`

Le nœud `counter3` introduit deux nouvelles variables, le booléen `cl` et l'entier `p`.

- Le booléen `cl` est une horloge, dont la période est deux fois plus longue que la période de l'horloge de base. C'est en fait l'horloge du raffinement, celle que nous avons appelé f dans le résumé du raffinement dans la section 8.3.1. L'horloge `cl` a pour but de marquer deux phases différentes :
- Dans la première phase, lorsque `cl` est vraie, la valeur précédente de `c` est incrémentée de 1 et la nouvelle valeur est stockée dans `p`. La valeur de `c` reste donc inchangée.
- Dans la seconde phase, lorsque `cl` passe à faux, la valeur de `p` est comparée à `n` et `c` est mise à jour selon le résultat de cette comparaison.

De cette manière, seule une UAL est nécessaire, car il y a une seule opération arithmétique dans chaque phase. Et, comme `counter3` fonctionne deux fois plus vite que `counter2`, le résultat est présent dans `c` à temps (à la fin de la seconde phase). La figure 9.7 schématise le circuit correspondant à `counter3` : nous avons dessiné seulement les flux de données, en omettant la partie contrôle.

Enfin, remarquons que `R` est devenu prioritaire sur $\rightarrow$.

Avant d'énoncer les obligations de preuve, récapitulons la situation :

- Puisque `counter2` et `counter3` ont les mêmes entrées, la projection de l'espace abstrait T sur l'espace concret U est une identité. Mais comme le système concret `counter3` fonctionne deux fois plus rapidement que le système abstrait $(ck = true \rightarrow (not(pre(ck))))$, le changement de variables σ est la fonction `current` qui associe à chaque flot d'entrée abstrait $e = (e_1, e_2, e_3, \ldots)$ le flot concret $(e_1, e_1, e_2, e_2, e_3, e_3, \ldots)$.

- Au niveau des flots de sortie, la situation est différente. N'oublions pas que les variables locales sont apparentées aux sorties pour ce qui concerne le raffinement. Ainsi, les variables concrètes (c,p,cl) seront projetées sur la variable abstraite c : en fait, la projection effacera simplement p et cl. De plus, comme il faut passer d'un système concret rapide vers un système abstrait plus lent, la fonction τ sera en fait une opération when qui associera au flot concret $(c_0, p_0, cl_0), (c_1, p_1, cl_1), \ldots$ le flot abstrait $c_0, c_2, \ldots$.

Comme précédemment, Flush nous permet d'exprimer les obligations de raffinement comme un nœud, donné sur la figure 9.8.

```
node counter3_rpo(R2, R3:bool ;  n2, n3:int ; c2, c3:int ; p3:int ; cl3:bool)
returns (ok:bool);
var ref,rea,cor,rc:bool;
let
  ok = ref and rea and cor ;
  ck = (true → not pre ck) ;
  assert (not ck) => (R2 = pre R2) ;                          le système abstrait
  assert (not ck) => (n2 = pre n2) ;                              est deux fois
  assert (not ck) => (c2 = pre c2) ;                                 plus lent

  cor = (true → ((n2>0) and ((not R2) → n2=(pre n2))))                          P
        and (R2=R3 and n2=n3)                                                   σ
        and (cl3=(true → (not pre cl3)))                                        S'
        and (p3=(if R3 then 0 else
                  (0 → if cl3 then pre c3+1 else pre p3)))
        and (c3 = (if R3 then 0 else 0 →
                    (if cl3 then pre c3 else (if p3=n3 then 0 else p3))))
        and ((not ck) => (c2=c3))                                               τ
        =>
        (c2=(0 → if R2 then 0 else                                              S
                  (if (pre c2=n2-1) then 0 else (pre c2)+1))) ;
  rea = true ;
  cor = true ;
tel;
```

FIG. 9.8 – Obligations de preuve du deuxième raffinement

Le seul point délicat de counter3_rpo est le traitement du changement de l'échelle temporelle. Même si nous avons présenté σ et τ comme des opérateurs current et when, il n'y a pas de current et when dans counter3_rpo. En fait, nous avons choisi de « remettre à plat » les différents domaines temporels :

- Le nœud counter3_rpo définit le flot booléen ck, qui est l'horloge du raffinement. Cette horloge est connue statiquement, parce que counter3 déclare raffiner l'échelle temporelle de counter2 par un facteur 2 (cf. « by factor 2 »). C'est pourquoi ck est vrai une fois sur deux.
- L'idée véhiculée par ck est que les variables concrètes (celles qui sont indexées par 3) peuvent évoluer à chaque instant, tandis que les variables abstraites (indexées par 2) ne peuvent changer qu'à la fin d'une période de ck. Ce comportement est assuré par les clausesassert au début du nœud.
- Du coup, σ devient simplement R2=R3 and n2=n3, car R2 et n2 évoluent à la vitesse donnée par ck, tandis que n3 et R3 suivent l'horloge de base.

– L'opération τ est traduite par `(not ck)⇒(c2=c3)` : ainsi, la seule contrainte qui pèse sur les sorties concrètes est qu'elles doivent être égales aux sorties abstraites en fin de période de `ck`.

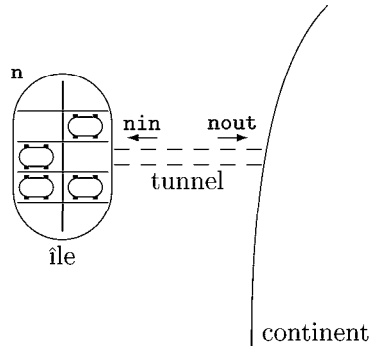
Une fois de plus, on peut prouver en utilisant Gloups que `ok` est vraie, si bien que `counter3` raffine `counter2` et, par transitivité, raffine aussi `counter`.

9.2 Exemple de l'île d'Abrial

Le raffinement que nous avons proposé au chapitre précédent est en partie inspiré par la méthode B de J.-R. Abrial. Certes, les deux approches diffèrent l'une de l'autre, car la méthode B se veut très générale, tandis que notre raffinement est d'envergure plus modeste, ce qui le rend plus accessible aux non-experts. Néanmoins, il nous a paru intéressant de comparer les deux méthodes sur un exemple qui a déjà été traité en B et dont nous proposons ici une version pour Lustre.

Cet exemple, en tant que mise en œuvre d'une théorie du raffinement et d'un procédé pratique pour prouver les obligations de correction (avec Gloups), a été présenté à la conférence FMICS'05 [34].

9.2.1 L'île



L'exemple que nous allons développer est celui d'une île [4] reliée avec le continent par un tunnel qui permet la circulation de voitures. Un parking est aménagé sur l'île, mais sa capacité d'accueil est limitée.
On suppose qu'initialement, il y a `ninit` voitures sur l'île. En une unité de temps, `nin` voitures arrivent sur l'île et `nout` voitures la quittent. Notre but est d'assurer que le nombre de voitures sur l'île (noté `n`) ne dépassera jamais la capacité d'accueil `nmax`.

Une première spécification La figure 9.9 présente la traduction en Lustre de la spécification précédente.

```
const ninit : int;
const nmax  : int;

node island(nin, nout : int)
returns(n : int);
let
  n = (ninit → pre n) + nin - nout;
tel;

node island_post(nin, nout, n : int)
returns(prop : bool);
let
  prop = (n <= nmax);
tel;
```

FIG. 9.9 – Spécification initiale de l'île

À part les déclarations de constantes, la spécification de la figure 9.9 est composée de deux nœuds :
- le modèle de l'île (`island`) qui définit exactement le nombre de voitures sur l'île (`n`) en fonction des arrivées (`nin`), des départs (`nout`) et de la situation initiale (`ninit`),
- et la propriété voulue (`island_post`) qui est en fait la postcondition du nœud précédent.

Une spécification améliorée Lorsque nous essayons de prouver la postcondition de la figure 9.9, la preuve échoue. À ce stade du développement, il serait possible que l'échec vienne tout simplement de notre modélisation : nous n'avons peut-être pas inclus toutes les propriétés importantes dans notre spécification. C'est pourquoi nous ajoutons une précondition à notre modélisation, qui stipule quelques contraintes supplémentaires, comme par exemple le fait que le nombre de voitures initial ne dépasse pas la capacité d'accueil de l'île.

```
node island_pre(nin, nout : int)
returns(prop : bool);
let
  prop = (0 <= ninit) and (ninit <= nmax) and
         (0 <= nin)   and (0 <= nout);
tel
```

FIG. 9.10 – La précondition

Tentative de preuve À présent, nous pouvons à nouveau essayer de prouver la correction du nœud `island` par rapport à sa postcondition, en tenant compte de la précondition. L'outil Flush génère un observateur synchrone pour cette preuve. La figure 9.11 présente une version lisible de la sortie de Flush : nous avons supprimé certaines variables inutiles en les remplaçant par leurs définitions.

```
node island_rpo(nin, nout, n : int)
returns(prop : bool);
let
  assert island_pre(nin, nout);
  prop = island_post(nin, nout,
                     island(nin, nout));
tel
```

FIG. 9.11 – Obligation de preuve pour la correction locale

Mais, encore une fois, la preuve échoue.

9.2.2 Un contrôleur non-déterministe

L'échec de la preuve précédente nous apporte néanmoins un précieux renseignement : notre système ne peut pas fonctionner correctement sans un contrôleur. Ainsi, nous pouvons considérer que notre modèle n'est pas biaisé, si bien que désormais, nous pouvons nous concentrer sur la conception d'un contrôleur.

Le contrôleur en question ne doit pas introduire de biais à notre modèle. En fait, Lustre permet d'assurer cette propriété grâce à con caractère fonctionnel : un nœud est une fonction et, comme telle,

il n'a pas d'effet de bord. Par conséquent, si nous ne modifions ni `island`, ni `island_pre`, le modèle de l'île restera inchangé.

Cette manière de procéder, inspirée de la théorie du contrôle, est assez différente de la méthode choisie par Abrial. Ce dernier élabore le modèle de l'île et du contrôleur en même temps, dans un cadre où la propriété de non-dépassement de la capacité d'accueil est vraie dès le départ. C'est seulement à la fin du processus de raffinement que l'île et son contrôleur sont séparés. Il est alors bien plus difficile de s'assurer de l'absence de biais dans la modélisation.

Notre premier contrôleur n'est pas déterministe : la postcondition d'`island` y est assurée par « un tout de passe-passe ». Ce contrôleur, présenté sur la figure 9.12, se contente de calculer la valeur de vérité des propriétés qui nous intéressent.

```
node controller(nin, nout: int)
returns(prop: bool);
let
  prop = island_pre(nin, nout) and
         island_post(nin, nout,
                     island(nin, nout));
tel;
```

FIG. 9.12 – Le contrôleur non-déterministe

Ensuite, nous encapsulons `island` et son `controller` au sein d'un même nœud, `controlled_island` (cf. figure 9.13), qui impose que la valeur rendue par le contrôleur soit toujours vraie.

```
node controlled_island()
returns(n : int);
var nin,  nout: int;
let
  assert controller(nin,  nout);
  n = island(nin, nout);
tel;

node controlled_island_post(n: int)
returns(prop : bool));
let
  prop = (n <= nmax);
tel;
```

FIG. 9.13 – L'île contrôlée

On peut remarquer que `controller` fait appel au nœud `island`. Cela ne signifie pas pour autant que contrôleur connaît comme par magie le nombre de voitures sur l'île. En fait, les deux instances d'`island` (celle de `controller` et celle de `controlled_island`) donnent le même résultat si et seulement si elles ont exactement les mêmes entrées. Ainsi, l'instance de l'île utilisée par le contrôleur aurait pu avoir un autre comportement que l'île elle-même.

Pour finir, Flush génère l'obligation de preuve `controlled_island_rpo`. Cette dernière peut être prouvée très facilement, car de simples réécritures par les égaux la rendent trivialement vraie.

9.2.3 Raffinement du temps

À ce point du développement, nous avons un contrôleur correct, mais non-déterministe. L'une des caractéristiques qui l'empêchent de devenir déterministe (et donc utilisable en pratique) est la manière abstraite dont il reçoit les informations sur son environnement. En effet, nous allons devoir introduire des capteurs pour mesurer les flux de voitures entrant et sortant.

Le problème que nous devons surmonter à ce stade est la nature booléenne des capteurs physiques, qui envoient vrai ou faux selon la présence ou l'absence de voiture en face d'eux. Pour relier l'information en provenance des capteurs avec les entiers `nin` et `nout`, qui représentent le nombre de voitures entrant et sortant en une unité de temps, il va nous falloir subdiviser cette unité de temps : nous allons raffiner le temps.

Les capteurs On peut facilement créer des pilotes pour les capteurs en utilisant les opérateurs `when` et `current`. Le résultat se trouve sur la figure 9.14.

```
const nm: int;

node countmod() returns (clock: bool);
var n:  int;
let
  n = ((0 → pre n) mod nm) + 1;
  clock = (n = nm);
tel;

node carcount(car, cl: bool)
returns (ncar: int when cl)
var nc: int;
let
  nc = 0 → (if car then 1 else 0) +
           (if pre cl then 0 else pre nc) ;
  ncar = nc when cl ;
tel;
```

FIG. 9.14 – Capteur pour compter les voitures

Dans la section 9.2.1, nous avons postulé qu'un certain nombre de voitures entrait et sortait par unité de temps, sans toutefois spécifier quelle était cette unité. À présent, nous sommes en mesure de préciser cette information, en reliant le temps et la vitesse des voitures grâce à la constant `nm` qui exprime le nombre maximum de véhicules qui peuvent passer devant le capteur en une unité de temps.

Sachant cela, le nœud `countmod` peut facilement fournir l'horloge `clock` qui devient vraie lorsqu'une unité de temps s'est écoulée : il suffit de compter les instants modulo `nm`. Ensuite, le nœud `carcount` calcule le nombre de voitures qui passent en une unité de temps (donnée par l'entrée `cl`) devant le capteur représenté par l'entrée `car`. Par l'utilisation de l'opérateur `when`, la sortie de `carcount` évolue plus lentement que les entrées.

Raffinement des variables d'état La modélisation des capteurs que nous venons d'établir nous permet de relier les mesures physiques effectuées par les capteurs avec les variables abstraites définies précédemment. Ce changement de variables est montré sur la figure 9.15 : nous avons simplement encapsulé les pilotes des capteurs et l'horloge qui donne l'unité de temps.

```
node refvar(cin, cout: bool)
returns(cl: bool; nin, nout: int when cl)
let
  cl   = countmod();
  nin  = carcount(cin, cl);
  nout = carcount(cout, cl);
tel;
```

FIG. 9.15 – Changement de variables

L'île raffinée Après avoir intégré le changement de variables précédent au modèle de l'île de la figure 9.13, nous obtenons un nouveau modèle, présenté sur la figure 9.16

```
node controller1(cin, cout: bool)
returns(cl, prop: bool);
var nin, nout: int when cl ;
let
  cl,  nin,  nout = refvar(cin, cout);
  prop = current(controller(nin, nout));
tel;

node controlled_island1()
returns(n : int);
refines controlled_island;
var cin, cout: bool;
    nin, nout: int when cl;
    cl, clc  : bool;
    prop     : bool;
let
  cl,  nin,  nout = refvar(cin, cout);
  clc,  prop = controller1(cin, cout);
  assert (cl = clc);
  assert prop;
  n = island(nin, nout);
tel;
```

FIG. 9.16 – L'île raffinée

On peut formuler plusieurs remarques sur le modèle de la figure 9.16 :

- Nous avons appliqué le même changement de variables *de manière indépendante* dans le contrôleur et dans le modèle de l'île. De cette façon, nous continuons à bien différencier les calculs effectifs du contrôleur et la modélisation de l'environnement.
- L'une des conséquences de cette séparation est l'existence simultanée de deux horloges : `cl` est calculée dans le modèle et représente le temps physique, tandis que `clc` est calculée dans le contrôleur et représente l'horloge de l'ordinateur.
  Dans le cadre de notre système, les deux horloges sont calculées de la même manière à partir des mêmes paramètres, si bien qu'elles sont égales. Ainsi, l'assertion `assert(cl = clc)` ne sert à rien,

mais nous avons décidé de la garder, pour insister sur l'importance de cette égalité. En effet, la correction d'une implémentation concrète reposera sur la capacité d'une machine donnée à mesurer correctement le temps physique : il faudra s'assurer que l'égalité des horloges est toujours vérifiée, par exemple en choisissant une implémentation tolérante aux pannes.

Enfin, nous avons prouvé que le raffinement de la figure 9.16 est correct : le changement de variables utilisé préserve trivialement le comportement du système.

9.2.4 Le contrôleur déterministe

Dans cette dernière partie, nous allons pouvoir tirer profit du changement de variables précédent, pour construire un contrôleur déterministe. Nous allons procéder comme suit :

```
node light(cin, red: bool)
returns(prop: bool);
let
  prop = (red => not cin);
tel;

node lightcontrol(nin, nout: int)
returns(red: bool);
let
  red = (ninit → pre island(nin, nout))
        + nm - 0 > nmax;
tel;

node controller2(cin, cout: bool)
returns(cl, prop: bool);
refines controller1;
var nin, nout: int when cl;
    red: bool;
let
  cl,  nin,  nout = refvar(cin, cout);
  red = current(true, cl,
                lightcontrol(nin, nout));
  assert light(cin, red);
  prop = true;
tel;
```

FIG. 9.17 – Le contrôleur déterministe

- Nous allons « installer » un feu tricolore à l'entrée du tunnel et interdire aux voitures d'entrer dans le tunnel (et donc sur l'île), lorsque le feu est rouge. Le nœud `light` modélise le comportement voulu.
- La gestion du feu rouge incombe au contrôleur `controller2`. En fait, le fonctionnement de celui-ci représente la partie créative de la modélisation.
  Le problème réside dans le fait que le contrôleur ne peut pas connaître le nombre exact de voitures qui entreraient sur l'île, si le feu était vert. Ainsi, il faut adopter une politique pessimiste (`lightcontrol`) qui prévoit le maximum d'entrées (`nm`) et le minimum de sorties (`0`) : si avec une

telle hypothèse, le nombre de voitures sur l'île dépasse la capacité d'accueil, alors il faut allumer le feu rouge et ne laisser entrer aucune voiture.

Le résultat est affiché sur la figure 9.17.

Le code du `controller2` illustre également quelques points subtiles du calcul d'horloges en Lustre : comme l'appel du nœud `lightcontrol` est sur l'horloge `cl`, tandis que le nœud `light` est sur l'horloge de base, nous devons les faire communiquer à travers d'un opérateur `current`. On peut remarquer que cet opérateur est initialisé à vrai, si bien que le feu commence par être rouge.

La sortie `prop` de `controller2` est devenue constante, égale à vrai. En effet, les règles du raffinement nous interdisent d'effacer cette sortie devenue inutile : lui assigner la valeur `true` est une manière de supprimer son influence dans `controlled_island`, car `assert prop` devient automatiquement vraie.

Il reste à prouver que `controller2` raffine son ancêtre `controller1`. On peut remarquer que cette preuve de raffinement est la seule qui présente quelques difficultés, car elle fait intervenir l'arithmétique linéaire.

Pour rendre la preuve plus facile, nous définissons quelques propriétés utiles : d'un part, une contrainte sur la valeur de la constante `nm` et d'autre part, des « lemmes »supplémentaires sous forme de postcondition. Cette postcondition stipule entre autres que la somme de `nm` booléens ne peut pas dépasser `nm` : nous prouvons ces propriétés au titre de la correction locale, puis nous les utilisons dans la preuve du raffinement. La figure 9.18 présente tous les ajouts mentionnés.

```
node controller1_pre(cin, cout: bool)
returns(prop: bool)
let
  prop = (0 < nm) and (nm <= nmax);
tel;

node controller2_post(cin, cout: bool)
returns(prop: bool)
var cl: bool;
    nin, nout: int when cl;
let
  cl,  nin,  nout = refvar(cin, cout);
  prop = current(true, cl,
                 (0 <= nin) and (nin <= nm) and
                 (0 <= nout) and (nout <= nm));
tel;
```

FIG. 9.18 – Précondition et postcondition du dernier raffinement

Nous pouvons prouver que `controller2_post` est effectivement déductible du contrôleur et que `controller2` raffine bien `controller1`. Ainsi, nous terminons l'exemple en ayant effectivement obtenu un contrôleur déterministe de l'accès sur l'île.

10 – Conclusion et perspectives

Nous concluons sur l'ensemble de nos travaux et proposons des perspectives pour de nouvelles recherches.

Les recherches que nous avons menées dans le cadre de notre thèse étaient centrées autour du développement formel de systèmes réactifs. Comme nous l'avons expliqué dans la partie introductive, les systèmes réactifs sont généralement peu visibles, mais vitaux, car ils s'acquittent des tâches critiques, qui ont de fortes contraintes de temps et d'espace. Ainsi, les systèmes réactifs constituent un terrain naturel d'application des méthodes de développement formel, car la correction de ces systèmes est primordiale.

Le langage Lustre et sa variante industrielle Scade sont des langages synchrones utilisés dans de nombreux développements de systèmes réactifs, aussi bien logiciels que matériels. Ces langages se prêtent bien à l'application des méthodes formelles grâce à leur sémantique rigoureusement définie. Par ailleurs, ils possèdent un certain nombre de caractéristiques uniques, telles que le synchronisme ou l'aspect flot-de-données, qui rendent leur étude théorique intéressante.

Notre contribution dans le domaine que nous venons de délimiter est double. Dans un premier temps, nous avons repris, corrigé et instrumenté une méthode de preuve inductive pour Lustre, initialement développée par Cécile Dumas. De cette manière, nous avons traité le côté vérification formelle du développement des systèmes avec Lustre. La seconde partie de nos recherches était consacrée à l'établissement d'une méthode de raffinement : ainsi, nous avons créé un mécanisme de génie logiciel pour le développement effectif des programmes.

Nos deux contributions sont donc complémentaires l'une de l'autre : tandis que la méthode de preuve inductive étaie le raffinement, qui, sans elle, ne serait qu'un exercice académique, le raffinement offre une manière effective de développement formel de systèmes avec Lustre. Les deux forment donc un tout.

Le travail que nous avons décrit dans ce mémoire peut être prolongé de nombreuses manières. Les tâches les plus immédiates pourraient concerner l'instrumentation de nos méthodes :

- Si Gloups est un outil de bonne qualité, il reste néanmoins un certain nombre de points dont le traitement pratique pourrait être amélioré. C'est par exemple le cas des horloges et des opérateurs d'horloges `when` et `current`.
- L'outil Flush est encore au stade expérimental. À l'état actuel, cet outil se contente d'un prétraitement quasi automatique des programmes Lustre, dont le but est l'établissement des obligations de preuve du raffinement. Nos expérimentations avec le raffinement ont mis en évidence de nouvelles exigences, notamment dans l'interaction avec l'utilisateur, qui devrait pouvoir manipuler les systèmes avec davantage de flexibilité. À terme, Flush pourrait devenir un véritable gestionnaire de développement par raffinements : l'outil gérerait aussi bien l'aspect vertical (raffinement) qu'horizontal (modules) du développement.

À plus long terme, d'autres développements théoriques pourraient être abordés. Par exemple, nous avons structuré la vérification et le raffinement autour des propriétés de sûreté : il serait intéressant d'explorer également le domaine des propriétés de vivacité, car une méthode de preuve pour (certaines de) ces propriétés permettrait d'augmenter l'expressivité du raffinement.

Une autre piste pour les futures recherches est le traitement de la modularité. Actuellement, nous disposons de deux manières rudimentaires de traiter des programmes modulaires : soit de supprimer toute la structure en remettant les modules à plat, soit de définir à la main les abstractions qui permettent d'effectuer les preuves module par module. Or aucune de ces deux approches n'est applicable aux systèmes de grande taille, c'est pourquoi il serait intéressant, sinon indispensable, d'explorer le domaine des preuves modulaires.

Quatrième partie

Annexes

A – Résumé de Lustre

Syntaxe

Déclarations globales

importation de type	`type` <*nom*> ;
définition de type	`type` <*nom*> `=` <*définition*> ;
importation de constante	`const` <*nom*> `:` <*type*> ;
définition de constante	`const` <*nom*> `=` <*valeur*> ;
définition de fonction	`function` <*nom*> (<*liste de paramètres*>) `returns` (<*liste de paramètres*>) ;
où un <*paramètre*> est	<*nom*> `:` <*type*> ;
définition de nœud	`node` <*nom*> (<*liste de paramètres*>) `returns` (<*liste de paramètres*>) ; `var` <*liste de variables locales*> `let` <*liste d'équations et d'assertions*> `tel` ;

Nœuds

déclaration d'une variable	<*nom*> `:` <*type*> [`when` <*horloge*>] ;	
une équation	<*variable*> `=` <*expression*> ;	
une assertion	`assert` <*expression*> ;	
définition d'<*expression*>	· <*constante*>	
	· <*variable*>	
	· <*un*> <*expression*>	*opération unaire*
	· <*expression*> <*bi*> <*expression*>	*opération binaire*
	· <*nom*>(<*liste d'arguments*>)	*appel de fonction/nœud*
	· `if` <*expression*>	*conditionnelle*
	`then` <*expression*>	
	`else` <*expression*>	

expressions unaires <*un*>	· `pre`	*valeur précédente*
	· `current`	*« accélération » de flot lent*
	· `int`	*transformation réels→entiers*
	· `real`	*transformation entiers→réels*
	· `-`	*moins unaire*
expressions binaires <*bi*>	· *opérations arithmétiques*	`+ - * / div mod`
	· *comparaisons*	`< > <= >= = <>`
	· *opérations logiques*	`and or xor =>`
	· `when`	*échantillonnage*
	· `->`	*initialisation de flot*
	· `fby`	*« followed by »*

Typage des expressions

Nous allons présenter le typage des expressions Lustre sous forme de règles d'inférence. Pour avoir une présentation simple, nous allons omettre le cas d'appel de fonction (ou de nœud), qui exigerait des complications excessives dans les notations.

Le jugement de typage aura la forme générale suivante :

$$\langle T, \Omega, \Gamma \rangle \vdash \text{<}\textit{expression}\text{>} : \text{<}\textit{type scalaire}\text{>} :: \text{<}\textit{horloge}\text{>}$$

où T est l'environnement[1] de typage des variables (T : <*variable*> $\mapsto$ <*type scalaire*>), Γ l'environnement de typage des constantes[2] et Ω l'environnement des horloges (Ω : <*variable*> $\mapsto$ <*horloge*>). Par ailleurs, nous utiliserons les types de base `int`, `real` et `bool` et l'horloge de base `true`.

Cas de base

$$(\text{const})\frac{c \text{ est une constante}}{\langle T, \Omega, \Gamma \rangle \vdash c : \Gamma(c) :: \texttt{true}} \qquad (\text{var})\frac{v \text{ est une variable}}{\langle T, \Omega, \Gamma \rangle \vdash v : T(v) :: \Omega(v)}$$

Opérations arithmétiques et logiques

$$(\text{arith}_1)\frac{\langle T, \Omega, \Gamma \rangle \vdash e_1 : \texttt{int} :: \omega \quad \langle T, \Omega, \Gamma \rangle \vdash e_2 : \texttt{int} :: \omega}{\langle T, \Omega, \Gamma \rangle \vdash e_1 \ \langle ari \rangle \ e_2 : \texttt{int} :: \omega} \quad \langle ari \rangle \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{div}, \texttt{mod}\}$$

$$(\text{arith}_2)\frac{\langle T, \Omega, \Gamma \rangle \vdash e_1 : \texttt{real} :: \omega \quad \langle T, \Omega, \Gamma \rangle \vdash e_2 : \texttt{real} :: \omega}{\langle T, \Omega, \Gamma \rangle \vdash e_1 \ \langle ari \rangle \ e_2 : \texttt{real} :: \omega} \quad \langle ari \rangle \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}$$

$$(\text{logic})\frac{\langle T, \Omega, \Gamma \rangle \vdash e_1 : \texttt{bool} :: \omega \quad \langle T, \Omega, \Gamma \rangle \vdash e_2 : \texttt{bool} :: \omega}{\langle T, \Omega, \Gamma \rangle \vdash e_1 \ \langle log \rangle \ e_2 : \texttt{bool} :: \omega} \quad \langle log \rangle \in \{\texttt{and}, \texttt{or}, \texttt{=>}\}$$

[1] Ces environnements sont construits à partir des déclarations de constantes (dans le cas de Γ) et des en-têtes des nœuds (T et Ω).

[2] Une constante a toujours un seul type : par exemple `1` est un entier et `1.0` est le réel correspondant.

$$(\text{compare}_1)\frac{\langle T,\Omega,\Gamma\rangle \vdash e_1 : \texttt{int} :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash e_2 : \texttt{int} :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash e_1 \ \langle com\rangle \ e_2 : \texttt{bool} :: \omega} \ \langle com\rangle \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{<>}, \texttt{=}\}$$

$$(\text{compare}_2)\frac{\langle T,\Omega,\Gamma\rangle \vdash e_1 : \texttt{real} :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash e_2 : \texttt{real} :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash e_1 \ \langle com\rangle \ e_2 : \texttt{bool} :: \omega} \ \langle com\rangle \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{<>}, \texttt{=}\}$$

Changements de type

$$(\text{to\_int})\frac{\langle T,\Omega,\Gamma\rangle \vdash e : \texttt{real} :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash \texttt{int} \ e : \texttt{int} :: \omega} \qquad (\text{to\_real})\frac{\langle T,\Omega,\Gamma\rangle \vdash e : \texttt{int} :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash \texttt{real} \ e : \texttt{real} :: \omega}$$

Opérateurs temporels

$$(\text{flow}_1)\frac{\langle T,\Omega,\Gamma\rangle \vdash e : \tau :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash \texttt{pre} \ e : \tau :: \omega}$$

$$(\text{flow}_2)\frac{\langle T,\Omega,\Gamma\rangle \vdash e_1 : \tau :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash e_2 : \tau :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash e_1 \ \langle op_2\rangle \ e_2 : \tau :: \omega} \ \langle op_2\rangle \in \{\texttt{fby}, \texttt{->}\}$$

Opérateurs d'horloges

$$(\text{current})\frac{\langle T,\Omega,\Gamma\rangle \vdash e : \tau :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash \omega : \texttt{bool} :: \sigma}{\langle T,\Omega,\Gamma\rangle \vdash \texttt{current} \ e : \tau :: \sigma} \ \omega \neq \texttt{true}$$

$$(\text{when})\frac{\langle T,\Omega,\Gamma\rangle \vdash e_1 : \tau :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash e_2 : \texttt{bool} :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash e_1 \ \texttt{when} \ e_2 : \tau :: e_2}$$

Conditionnelle

$$(\text{if})\frac{\langle T,\Omega,\Gamma\rangle \vdash e_1 : \texttt{bool} :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash e_2 : \tau :: \omega \quad \langle T,\Omega,\Gamma\rangle \vdash e_3 : \tau :: \omega}{\langle T,\Omega,\Gamma\rangle \vdash \texttt{if} \ e_1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3 : \tau :: \omega}$$

Appel de fonction/nœud

Il est plus commode d'énoncer les contraintes de typage pour les appels d'une manière informelle :
- Le nombre et le type (scalaire) des paramètres d'entrée doivent être respectés.
- Le type de l'appel est alors le type de retour de la fonction/nœud.

- Pour les appels de fonction, toutes les entrées et toutes les sorties partagent une même horloge (pas nécessairement l'horloge de base).
- La situation est plus compliquée pour les appels de nœuds, car il peut y avoir des horloges différentes pour des paramètres différents, et surtout, certains paramètres peuvent être déclarés comme horloge d'un autre paramètre : il faut donc vérifier que ces dépendances croisées soient cohérentes entre elles et qu'en particulier, on puisse construire un ordre partiel entre les horloges.
  Par exemple, si on a défini un nœud au profil suivant,

```
node ex (ck:bool; a:int when ck; b:int)
returns (res:int when ck);
...
```

  alors à l'appel `ex(x,y,z)`, il faudra vérifier que `x` et `z` sont sur l'horloge de base et que `y` est sur `x`. Alors, on pourra conclure que `ex(x,y,z)` est également sur `x`.

Sémantique des expressions

1) Expressions combinatoires

Toutes les expressions combinatoires binaires obéissent au schéma suivant :

$$\begin{array}{rcrcl} x_0.\mathtt{x} & <op> & y_0.\mathtt{y} & = & (x_0 <op> y_0).(\mathtt{x} <op> \mathtt{y}) \\ \varepsilon & <op> & \varepsilon & = & \varepsilon \end{array}$$

$<op>$ est l'un des opérateurs suivants :

$+$	addition d'entiers ou de réels
$-$	soustraction d'entiers ou de réels
$*$	multiplication d'entiers ou de réels
div	division euclidienne
mod	reste de la division euclidienne
and	conjonction logique
or	disjonction logique
$\Rightarrow$	implication logique

Les expressions combinatoires unaires sont définies par

$$\begin{array}{rcrcl} <op> & x_0.\mathtt{x} & = & (<op> x_0).(<op> \mathtt{x}) \\ <op> & \varepsilon & = & \varepsilon \end{array}$$

où $<op>$ peut être

$-$	l'opposé d'un entier ou de réel
$real$	identité $id : \mathbb{N} \to \mathbb{R}$
int	arrondi ($\mathbb{R} \to \mathbb{N}$)

Enfin, la conditionnelle est définie par

$$\begin{array}{lcccccl} \mathtt{if} & t.\mathtt{c} & \mathtt{then} & x_0.\mathtt{x} & \mathtt{else} & y_0.\mathtt{y} & = & x_0.(\mathtt{if\ c\ then\ x\ else\ y}) \\ \mathtt{if} & f.\mathtt{c} & \mathtt{then} & x_0.\mathtt{x} & \mathtt{else} & y_0.\mathtt{y} & = & y_0.(\mathtt{if\ c\ then\ x\ else\ y}) \\ \mathtt{if} & \varepsilon & \mathtt{then} & \varepsilon & \mathtt{else} & \varepsilon & = & \varepsilon \end{array}$$

2) Expressions temporelles

définition auxiliaire			$\mathsf{pre}'(v, \varepsilon)$	=	ε
			$\mathsf{pre}'(v, x_0.\mathsf{x})$	=	$v.\mathsf{pre}'(x_0, \mathsf{x})$
valeur précédente			pre x	=	pre′(?,x)
initialisation	$x_0.\mathsf{x}$	$\rightarrow$	$y_0.\mathsf{y}$	=	$x_0.\mathsf{y}$
	ε	$\rightarrow$	ε	=	ε
valeur précédente	$x_0.\mathsf{x}$	fby	y	=	$\mathsf{pre}'(x_0, \mathsf{y})$
avec initialisation	ε	fby	y	=	ε
échantillonnage	$x_0.\mathsf{x}$	when	$t.\mathsf{c}$	=	$x_0.(\mathsf{x\ when\ c})$
	$x_0.\mathsf{x}$	when	$f.\mathsf{c}$	=	x when c
	ε	when	ε	=	ε
blocage			$\mathsf{current}(d, t.\mathsf{h}, y_0.\mathsf{y})$	=	$y_0.\mathsf{current}(y_0, \mathsf{h}, \mathsf{y})$
			$\mathsf{current}(d, f.\mathsf{h}, \mathsf{y})$	=	$d.\mathsf{current}(d, \mathsf{h}, \mathsf{y})$
			$\mathsf{current}(d, \varepsilon, \varepsilon)$	=	ε

B – Processus de preuve

B.1 Règles de preuve

B.1.1 Règles de déduction

Induction continue

$$(\text{cont})\frac{\Box P(\varepsilon) \qquad \forall y \in D^*.\ \Box P(y) \Rightarrow \Box P(f(y))}{\Box P(\mu_x.x = f(x))}$$

Induction continue avec dépliages en bloc

$$(\text{unfold}_n)\ \frac{\begin{array}{l}\Box P(\varepsilon)\\ \Box P(f(\varepsilon))\\ \quad\ldots\\ \Box P(f^{n-1}(\varepsilon))\\ \forall x \in D^*.\Box P(x) \wedge \Box P(f(x)) \wedge \ldots \wedge \Box P(f^{n-1}(x)) \Rightarrow \Box P(f^n(x))\end{array}}{\Box(P(\mu_x.\ x = f(x))}$$

Induction structurelle

$$(\text{ind})\frac{Q(\varepsilon) \qquad \forall z \in D^+.\ Q(tl\ z) \Rightarrow Q(z)}{\forall y \in D^*.\ Q(y)}$$

où tl est l'opération « queue de liste »

Avancement

$$(\text{cons}_1)\frac{}{\Box P(\varepsilon)} \qquad (\text{cons}_2)\frac{x \qquad \Box P(X)}{\Box P(x.X)} \qquad (\text{hd})\frac{\Box P(x.X)}{x} \qquad (\text{tl})\frac{\Box P(x.X)}{\Box P(X)}$$

B.1.2 Règles de simplification

Les règles de simplification sont en fait des règles de réécriture dérivées de la sémantique de Lustre. Les réécritures de base sont :

`int` $\varepsilon \to \varepsilon$
`real` $\varepsilon \to \varepsilon$
`-` $\varepsilon \to \varepsilon$
	(ε <*op*> e) $\to \varepsilon$
	(e <*op*> ε) $\to \varepsilon$
	où <*op*>$\in$`{+,-,*,/,div,mod,`
	`<,>,<=,>=,=,<>,=>,`
	`and,or,xor,when,->}`
		`if` ε `then` e `else` e $\to \varepsilon$
		`if` e `then` ε `else` e $\to \varepsilon$
		`if` e `then` e `else` $\varepsilon \to \varepsilon$
			`default(`e`,`ε`,`e`)` $\to \varepsilon$
			`default(`e`,`e`,`ε`)` $\to \varepsilon$
			ε `fby` e $\to \varepsilon$

où e est une expression quelconque			

La simplification consiste alors à appliquer les règles ci-dessus jusqu'à ce qu'il n'y ait plus de réécriture possible.

B.2 Algorithme de preuve

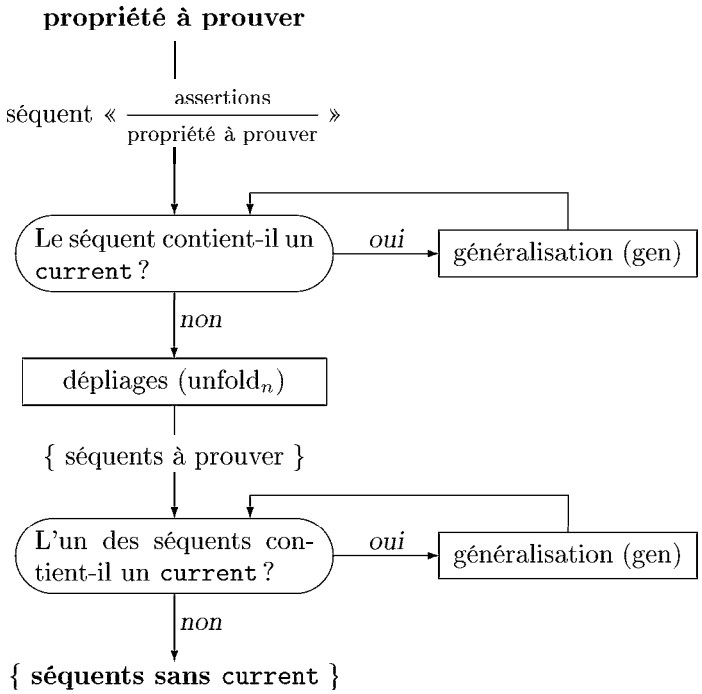


FIG. B.1 – La phase de dépliages–généralisations

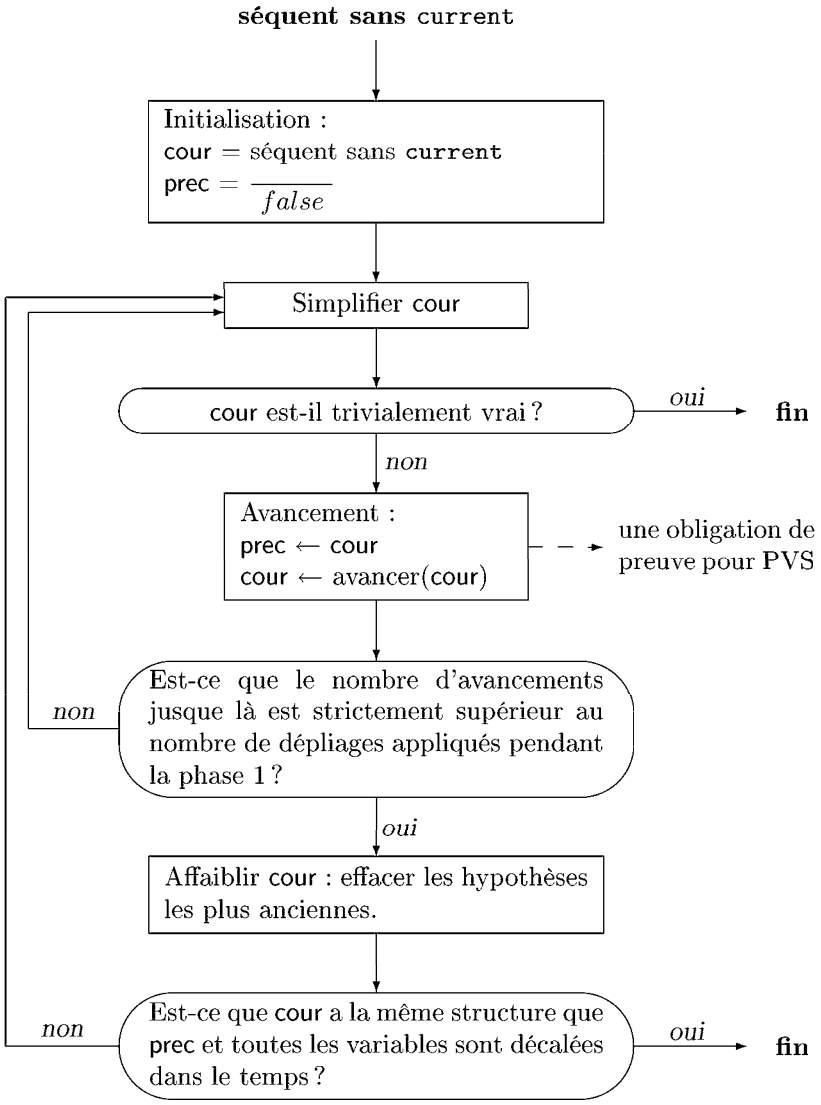


FIG. B.2 – La phase d'avancements–induction

C – Fichiers pour PVS

C.1 Obligations de preuve pour la suite de Fibonacci

Nous reproduisons ici les fichiers PVS que Gloups crée comme obligations de preuve dans le cas de l'exemple 6.1 : il s'agit de prouver que tous les éléments de la suite de Fibonacci sont strictement positifs. Le code source de l'exemple est :

```
node fibo(rien:int)
returns (f:int);
var g:int ;
let
  f = 1 -> pre (f+g) ;
  g = 0 -> pre f ;
  prove (f > 0) ;
tel;
```

Gloups produit six fichiers au total : le document maître, le document de typage et quatre obligations de preuve. Nous avons utilisé toutes les optimisations, si bien que les obligations de preuve sont « épurées » : il ne reste que les propriétés vraiment en rapport avec le but.

```
% Main PVS file for the property ex_these_lus_fibo1.
%
% Prove all IMPORTed files to prove the theorem ALL_OK here.

ex_these_lus_fibo1 : THEORY
BEGIN

  IMPORTING ex_these_lus_fibo1_lemma_TH1 ;
  IMPORTING ex_these_lus_fibo1_lemma_TH2 ;
  IMPORTING ex_these_lus_fibo1_lemma_TH3 ;
  IMPORTING ex_these_lus_fibo1_lemma_TH4 ;

ALL_OK : THEOREM
  TRUE AND TH3_value AND TH2_value AND TH4_value AND TH1_value ;

END ex_these_lus_fibo1
```

FIG. C.1 – Document maître ex_these_lus_fibo1.pvs

```
% Variables for the property ex_these_lus_fibo1
ex_these_lus_fibo1_variables : THEORY
BEGIN
  f_2_1 : int ;
  f_2_0 : int ;
  g_0_1 : int ;
  g_0_0 : int ;
  f_0_1 : int ;
  f_0_0 : int ;
  g_1_2 : int ;
  g_1_1 : int ;
  f_1_2 : int ;
  g_1_0 : int ;
  f_1_1 : int ;
  f_1_0 : int ;
  f_2_3 : int ;
  f_2_2 : int ;
END ex_these_lus_fibo1_variables
```

FIG. C.2 – Typage ex_these_lus_fibo1_variables.pvs

```
% A Lemma for ex_these_lus_fibo1.

% Expand the definition of TH1_value to prove it.

ex_these_lus_fibo1_lemma_TH1 : THEORY
BEGIN

  IMPORTING Lustre_helpers, ex_these_lus_fibo1_variables ;

TH1_value : bool =
(TRUE % clock values

% scalar antecedents
% flow antecedents
AND ((f_2_0=1))
) => % consequent
(((f_2_0>0)))
 ;

TH1 : LEMMA
  TH1_value ;

END ex_these_lus_fibo1_lemma_TH1
```

FIG. C.3 – Première obligation `ex_these_lus_fibo1_lemma_TH1.pvs`

```
% A Lemma for ex_these_lus_fibo1.

% Expand the definition of TH2_value to prove it.

ex_these_lus_fibo1_lemma_TH2 : THEORY
BEGIN

  IMPORTING Lustre_helpers, ex_these_lus_fibo1_variables ;

TH2_value : bool =
(TRUE % clock values

% scalar antecedents
AND ((f_1_0=1))
AND ((g_1_0=0))
AND (((f_1_0>0)))
% flow antecedents
AND ((f_2_1=(f_1_0+g_1_0)))
) => % consequent
(((f_2_1>0)))
 ;

TH2 : LEMMA
  TH2_value ;

END ex_these_lus_fibo1_lemma_TH2
```

FIG. C.4 – Deuxième obligation `ex_these_lus_fibo1_lemma_TH2.pvs`

```
% A Lemma for ex_these_lus_fibo1.

% Expand the definition of TH3_value to prove it.

ex_these_lus_fibo1_lemma_TH3 : THEORY
BEGIN

  IMPORTING Lustre_helpers, ex_these_lus_fibo1_variables ;

TH3_value : bool =
(TRUE % clock values

% scalar antecedents
AND (((f_0_0>0)))
AND ((f_1_1=(f_0_0+g_0_0)))
AND ((g_1_1=f_0_0))
AND (((f_1_1>0)))
% flow antecedents
AND ((f_2_2=(f_1_1+g_1_1)))
) => % consequent
(((f_2_2>0)))
 ;

TH3 : LEMMA
  TH3_value ;

END ex_these_lus_fibo1_lemma_TH3
```

FIG. C.5 – Troisième obligation `ex_these_lus_fibo1_lemma_TH3.pvs`

```
% A Lemma for ex_these_lus_fibo1.

% Expand the definition of TH4_value to prove it.

ex_these_lus_fibo1_lemma_TH4 : THEORY
BEGIN

  IMPORTING Lustre_helpers, ex_these_lus_fibo1_variables ;

TH4_value : bool =
(TRUE % clock values

% scalar antecedents
AND (((f_0_1>0)))
AND ((f_1_2=(f_0_1+g_0_1)))
AND ((g_1_2=f_0_1))
AND (((f_1_2>0)))
% flow antecedents
AND ((f_2_3=(f_1_2+g_1_2)))
) => % consequent
(((f_2_3>0)))
 ;

TH4 : LEMMA
  TH4_value ;

END ex_these_lus_fibo1_lemma_TH4
```

FIG. C.6 – Quatrième obligation `ex_these_lus_fibo1_lemma_TH4.pvs`

C.2 Théorie sous-jacente

Toutes les obligations de preuve s'appuient sur la théorie `Lustre_helpers`. Cette théorie fournit à PVS une définition possible des opérateurs Lustre suivants : `div` (division euclidienne), `mod` (reste de la division euclidienne), `int` (transformation des réels en entiers), `real` (transformation des entiers en réels) et `#` (l'opérateur logique « au plus un »).

C.2.1 Les opérateurs div et mod

Nous allons présenter les deux opérateurs de la division euclidienne ensemble, car leurs définitions sont liées. En effet, nous les avons définis de manière à vérifier les cinq propriétés suivantes :

1. décomposition $\forall i \in \mathbb{Z}, m \in \mathbb{Z}^*.\ i = (i\ mod\ m) + m(i\ div\ m)$
2. valeur du reste $\forall i \in \mathbb{Z}, m \in \mathbb{Z}^*.\ |i\ mod\ m| < |m|$
3. signe du reste $\forall i \in \mathbb{Z}, m \in \mathbb{Z}^*.\ (i\ mod\ m) \geq 0 \Leftrightarrow i \geq 0$
4. signe de la division $\forall i \in \mathbb{Z}, m \in \mathbb{Z}^*.\ (i\ div\ m) \geq 0 \Leftrightarrow \frac{i}{m} \geq 0$
5. valeur absolue $\forall i \in \mathbb{Z}, m \in \mathbb{Z}^*.\ |i|\ div\ |m| = |i\ div\ m|$

Pour éviter des vérifications de typage fastidieuses, nous avons décidé de définir la division et le reste sur tout le domaine des entiers relatifs. Pour cela, nous introduisons des constantes arbitraires,

```
div_rand : int ;
mod_rand : int ;
```

constantes qui représentent le résultat de la division (resp. du reste de la division) par zéro. Ainsi, les obligations de preuve pourront contenir une division par zéro (aucune erreur de typage ne sera levée), mais l'utilisateur ne pourra pas mener la preuve à bien, car ces constantes peuvent représenter n'importe quelle valeur.

```
div : [ int, int -> int ] =
  (LAMBDA (a,b : int) :
    IF b=0
    THEN div_rand
    ELSE IF b>0
         THEN (IF a>=0 THEN floor(a/b) ELSE -floor(-a/b) ENDIF)
         ELSE (IF a>=0 THEN -floor(a/-b) ELSE floor(a/b) ENDIF)
         ENDIF
    ENDIF) ;

mod : [ int, int -> int ] =
  (LAMBDA (a,b : int) :
    IF b=0
    THEN mod_rand
    ELSE IF b>0
         THEN (IF a>=0 THEN rem(b)(a) ELSE -rem(b)(-a) ENDIF)
         ELSE (IF a>=0 THEN rem(-b)(a) ELSE -rem(-b)(-a) ENDIF)
         ENDIF
    ENDIF) ;
```

On démontre que ces deux définitions vérifient bien les cinq propriétés de départ. Cette démonstration fait partie inhérente de la théorie `Lustre_helpers`, sous la forme des cinq théorèmes suivants :

```
div_mod_decomposition : THEOREM
  (FORALL (a,b:int) : b/=0 IMPLIES a = b*div(a,b)+mod(a,b)) ;

mod_absolute_value : THEOREM
  (FORALL (a,b:int) : b/=0 IMPLIES abs(mod(a,b))<abs(b)) ;

mod_sign : THEOREM
  (FORALL (a,b:int) :
    b/=0 IMPLIES ((mod(a,b)>0 IMPLIES a>=0)
              AND (a>=0 IMPLIES mod(a,b)>=0))) ;

div_sign : THEOREM
  (FORALL (a,b:int) :
    b/=0 IMPLIES ((div(a,b)>0 IMPLIES a/b>=0)
              AND (a/b>=0 IMPLIES div(a,b)>=0))) ;

floor_sign : SUBLEMMA
  (FORALL (a,b : int) : (a>=0 AND b>0) IMPLIES floor(a/b)>=0) ;

div_absolute_value : THEOREM
  (FORALL (a,b:int) : b/=0 IMPLIES abs(div(a,b))=div(abs(a),abs(b))) ;
```

Les définition de `mod` et `div` que nous avons données sont donc théoriquement correctes. Néanmoins, du point de vue pratique, elles ne sont pas faciles à manipuler : c'est pourquoi nous avons également démontré les lemmes suivants, qui permettent de faire des calculs effectifs sur les deux opératuers.

$\forall\ b \in \mathbb{Z}.\ b \neq 0 \ \Rightarrow\ (0\ div\ b) = 0$

```
div_zero_lemma : LEMMA
  (FORALL (b:int) : b/=0 IMPLIES div(0,b) = 0) ;
```

$\forall\ a \in \mathbb{N}, b \in \mathbb{Z}.\ b \neq 0 \ \Rightarrow\ (a\ div\ b) = \frac{a - rem(|b|)(a)}{b}$

```
div_pos_rem_lemma : LEMMA
  (FORALL (a:posint,b:int) : b/=0 IMPLIES div(a,b) = (a - rem(abs(b))(a))/b);
```

$\forall\ a \in -\mathbb{N}, b \in \mathbb{Z}.\ b \neq 0 \ \Rightarrow\ (a\ div\ b) = \frac{a + rem(|b|)(-a)}{b}$

```
div_neg_rem_lemma : LEMMA
  (FORALL (a:negint,b:int) : b/=0 IMPLIES div(a,b) = (a + rem(abs(b))(-a))/b);
```

L'opérateur `rem` utilisé dans ces lemmes est un opérateur standard de PVS : il définit le reste de la division euclidienne pour les entiers positifs. Nous avons également démontré des propriétés pour cet opérateur :

```
rem_pos_reduction : LEMMA
  (FORALL (a:nat,b:posnat) : a >= b IMPLIES rem(b)(a) = rem(b)(a-b)) ;
```

```
rem_pos_trivial : LEMMA
  (FORALL (a:nat,b:posnat) : a < b IMPLIES rem(b)(a) = a) ;

rem_neg_reduction : LEMMA
  (FORALL (a:negint,b:posnat) : a < 0 IMPLIES rem(b)(a) = rem(b)(a+b)) ;
```

En fait, si nous orientons les égalités de gauche à droite dans les six lemmes précédents, nous obtenons un système de réécriture confluent et terminant, qui calcule effectivement la division euclidienne. Par exemple :

`div(15,4)`	$\rightarrow$	`(15-rem(abs(4))(15))/4`	par `div_pos_rem_lemma`
	$\rightarrow$	`(15-rem(4)(15))/4`	par définition de `abs` (valeur absolue)
	$\rightarrow$	`(15-rem(4)(11))/4`	par `rem_pos_reduction`
	$\rightarrow$	`(15-rem(4)(7))/4`	par `rem_pos_reduction`
	$\rightarrow$	`(15-rem(4)(3))/4`	par `rem_pos_reduction`
	$\rightarrow$	`(15-3)/4`	par `rem_pos_trivial`
	$\rightarrow$	`3`	par application des règles d'arithmétique

C.2.2 Les opérateurs int et real

L'opérateur `real` sert à transformer un flot entier en un flot réel. En PVS, il existe déjà un mot-clé `real`, et c'est pourquoi nous avons décidé de désigner l'opérateur Lustre sous le nom de `int_to_real` :

```
int_to_real: [ int -> real ] =
  (LAMBDA (x:int): x) ;
```

En fait, PVS sait déduire que tout entier est un réel, si bien que cette transformation est une simple identité.

L'opérateur `int` est l'inverse du précédent : il permet de transformer des réels en entiers, en effectuant un arrondi. Nous avons choisi d'implémenter une manière possible d'arrondir, sachant que d'autres manières existent sur différentes plate-formes, notamment pour les réels négatifs :

```
real_to_int: [ real -> int ] =
  (LAMBDA (x:real): floor(5/10 + x)) ;
```

Nous pouvons alors démontrer que tout réel diffère de son arrondi d'au plus 5/10.

```
real_to_int_prop: LEMMA
  (FORALL (x:real,i:int): (i = real_to_int(x)) IMPLIES
    IF i<x THEN x-i < 5/10 ELSE i-x <= 5/10 ENDIF) ;
```

C.2.3 L'opérateur 'au plus un'

L'opérateur « au plus un » est un opérateur Lustre particulier, car il admet un nombre variable de paramètres booléens :

$$\#(b_1,b_2,\ldots,b_n)$$

Cet opérateur rend la valeur *vrai* si au plus un de ses arguments est vrai.

Nous avons modélisé le comportement de # en deux temps. D'abord, nous avons défini une structure arborescente (en « peigne droit ») pour représenter la structure de l'opérateur :

```
Big_Xor_Construct : DATATYPE
BEGIN
  one(a : bool) : one?
  two(a : bool, b : Big_Xor_Construct) : two?
END Big_Xor_Construct
```

Par exemple, l'expression `#(x,y,z)` serait représentée en PVS par

$$\texttt{two(x, two(y, one(z)))}$$

Ensuite, nous avons donné les règles d'évaluation d'une telle expression. La règle principale est celle qui vérifie qu'au plus un des arguments s'évalue à *vrai* :

```
evaluate_big_xor : [ Big_Xor_Construct -> bool ] =
  (LAMBDA (bx : Big_Xor_Construct) : count_big_xor(bx) <= 1) ;
```

Cette règle s'appuie sur la fonction `count_big_xor` qui, elle, compte le nombre d'arguments vrais :

```
count_big_xor : RECURSIVE [ Big_Xor_Construct -> int ] =
  (LAMBDA (bx : Big_Xor_Construct) :
    CASES bx OF
      one(a)   :  IF a THEN 1 ELSE 0 ENDIF,
      two(a,b) : (IF a THEN 1 ELSE 0 ENDIF) + count_big_xor(b)
    ENDCASES)
  MEASURE (LAMBDA (bx : Big_Xor_Construct) : size_big_xor(bx))
```

La fonction `count_big_xor` est récursive, et c'est pourquoi nous avons dû indiquer une mesure (`measure`) : cette dernière permet de prouver que la taille de l'argument diminue strictement à chaque appel récursif, si bien que la fonction `count_big_xor` termine.

```
size_one : [ bool -> nat ] = (LAMBDA (b:bool) : 1) ;
size_two : [ bool,nat -> nat ] = (LAMBDA (b:bool,n:nat) : n+1) ;
size_big_xor : [ Big_Xor_Construct -> nat ]=reduce_nat(size_one,size_two) ;
```

C.3 Stratégies de réécriture

Outre les définitions théoriques de certains opérateurs Lustre, présentées dans la partie précédente, nous avons élaboré un certain nombre de stratégies de preuve pour PVS, fondées sur la réécriture.

C.3.1 Réécriture des opérateurs Lustre

La première stratégie, `lus-macros`, installe en fait le système confluent et terminant, dont nous avons parlé dans la section C.2.1. En plus, cette stratégie permet aussi d'évaluer l'opérateur « au plus un » :

```
(defstep lus-macros ()
  (then (auto-rewrite!! "count_big_xor")
        (auto-rewrite!! "evaluate_big_xor")
        (auto-rewrite!! "div_zero_lemma")
        (auto-rewrite!! "div_pos_rem_lemma")
        (auto-rewrite!! "div_neg_rem_lemma")
        (auto-rewrite!! "rem_pos_reduction")
        (auto-rewrite!! "rem_neg_reduction")
        (auto-rewrite!! "rem_pos_trivial")
        (auto-rewrite!! "mod")
        (auto-rewrite!! "abs")
  )
"Installs effective rewrite rules for 'div', 'rem' and 'big_xor'.
 These rules calculate upon the beformentioned expressions as long
 as these have integer [resp. boolean] constants as arguments."
"Installing rewrite rules for Lustre operators 'div', 'rem' and #."
)
```

Ainsi, en présence d'une expression faisant intervenir `mod`, `div` ou `#`, l'utilisateur pourra utiliser la règle `(lus-macros)`, qui aura pour effet de réduire les opérateurs cités autant de fois que possible, jusqu'à les éliminer éventuellement.

C.3.2 Élimination de variables

Une deuxième stratégie, `lus-vars`, est destinée à diminuer la taille du séquent dans la fenêtre de PVS, en diminuant le nombre d'antécédents. Ainsi, son but premier est de rendre le séquent plus lisible.

Cette stratégie est motivée par le fait qu'en pratique, au fil des différentes réductions, un certain nombre d'antécédents se présentent sous la forme

$$variable_1 = variable_2$$

Alors, la stratégie `lus-vars` détecte ces antécédents, remplace partout $variable_1$ par $variable_2$ et efface l'antécédent devenu inutile : elle effectue en fait un calcul de quotient du séquent par l'ensemble des classes d'équivalence des variables.

```
(defstep lus-vars ()
  (let
    ((antes
       (gather-fnums
        (s-forms *goal*)
        '-
        nil
        #'(lambda (sf)
            (if (and (negation? (formula sf))
                     (equality? (args1 (formula sf))))
                (and (name-expr? (args1 (args1 (formula sf))))
                     (name-expr? (args2 (args1 (formula sf)))))
                nil))))
    )
    (if (null antes)
        (skip)
        (then (replace* antes) (hide antes)))
  ) "" ""
)
```

C.3.3 Élimination des constantes

Enfin, une dernière stratégie de réécriture, `lus-consts`, élimine les antécédents de la forme

$$constante = variable \qquad \text{ou} \qquad variable = constante$$

Comme précédemment, on remplace partout la variable par la constante correspondante et on efface l'antécédent qui a permis de faire la substitution.

Cette stratégie est définie à l'aide de plusieurs sous-stratégies. `lus-rvv1` élimine touts les antécédents $variable = constante$.

```
(defhelper lus-rvv1 ()
  (let
    ((antes
       (gather-fnums
          (s-forms *goal*)
          '-
          nil
          #'(lambda (sf)
              (if (and (negation? (formula sf))
                       (equality? (args1 (formula sf))))
                  (and (name-expr? (args1 (args1 (formula sf))))
                       (number-expr? (args2 (args1 (formula sf)))))
                  nil))))
    )
    (if (null antes) (skip) (then (replace* antes) (hide antes)))
  ) "" ""
)
```

`lus-rvv2` élimine un antécédent de la forme *constante* = *variable*

```
(defhelper lus-rvv2 ()
  (let
    ((antes
       (gather-fnums
         (s-forms *goal*)
         '-
         nil
         #'(lambda (sf)
             (if (and (negation? (formula sf))
                      (equality? (args1 (formula sf))))
                 (and (name-expr? (args2 (args1 (formula sf))))
                      (number-expr? (args1 (args1 (formula sf)))))
                 nil))))
    )
    (if (null antes)
        (skip)
        (let ((fn (car antes)))
             (then (replace fn :dir RL) (hide fn)))))
"" ""
)
```

`lus-rvv3` élimine tous les antécédents de la forme *constante* = *variable*.

```
(defhelper lus-rvv3 ()
  (try (lus-rvv2) (lus-rvv3) (skip))
"" ""
)
```

Et enfin, voici la définition de la stratégie d'élimination totale :

```
(defstep lus-consts ()
  (try (lus-rvv1)
       (lus-consts)
       (try (lus-rvv3) (lus-consts) (skip)))
"Finds all antecedents of the form 'constant = variable' or
 'variable = constant', rewrites the sequent by these equations and
 hides them.
 If no such antecedent exists, the sequent is left unchanged."
"Propagating constants."
)
```

Bibliographie

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. 102

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2) :253–284, 1991. 13, 107

[3] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1995. 13, 88, 97

[4] J.R. Abrial. B : A formalism for complete correct system development. Conference given at Inria Rhône-Alpes, October 1999. 117

[5] L. Arditi, A. Bouali, H. Boufaied, G. Clav'e, M. Hadj-Chaib, and R. de Simone. Using esterel and formal methods to increase the confidence in the functional validation of a commercial dsp, 1999. 14

[6] R. J. R. Back and J. von Wright. Refinement calculus : Part I and II. In *Proceedings on Stepwise refinement of distributed systems : models, formalisms, correctness*, pages 42–63 and 67–93. Springer-Verlag New York, Inc., 1990. 13

[7] K. Baukus, Y. Lakhnech, K. Stahl, and M. Steffen. Divide, Abstract and Model-Check. In *Proceedings of the 5th and 6th SPIN Workshops on Theoretical and Practical Aspects of Model Checking*, volume 1680 of *LNCS*. Springer–Verlag, 1999. 12

[8] J.L. Bergerand and E. Pilaud. SAGA ; a software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988. 14, 87

[9] D. Bert. Building Lustre synchronous control systems from B abstract machines : A case study. Technical report, LSR-IMAG, 1997. 13

[10] F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, 79(9) :1293–1304, september 1991. 14

[11] J. P. Bowen, A. Fett, and M. G. Hinchey, editors. *ZUM'98 : The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, 24–26 September 1998*, volume 1493 of *lncs*. Springer-Verlag, 1998. 13, 88

[12] D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December 1994. ERA Technology. 14, 87

[13] P. Caspi and M. Pouzet. Lucid Synchrone, une extension fonctionnelle de Lustre. In *Journées Francophones des Languages Applicatifs*, pages 1–20. INRIA, février 1999. conférence invitée. 14

[14] Th. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*. Springer Verlag, 1993. 43, 73

[15] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Principles of Programming Languages*, January 1977. 12, 60

[16] C. Dumas and P. Caspi. A PVS proof obligation generator for Lustre programs. In *7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, 2000. 29

[17] Cécile Dumas. Méthodes déductives pour la preuve de programmes lustre. Thèse de doctorat de l'université Joseph Fourier, octobre 2000. 13, 24, 29, 32, 92

[18] Cécile Dumas. Méthodes déductives pour la preuve de programmes lustre. Thèse de doctorat de l'université Joseph Fourier, octobre 2000. 15

[19] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs, TYPES'94*, volume 996 of *Lecture Notes in Computer Science*. Springer Verlag, 1995. 82

[20] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993. 14

[21] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag. 29, 93, 98

[22] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991. 14, 23

[23] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5) :279–295, 1997. 12

[24] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. rapport technique 204, INRIA, Août 1997. Version révisée distribuée avec Coq. 12

[25] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, Venezia (Italy), September 1999. 14, 94

[26] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990. 13, 88

[27] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, May 1994. 13, 107

[28] G. LeGoff. Using synchronous languages for interlocking. In *First International Conference on Computer Application in Transportation Systems*, 1996. 14, 87

[29] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL : a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2) :362–374, 1986. 14, 107

[30] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6 :1–32, 1995. 60

[31] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. 12

[32] Jan Mikáč. Raffinement pour Lustre. rapport de DEA, 2002. unpublished. 97

[33] Jan Mikáč and Paul Caspi. How many times should a program be unfolded for proving invariant properties ? Technical Report TR-2004-9, Verimag Technical Report, 2004. 55

[34] Jan Mikáč and Paul Caspi. Flush : A system development tool based on scade/lustre. In T. Margaria and M. Massink, editors, *Tenth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 05)*, Lison, Portugal, 2005. ACM. 117

[35] Jan Mikáč and Paul Caspi. Temporal refinement for Lustre. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science. 95

[36] P.S. Miner and S.D. Johnson. Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit. In M. Sheeran and S. Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Båstad, Sweden, 1996. Springer–Verlag. 80

[37] Lionel Morel. Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille. Thèse de doctorat de l'INPG, mars 2005. 12

[38] P. Caspi S. Tripakis N. Scaife, C. Sofronis and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo, editor, *4th International Conference on Embedded Software, EMSOFT04*. ACM, 2004. 14

[39] D. Nowak, J.R. Beauvais, and J.P. Talpin. Co-inductive axiomatization of a synchronous language. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 387–399. Springer Verlag, October 1998. http ://www.irisa.fr/prive/nowak/publis/tphols98.ps.gz. 107

[40] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1) :23–58, January 2000. 14

[41] Sam Owre Natarajan Shankar John Rushby D.W.J Stringer-Calvert. PVS language reference. Technical report, SRI International, December 2001. 12

Résumé des chapitres

1. Pour délimiter le cadre de cette thèse, nous la replaçons dans le contexte des méthodes formelles appliquées aux systèmes réactifs synchrones. Nous citons également un travail antérieur, sur lequel elle s'appuie et qu'elle prolonge directement. Nous terminons en annonçant le plan du document.
2. Dédié à la présentation du langage Lustre, ce chapitre commence par une introduction informelle avec exemple. Ensuite, la définition formelle décrit tour à tour la syntaxe et la sémantique des flots, des expressions sur les flots et des fonctions de flots. Pour terminer, on s'attaque à la question des horloges et du synchronisme.
3. Pour présenter la méthode de preuve de Cécile Dumas, nous procédons en plusieurs étapes : après avoir défini le problème auquel nous nous attaquons, nous énumérons les règles de base utilisées dans la stratégie de preuve. Vient ensuite un exemple d'application, suivi du reste des règles, celles d'usage plus restreint. Pour terminer, nous évoquons des limitations de la méthode et une tentative incorrecte pour y remédier.
4. Nous essayons ici de corriger la règle incorrecte d'induction continue fragmentée. Nous justifions qu'il est possible de fragmenter et nous déduisons une règle correcte, mais elle s'avère d'un intérêt pratique très limité.
5. Nous prouvons l'équivalence entre la preuve d'une propriété invariante et la preuve de la même propriété dépliée n fois. Nous en déduisons une règle d'induction continue généralisée aux dépliages. Se pose alors la question du nombre dépliages : nous donnons une caractérisation des propriétés prouvables par induction et en déduisons une règle pour fixer ce paramètre.
6. Cette présentation de l'outil de preuve Gloups procède du général au particulier. D'abord, nous exposons nos motivations pour implémenter une nouvelle version de cet outil et nous proposons un tour du propriétaire de la nouvelle version. Ensuite, nous expliquons l'algorithme de preuve utilisé, ainsi que la forme générale des obligations produites. Puis, nous détaillons deux optimisations importantes qui nous ont permis de repousser les limites de notre logiciel. Pour finir, nous proposons un exemple d'application réel.
7. Dans ce chapitre, nous exposons les motivations des recherches présentées dans la troisième partie de ce mémoire. Après avoir souligné l'intérêt du raffinement en général, nous présentons un cadre bien établi de développement par raffinements, la méthode B. Cette présentation nous conduit à une première manière d'introduire le raffinement en Lustre, manière qui s'avère peu pratique. Nous terminons en annonçant une autre technique de raffinement pour Lustre, qui sera développée dans les chapitres suivants.
8. Nous établissons le calcul de raffinement pour Lustre en deux phases : d'abord, nous instaurons un cadre général correct, puis à l'intérieur de ce cadre, nous délimitons une partie adaptée à la fois au langage et à nos moyens de preuve pratiques. Nous terminons par une discussion des points les plus importants du raffinement proposé.
9. Pour illustrer le raffinement proposé, nous détaillons dans ce chapitre deux exemples de développement par raffinements. Le premier exemple montre la technique et la mécanisation du raffinement dans le cas d'un seul nœud ; le second exemple fait intervenir plusieurs nœuds.
10. Nous concluons sur l'ensemble de nos travaux et proposons des perspectives pour de nouvelles recherches.

Raffinements et preuves de systèmes Lustre

Résumé Notre thèse se situe dans le domaine des méthodes formelles appliquées aux systèmes réactifs. Nous modélisons et traitons ces systèmes, en continuelle interaction avec leur environnement, grâce au langage synchrone Lustre.

D'abord, sur la base d'un travail précurseur, nous établissons pour Lustre une méthode de preuve inductive des propriétés de sûreté. Cette méthode est optimisée, afin de prendre en compte au mieux la dynamique des systèmes. Elle est implémentée dans un outil de preuve, Gloups.

Ensuite, suivant le modèle de la méthode B, nous définissons un calcul de raffinement pour Lustre. Ce calcul est à la fois adapté à Lustre et exprimé en ce langage. Les obligations de preuve qui assurent la correction du raffinement peuvent être traitées par Gloups. Pour faciliter le développement, un autre outil, Flush, génère automatiquement les obligations pour Gloups.

Ainsi, nous utilisons Lustre à la fois comme langage de programmation et comme cadre formel d'un développement maîtrisé. L'intérêt de ce procédé réside dans la simplicité du langage et dans son adaptation aux systèmes réactifs : en ce domaine, notre méthode de raffinement est suffisamment expressive, sans être inutilement compliquée. Des exemples viennent démontrer l'intérêt de la méthode.

Mots-clés systèmes réactifs, méthodes formelles, programmation synchrone, raffinement, preuve inductive, invariants

Refinements and Proofs of Lustre Systems

Abstract This thesis is set into the domain of formal methods applied to reactive systems. These systems, characterized by their continuous interaction with their environment, are modeled and managed by the means of the synchronous programming language Lustre.

Firstly, thanks to a previous work, we build an induction-based proof method of Lustre safety properties. The method is optimized in that it takes into account the dynamics of the systems in the best possible way. We implemented it in the Gloups proof tool.

Then we define a refinement calculus for Lustre which follows the B method model. The calculus is both adapted to and expressed in Lustre. The proof obligations which ensure the refinement correctness can be handled by Gloups. In order to simplify the development, another tool called Flush automatically generates the proof obligations for Gloups.

Thus we use Lustre as both a programming language and a formal development framework. The interest of our method stems from the simplicity of Lustre and its adaptation to the reactive systems : in this domain, our refinement method is expressive enough, yet not exceedingly complex. To finish, we show the interest on some examples.

Keywords reactive systems, formal methods, synchronous programming, refinement, inductive proof, invariants