Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille

Lionel Morel

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 COMMUNICATION »

préparée au laboratoire VERIMAG
dans le cadre de l'**École doctorale "MATHÉMATIQUES, SCIENCES ET TECHNOLOGIES DE L'INFORMATION"**

présentée et soutenue publiquement

par

Lionel MOREL

le 16 Mars 2005

Titre :

Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille

Directrice de thèse :

Florence Maraninchi

JURY

Jacques Chassin de Kergommeaux	Président
Marc Pouzet	Rapporteur
Patrice Quinton	Rapporteur
Florence Maraninchi	Directrice de thèse
Jean-Louis Colaço	Examinateur

Remerciements

Je tiens en premier lieu à remercier chaleureusement les membres du jury :
Monsieur Jacques Chassin-de-Kergommeaux professeur à l'Institut National Polytechnique de Grenoble de m'avoir fait l'honneur d'accepter de présider mon jury de thèse. Merci infiniment d'avoir surmonter des difficultés personnelles afin d'assurer ce rôle.
Messieurs Patrice Quinton, Professeur des Universités et Directeur de l'ENS Cachan - Antenne de Bretagne et Mr Marc Pouzet, Maitre de Conférence à l'Université Pierre et Marie Curie, Paris 6 d'avoir accepté de juger ce travail. Merci pour l'intérêt qu'ils ont porté à ce travail, pour les remarques nombreuses et constructives qui m'ont fait voir plus loin.
Monsieur Jean-Louis Colaço de la société Esterel-Technologies d'avoir accepté de participer au jury.
Madame Florence Maraninchi, Professeur à l'Institut National Polytechnique de Grenoble d'avoir dirigé ce travail. Merci pour ces 6 années pendant lesquelles elle m'a beaucoup appris de la recherche et de l'informatique en général. Merci pour les discussions nombreuses, sa grande patience, ses explications limpides, ses questions judicieuses, ses relectures nombreuses du manuscript.

Je tiens également à remercier toutes les personnes qui ont rendu possible la réalisation de ce travail : Monsieur Joseph Sifakis, directeur de recherche au CNRS, de m'avoir accueilli au sein du laboratoire Verimag ;
Tous les membres de l'équipe synchrone avec qui j'ai eu (et ai toujours) un grand plaisir à travailler. Merci en particulier à Erwan, Fabien et Yvan pour leur travail impeccable qui fait que celui des autres peut marcher. Merci également au reste du laboratoire verimag, membres permanents, étudiants et personnel administratif pour l'ambiance très agréable qui règne chaque jour. Je remercie également tous les collègues de l'UFRIMA et de l'ENSIMAG avec qui j'ai partagé de nombreux moments studieux tout au long de cette thèse.
Je remercie également Cyril et Moussa pour les années d'étude et de thèse que nous avons supporté ensemble. Merci à Jan pour tous ses petits trucs latex/java et pour travailler sur Gloups, qui marche ! et Liana pour avoir aidé à installer dans le bureau une ambiance ma foi fort agréable, merci pour les biscuits (j'attends toujours la récompense pour les corrections de fautes de français).
Et puis tous les autres: Doms, Franck et Josselin pour avoir été là dans l'aventure Titch Ka Ra qui, je l'espère, continuera, pleine de disques et de concerts. Thierry et les buveurs de guinness des sessions irlandaises pour les soirées détente (même si se coucher à 2h après avoir bu de la guinness toute la soirée en jouant de la musique, ça fatigue aussi). Merci à tous mes amis, Sylvain, Marie-Laure, Gauthier, Sylvie, Bénédicte, Stéphane, Julia, et tous les autres pour tous les moments partagés, les soirées endiablées et les sorties en montagne qui vident la tête. Merci également à tous ceux de l'Alpes-Club pour les innombrables sorties montagne. Merci à Michaël pour les séances escalades, les bavantes à peau de phoques. Merci pour le 6b, pour les devers, je m'occupe des dalles, des dièdres et des photos.
Merci enfin à ma famille, plus ou moins proche (la citer toute entière serait un peu trop long). Merci à mon grand frère, qui est déjà passé par là et qui a été et est toujours bien plus qu'un grand frère. Merci à lui pour le goût de la musique, les prêts d'instruments et le mentoring efficace. Merci à mes parents, sans qui je n'aurais jamais fait d'étude. Merci pour leur soutien durant ces longues années, merci pour l'amour de la montagne que vous m'avez légué et qui me fait vivre tous les jours, et le goût du travail bien fait qui m'a, sans aucun doute, mené jusqu'ici.
Merci à Séverine, surtout, de supporter mes humeurs, mes retours tardifs le soir, mes oublis, mes engagements dans tous les sens qui prennent beaucoup de notre temps. Merci pour le pot, ce travail-là aussi mérite reconnaissance ! Merci d'être là, toujours, à mes côtés. Merci pour tout, le reste et ce qui est encore à venir...

Table des matières

Chapitre 1

Introduction

1.1 Contexte général : développement des systèmes réactifs

On peut classer les systèmes informatiques en trois grandes catégories selon la relation qu'ils entretiennent avec leur environnement. La première catégorie est celle des systèmes *transformationnels*. Ces systèmes sont conçus pour calculer un résultat final à partir d'un ensemble de données d'entrée disponibles au début de leur exécution. L'interaction qui existe entre un tel système et son environnement est très simple : lecture de donnée fournie par l'environnement au début de l'exécution ; écriture d'un résultat fourni à l'environnement en fin d'exécution. Un compilateur est un exemple typique de système transformationnel.

La seconde catégorie est celle des systèmes dits *interactifs*, tels que les environnements graphiques que nous utilisons tous sur nos ordinateurs. L'interaction système/environnement est ici plus fine, puisque les échanges peuvent être fréquents pendant l'exécution du système, mais elle est principalement dirigée par le système : c'est lui qui en impose le rythme.

La troisième et dernière catégorie, qui nous intéresse plus particulièrement dans ce travail, est celle des systèmes dits *réactifs*. L'interaction environnement/système est encore ici permanente, mais le rythme des échanges est fixé par l'environnement (voir figure 1.1). Le comportement d'un tel système peut être *discrétisé* et vu comme la succession des valeurs d'entrées E et de sorties S aux différents instants de l'exécution : $(< E_1, S_1 >, < E_2, S_2 >, ..., < E_i, S_i >, ...)$ à un rythme imposé par l'environnement. Un tel système a une mémoire, notée ω, qui représente une abstraction de l'histoire de l'environnement et peut être utilisée pour calculer les nouvelles valeurs des sorties.

De manière orthogonale, on peut aussi définir une catégorie de systèmes dits *embarqués*. Par systèmes embarqués, on désigne tous les systèmes informatiques qui sont incorporés dans des systèmes physiques plus ou moins complexes : voiture et autres moyens de transports, téléphonie, mais aussi systèmes de contrôle/commande de processus industriels comme dans les centrales nucléaires.

Parmi les systèmes embarqués, la catégorie des systèmes dits *critiques* fait l'objet d'une attention particulière depuis les vingt dernières années. Un « *système critique* », est simplement un *système dont le dysfonctionnement peut avoir des conséquences graves sur son environnement*. Un système de pilotage de centrale nucléaire est *critique*, de même que le pilote automatique d'un avion ou le système de freinage ABS d'une voiture. Que ce soit pour la description ou la validation de ces systèmes, on doit se donner des outils formels qui permettent de garantir leur bon fonctionnement.

Dans cette thèse, nous nous intéressons à la description et à la validation des systèmes *réactifs critiques*. Une contribution importante dans ce domaine a été la définition de l'approche dite *synchrone* que nous présentons au paragraphe suivant.

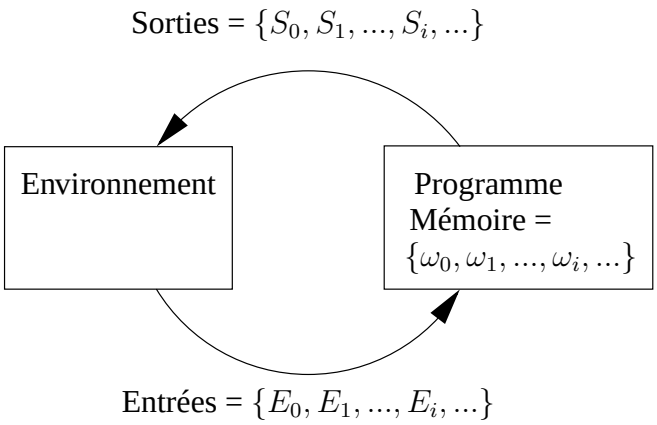


FIG. 1.1 – Interaction discrétisée d'un système réactif avec son environnement.

1.2 L'approche synchrone

1.2.1 Introduction

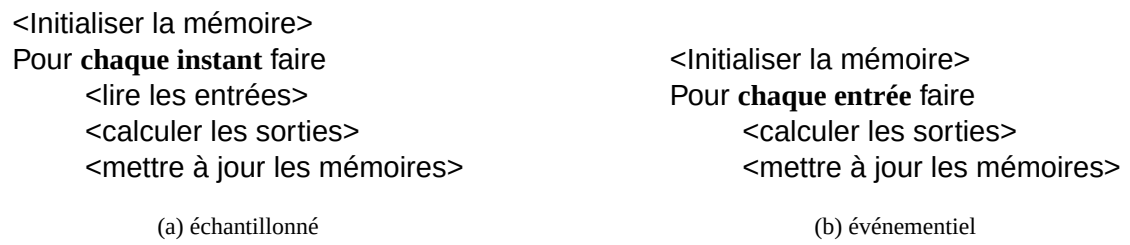
Apparue au début des années 80, l'approche synchrone [Hal93a] a représenté une contribution importante dans le domaine de la programmation des systèmes réactifs. Elle repose sur l'hypothèse de synchronisme, qui établit que le temps de réaction d'un système est nul. D'un point de vue externe, cela veut dire que les valeurs de sorties du système sont produites de façon simultanée avec la réception des entrées. Clairement, ce choix de conception est impossible à implémenter dans la pratique. Néanmoins, un système synchrone est tout à fait réalisable du moment qu'il « *semble* » fonctionner de manière instantanée vu depuis son environnement. Pour cela, il suffit d'assurer que le programme calcule ses sorties dans un temps inférieur au temps de cycle de son environnement. Par exemple, si les modifications significatives dans les entrées du système arrivent au plus toutes les secondes, alors le système doit être capable de calculer les sorties correspondantes en moins d'une seconde.

Lorsqu'on spécifie un système réactif à l'aide d'un langage synchrone, on doit vérifier que le code produit par compilation exécute bien une réaction suffisamment rapidement. On calcule donc un temps d'exécution « *au pire* » (*WCET* pour l'anglais *Worst Case Execution Time*). Ce calcul est habituellement rendu difficile par les constructions récursives ou les boucles présentes dans les langages de programmation classiques. La structure du code produit pour un langage synchrone est telle qu'on peut *toujours* fournir une sur-approximation du temps d'exécution.

D'un point de vue *interne*, l'hypothèse de synchronisme établit que le délai de communication entre les composants d'un programme est nul. La sémantique de composition dans un langage synchrone est donc simple.

1.2.2 Langages synchrones

Dans la famille des langages synchrones [BCE+03] on distingue deux schémas d'exécution. Le premier est un schéma dit *échantillonné* (figure 1.2(a)) dans lequel l'exécution du programme est fixée par une horloge globale et peut être vue comme une boucle infinie. A chaque passage dans la boucle, le système commence par lire des nouvelles valeurs d'entrées. Il calcule ensuite les valeurs correspondantes pour les variables de sorties (en prenant en compte la valeur courante des entrées *et* de la mémoire) et met à jour la mémoire.



(a) échantillonné (b) événementiel

FIG. 1.2 – Schémas d'exécution synchrones.

Le second schéma d'exécution possible est dit *événementiel* (voir figure 1.2(b)). Le rythme d'exécution est cette fois-ci fixé par la réception d'un ou plusieurs signaux d'entrées. Le système calcule alors les valeurs correspondantes pour les variables de sortie et met à jour la mémoire, avant de se mettre en attente d'un nouvel événement d'entrée. Le calcul d'une valeur de sortie ne nécessite pas forcément, dans ce cas, la présence de toutes les entrées.

Plusieurs langages synchrones ont été proposés, adoptant tous des styles de description différents, chacun adapté à des domaines d'applications différents : LUSTRE [HCRP91], ESTEREL [BG92], SIGNAL [GB87], ARGOS [MR01]. Nous présentons maintenant succinctement chacun de ces langages[1].

ESTEREL est un langage impératif dédié à une description *orientée contrôle* des systèmes. Il adopte le schéma d'exécution *événementiel* décrit plus haut. Un programme ESTEREL est un ensemble de processus concurrents (potentiellement imbriqués) dont l'exécution est synchronisée sur une unique horloge globale. Au début de chaque réaction (identifiée par l'arrivée de nouvelles valeurs d'entrée), chaque processus reprend son exécution à partir du point où il avait été interrompu à la réaction précédente, exécute une séquence d'instructions impératives et enfin, soit il termine, soit il se remet en pause, attendant ainsi la réaction suivante. Le langage ARGOS, inspiré des STATECHARTS [HP85] est assez proche d'ESTEREL : les systèmes sont décrits comme des compositions parallèles et hiérarchiques d'automates qui décrivent des réactions locales, un peu comme les processus ESTEREL.

LUSTRE et SIGNAL sont quant à eux des langages déclaratifs dans lesquels une sortie du système est définie par une *équation* (relation d'équivalence entre le nom de la variable et l'expression), non pas par une affectation. Les deux langages se différencient par les moyens qu'on a pour spécifier le rythme d'exécution d'un programme. Pour LUSTRE, il existe une horloge globale unique. Il est possible de décrire des systèmes plus lents que cette horloge globale en définissant des horloges locales qui sont toujours en relation avec l'horloge globale. Un programme LUSTRE, qui manipule des variables comme des flots de valeurs (les valeurs que prennent les variables du système dans les passages successifs du corps de boucles de la figure 1.2(a)), est toujours compilé vers un schéma *échantillonné*. En SIGNAL, par contre, on peut définir des horloges qui n'ont pas nécessairement de relations les unes avec les autres. Une variable peut alors avoir une valeur « absente ». Il est possible de compiler un programme SIGNAL dans chacun des schémas *échantillonné* ou *événementiel*. A l'opposé d'ESTEREL, ces langages sont adaptés à la description des systèmes orientés données.

Ces trois langages sont adaptés à la description de systèmes différents. Dans la pratique, il est fréquent de rencontrer des systèmes complexes dont certaines parties sont plus facilement décrites par une approche *donnée* et d'autres par une approche *contrôle*. Plusieurs travaux [JLMR94, MR03] ont

[1] Une partie des explications qui suivent sont directement inspirées de [BCE+03] et de l'introduction de [Gau03]

proposé la mixité des approches notamment par la description des systèmes sous la forme d'automates (des processus type ESTEREL) dont les états correspondent à des sous-comportements décrits, par exemple, en LUSTRE.

1.2.3 LUSTRE : un langage synchrone flot-de-donnée

Nous l'avons évoqué plus haut, LUSTRE est un langage flot-de-donnée : chaque variable X représente un flot de valeurs infini $(X_1, X_2, ...X_i, ...)$, la valeur X_i étant la valeur prise par X à l'instant i de l'exécution du programme. Le paradigme flot-de-donnée a été introduit dans le courant des années 70 [Kah74] avant d'être mis en pratique notamment dans le langage LUCID [WA85a][2].

Un programme LUSTRE, appelé *nœud*, définit ses sorties par un ensemble d'équations qui doit être vu comme une conjonction d'équivalences entre les variables et les expressions qui les calculent. Chaque équation définit la valeur que prend la variable à chaque instant, en fonction des valeurs courantes des entrées et de mémoires (symbolisées par un opérateur pre) des variables de sorties. L'ordre des équations n'a aucune importance et un des rôles du compilateur consiste à déterminer un ordre de dépendance des variables locales et de sorties afin de séquentialiser les calculs de ces variables pour obtenir les actions impératives permettant d'exécuter un instant du comportement du système. On interdit notamment les dépendances instantanées afin de garantir l'existence et de la séquentialisation.

Par ailleurs, le domaine d'application visé (systèmes critiques) impose des choix au niveau du langage. On interdit par exemple toute forme d'allocation dynamique de mémoire ; la taille d'un tableau doit être connue statiquement ; on interdit toute forme d'itération (type for, while) dont on ne peut pas garantir l'arrêt statiquement.

Le langage LUSTRE a fait l'objet d'un intérêt important de la part d'industriels développant des applications de contrôle critiques. Il est le langage cœur de l'outil SCADE, commercialisé par *Esterel Technologies* qui est utilisé par des entreprises dans le domaine des transports (EADS, Airbus, RATP, SNCF) et du contrôle de processus industriel notamment autour de la gestion et de la production de l'énergie électrique (Schneider-Electric, EDF).

1.3 Motivations - Objectifs du travail

Cette thèse a pour cadre général la description et la validation des systèmes réactifs à l'aide du langage synchrone LUSTRE. Nous présentons celui-ci plus en détail au chapitre 2.

1.3.1 Structures régulières de programmes : tableaux et itérateurs

Notre première motivation était une extension du langage pour permettre la description des programmes réguliers au travers d'itérateurs de tableaux et l'étude de la compilation de cette extension vers du code efficace. Puis, nous avons cherché à tirer partie de cette structuration pour la validation des programmes. Les quelques paragraphes suivants tentent de retracer le cheminement suivi.

Les tableaux ont été introduits dans la version 4 de LUSTRE [Roc92] dans le but de faciliter la description des systèmes matériels. Dans ce cas, les tableaux sont forcément expansés en variables indépendantes qui sont elles-mêmes traduites en *fils* dans le circuit. En effet, la structure tableau n'est pas exploitée au niveau circuit : au bout du compte, chaque élément de tableau est représenté par un fil indépendant dans le circuit. Les opérateurs proposés dans le langage sont tout à fait adaptés à cette

[2] le nom LUSTRE est un acronyme pour « LUcid Synchrone Temps-RÉel »

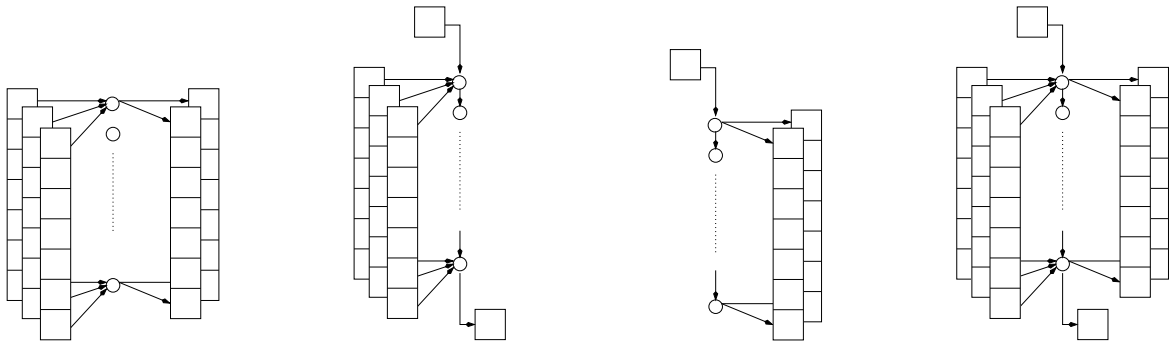


FIG. 1.3 – Les 4 itérateurs map, red, fill et map_red.

approche : définition par tranches des tableaux, récursivité statique, etc. Le paragraphe 4.1 rappelle ces opérateurs.

LUSTRE étant surtout utilisé pour le développement de logiciels, on s'est vite rendu compte de l'inconvénient de l'expansion systématique des tableaux : le code expansé est gros et lent. A la place, on pourrait très bien imaginer générer du code avec des tableaux et des boucles.

La version 5 du compilateur a représenté un premier pas dans cette direction. Il y était proposé de générer du code avec tableaux et boucles à partir des opérateurs de la version 4 du langage. Comme nous le verrons plus tard, une telle génération n'est pas toujours possible et cette solution n'a pas été retenue.

Nous avons alors proposé l'introduction de nouveaux opérateurs dans le langage, permettant la manipulation des tableaux de manière à ce que la génération de code à tableaux et boucles soit toujours possible. Ces itérateurs, que nous présentons au chapitre 4, sont directement inspirés des langages fonctionnels. On définit (voir figure 1.3) :

- l'opérateur map qui permet d'appliquer un même calcul à tous les éléments d'un (ou plusieurs) tableau(x) ;
- l'opérateur red (appelé parfois fold dans le cadre des langages fonctionnels) qui permet de calculer un accumulateur en parcourant successivement tous les éléments d'un (ou plusieurs) tableau(x) ;
- l'opérateur fill qui permet de remplir un (ou plusieurs) tableau(x) itérativement à partir d'une valeur initiale, en appliquant un même calcul pour calculer chaque élément ;
- enfin l'opérateur map_red qui généralise les trois précédents et permet de calculer à la fois un (ou plusieurs) tableau(x) et un accumulateur à partir d'un (ou plusieurs) tableau(x) d'entrée et d'une valeur initiale.

Ces opérateurs donnent un nouveau visage à l'utilisation des tableaux dans le langage : ceux-ci permettent de structurer les données manipulées par les programmes, mais ils permettent surtout, par l'utilisation des itérateurs, de *structurer les programmes* eux-mêmes.

L'intérêt pour de tels opérateurs *réguliers* est notamment né lors une collaboration avec la société Airbus sur une application de gestion de la répartition des charges électriques dans un avion de ligne. L'essentiel des calculs présents dans ce programme était réguliers et se voyait compilé vers du code avec une duplication en grand nombre des composants (voir chapitre 7). L'introduction des itérateurs a permis de réduire de manière considérable la taille du code généré pour cette application. L'étude de cas a été concluante, et les itérateurs ont été introduits dans le compilateur expérimental de la société *Esterel Technologies*, qui commercialise SCADE, la version industrielle de LUSTRE. Cette introduction a représenté une implémentation directe des propositions que nous faisons au chapitre 4.

1.3.2 Spécification locale par contrat des composants

En parallèle, nous avons étudié une notion de spécification partielle des composants. Globalement, les méthodes de validation associées à LUSTRE ont habitué les utilisateurs à écrire des propriétés sur leurs programmes. Ils spécifient en général : une *assertion* représentant les hypothèses faites sur l'environnement du système et la *propriété* que le système doit satisfaire si l'assertion est satisfaite. Dans le cas de LUSTRE, ces propriétés sont exprimées en utilisant le même langage que pour décrire les programmes eux-mêmes, et un ensemble d'outils est disponible pour prouver formellement ces propriétés.

On peut utiliser un processus de *validation modulaire* (voir [HLR93]), où on commence par prouver qu'un composant P satisfait une propriété ω et où on utilise ensuite ω directement, en abstrayant P, pour prouver que les autres composants satisfont bien les propriétés qu'on leur a associées.

D'un point de vue langage, on peut envisager d'imposer une forme de spécification locale des composants qui décrit partiellement le comportement d'un composant. Cette spécification servira tout d'abord dans l'étape de description du système puis dans l'étape de validation, comme nous venons de le décrire.

Pour décrire ces spécifications partielles, nous nous sommes intéressés à la notion de *spécification par contrats*, dans laquelle on spécifie un composant par deux propriétés. La première, généralement appelée *assertion*, décrit l'environnement dans lequel le composant est supposé pouvoir fonctionner correctement, la seconde, appelée *guarantee*, décrivant un ensemble de comportements que le composant garantit lorsqu'il est placé dans un environnement satisfaisant son assertion. On conserve donc l'idée de séparer la propriété à prouver des hypothèses faites sur l'environnement, mais en considérant ce couple comme une *véritable spécification du comportement d'un composant*. Le chapitre 5 définit une notion de contrat pour les composants réactifs.

Cette notion de contrat a été largement étudiée pour les langages orientés objets et a connu un succès relatif. Dans ce contexte, on associe un couple *assume-guarantee* à chaque méthode. L'utilisation qui en est faite relève principalement de l'exécution défensive des programmes : on se sert du contrat comme d'un garde-fou pour identifier d'éventuels défauts d'utilisation des méthodes d'une classe. Les utilisations pour la validation formelle sont quasiment inexistantes. Cette utilisation en programmation défensive est aussi possible pour les contrats LUSTRE. Elle a été étudiée par Marc Vareille pendant un stage de magistère et implémentée dans le débogueur LUDIC développé par Fabien Gaucher dans le cadre de sa thèse [Gau03]. Nous en rappellerons les principes au paragraphe 9.3.

Nous nous sommes plus intéressés ici à l'utilisation des contrats pour la vérification. Nous avons exploré la validation des composants vis-à-vis de leur contrat : un contrat est-il implémentable ? (c'est-à-dire les contraintes *assume* et *guarantee* ne sont-elles pas contradictoires ?) ; un nœud est-il une implémentation valide d'un contrat ? etc. Ces questions sont étudiées au chapitre 9.

Par ailleurs, les contrats sont une nouvelle forme de description de comportements de composants synchrones. Nous avons voulu la comparer aux autres formes utilisées par ailleurs : les nœuds LUSTRE, les observateurs synchrones, les automates « *Lucky* » (permettant de décrire des comportements non-déterministes probabilistes), etc. Nous avons pour cela défini ces différentes formes dans un même cadre sémantique et étudié les éventuelles relations qui existent entre elles. C'est l'objet du chapitre 3.

1.3.3 Validation des programmes réguliers

Une fois que les itérateurs de tableaux ont été ajoutés au langage et qu'on a montré leur utilité dans la description et la compilation des systèmes, il s'est posé la question de l'utilisation des tableaux

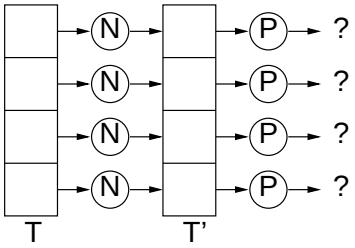


FIG. 1.4 – Une propriété simple sur une itération.

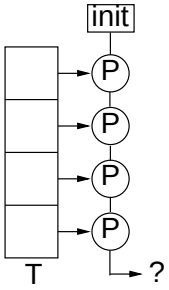


FIG. 1.5 – Cas général.

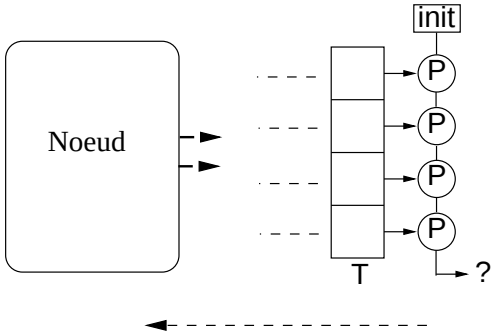


FIG. 1.6 – Propagation à un nœud.

dans la *validation*. La solution qui s'offrait à nous alors n'était pas très satisfaisante : elle consistait à expanser le code LUSTRE avec des itérateurs pour obtenir du code utilisable par les outils de validation (model-checkers, outils de test). Cette approche présente deux inconvénients majeurs. Le premier est que le code expansé est en général très gros et les outils de model-checking n'arrivent pas à le traiter. Le second est que lorsqu'on expanse les tableaux, on perd une information importante pour la compréhension du comportement du programme. Il semble pourtant intéressant de pouvoir tirer partie de cette information.

L'idée que nous avons explorée est celle de « *découper* » (slicer) les programmes selon les tableaux. Admettons par exemple qu'on calcule par une itération map un tableau T' à partir d'un tableau T tels que T'[i] = N(T[i]) (où N est un nœud LUSTRE quelconque) et que l'on souhaite vérifier que tous les éléments de T' satisfont une propriété P (voir figure 1.4). Il est inutile de prouver P pour chaque élément de T' : un seul suffit. Dans le cas général, une propriété sur un tableau est exprimée à l'aide d'une réduction (red) comme celle de la figure 1.5. La méthode que nous proposons consiste à tenter de prouver un renforcement de la propriété : on va essayer de montrer que cette propriété est un invariant de l'itération considérée. Cette méthode est décrite au chapitre 8. On peut propager ce slice à toutes les itérations utilisées pour calculer la propriété.

Lors de la propagation de cette transformation, il se peut que l'on rencontre des nœuds (voir figure 1.6). On pourrait à l'avance expanser ces nœuds, mais cette solution paraît un peu brutale puisqu'on cherche à tirer partie de la structuration des programmes et que, justement, les nœuds sont une forme de structuration. Nous avons alors étudié les possibilités d'utilisation des contrats dont nous avons parlé plus haut dans cette preuve.

Cette thèse s'inscrit dans le domaine de la spécification mais aussi de la preuve formelle des programmes. Comme nous le verrons plus loin, plusieurs méthodes ont été largement étudiées pour répondre à cette nécessité de validation des systèmes.

Le *model-checking* est parfois considéré comme une de méthode idéale car il est entièrement automatique et peut représenter une étude *exhaustive* et *automatique* du système à vérifier. Néanmoins, cette méthode est limitée par un problème d'explosion combinatoire lié à la taille des modèles manipulées.

Plusieurs approches différentes permettent de contourner ce problème d'explosion combinatoire. Les méthodes de *test* conservent un caractère automatique, mais présentent l'inconvénient d'être non exhaustives. Les *méthodes déductives*, basées sur l'utilisation de systèmes de preuve interactifs, recouvrent un caractère exhaustif, au détriment de l'automatisation. Ces méthodes présentent deux inconvénients majeurs. Le premier est la nécessité d'expertise à la fois dans les formalismes logiques qu'ils manipulent et dans leurs mécanismes propres qui sont relativement complexes à appréhender. Le second est le fait que lorsqu'on étudie des programmes venant d'un langage à sémantique de haut niveau d'abstraction (comme c'est le cas pour LUSTRE), il est difficile de garder une connexion claire entre le programme et la propriété écrits par le développeur et la preuve de la propriété elle-même.

Mais ces méthodes ne tiennent pas compte (ou très peu) de la structuration des programmes : ceux-ci sont manipulés sous une forme « brute », le plus souvent un automate unique représentant le comportement du système entier. On oublie totalement les différentes formes de structure syntaxique des programmes (notamment les structures régulières et le découpage en sous-comportements). En LUSTRE, par exemple, les nœuds sont expansés, les tableaux transformés en variables indépendantes pour chaque élément.

Le travail présenté dans cette thèse propose d'exploiter certaines formes de structuration des programmes (précisément des *spécifications locales* données sous la forme de *contrats* et des *programmes réguliers* décrits à l'aide d'*itérations*) pour faciliter la construction correcte et la validation des systèmes.

1.4 Contributions

Les contributions principales de ce travail sont :

- la définition d'itérateurs de tableaux pour le langage flot-de-donnée synchrone LUSTRE ;
- une notion de contrats pour LUSTRE ;
- un cadre sémantique unifié permettant la description des différentes formes de spécification utilisées autour de LUSTRE ;
- un outil mettant en place cette méthodologie.
- une méthodologie de construction correcte et de validation basée sur les deux éléments ci-dessus ;

1.5 Plan du document

La suite du document est organisée en 4 grandes parties.

Partie I - « LUSTRE *et modèles synchrones* » – Cette partie concerne la définition des concepts de base autour du modèle synchrone. Nous présentons au chapitre 2 le langage LUSTRE, ainsi que les différentes méthodes et outils de validation associées. Au chapitre 3, nous définissons un cadre formel dans lequel nous décrivons les différentes formes de spécification utilisées autour de LUSTRE :

nœuds, observateurs, nœuds à oracles, automates Lucky, etc. Les définitions de bases seront aussi utilisées plus loin pour décrire les contrats et les manipulations de contrats. Ces définitions ne sont pas forcément toutes utilisées dans la suite du document, mais nous avons trouvé intéressant de définir la sémantique de ces modèles de spécifications dans des notations identiques et de pouvoir les comparer.

Partie II - « *Tableaux et Contrats* » – Cette partie présente les extension que nous proposons pour le langage LUSTRE. Les itérations, présentées au chapitre 4 doivent faciliter la description des systèmes réguliers. Au chapitre 5 nous présentons les contrats *assume-guarantee* que nous proposons pour aider à la spécification des composants réactifs.

Partie III - « *Exemples* » – Les chapitre 6 et 7 introduisent deux études de cas sur lesquelles nous avons travaillé durant cette thèse. Ces exemples ont constitué la motivation principale de ce travail, tant au point de vu des itérations (notamment l'ELMU du chapitre 7) que pour les contrats. Nous mettons ici en relief l'apport des propositions faites dans la partie précédente au vu de la *description* des systèmes. Nous soulevons également le problème de l'effort de spécification supplémentaire qu'impliquent nos propositions langages.

Partie IV - « *Exploitation* » – Cette partie tend à montrer comment tirer partie des aspects langage, introduits dans la partie II, dans la *validation des systèmes*. Nous tâcherons de mettre en évidence le fait que le prix payé en terme d'effort de spécification est compensé par un gain au niveau validation grâce la mise en place d'une série de manipulations de base (à la fois sur les itérations et sur les contrats) facilitant la validation d'un système. Au chapitre 8, nous proposons une technique de transformation des itérations. Au chapitre 9, nous proposons une série de techniques pour utiliser les contrats pour la validation par composant d'un programme réactif. Le chapitre 10 reprend les études de cas de la partie IV et montre l'application concrète des techniques proposées dans les deux chapitres précédents. Enfin, le chapitre 11 présente un prototype implémentant les techniques introduites aux chapitres précédents ainsi qu'une interface graphique simple permettant de commander l'application de ces techniques.

Finalement, nous présentons les conclusions de ce travail au chapitre 12 et ses perspectives au chapitre 13.

Le lecteur remarquera qu'il n'y a pas, dans ce document, de *bibliographie* globale. Devant la variété des thèmes abordés, nous avons préféré « découper » cette bibliographie. Ainsi, les chapitres 4, 5, 8 et 9 contiennent chacun une partie *travaux connexes* qui décrit les travaux d'autres auteurs que nous avons trouvé pertinents pour chaque thème abordé.

Partie I

LUSTRE et modèles synchrones

Chapitre 2

LUSTRE et outils

Dans ce chapitre, nous présentons le langage LUSTRE *ainsi que les méthodes de compilation pour traduire un programme* LUSTRE *en du code impératif. Nous survolons rapidement les méthodes de validation associées : model-checking, preuve déductive, test et débogage.*

2.1 Introduction

Le langage LUSTRE [HCRP91] est un langage flot-de-donnée synchrone développé à Verimag depuis le milieu des années 80. La programmation flot-de-donnée (introduite dans [Kah74]) permet de décrire des programmes qui manipulent des flots de valeurs. Un des premiers exemples de langage implémentant cette approche est Lucid [AW77, WA85b].

Les langages flots de donnée *synchrones* sont nés de la nécessité rencontrée par les automaticiens de passer de la conception des systèmes analogiques à la programmation des machines séquentielles. Ayant l'habitude de manipuler des systèmes d'équations sur des flots de valeurs pour décrire le comportement des systèmes, ils trouvaient particulièrement inefficace (et source d'erreur) le fait de décrire leurs programmes de manière séquentielle.

LUSTRE et SIGNAL ont été développés dans cette idée : fournir des langages permettant de définir des flots de données par des systèmes d'équations récursifs. Ces langages sont directement traduits en code séquentiel et fournissent donc un mode de description parfait pour ce genre d'applications. Comme nous l'avons dit dans l'introduction, ils sont particulièrement adaptés à la description de systèmes dirigés par les données.

Dans la section 2.2, nous présentons la syntaxe et la sémantique du langage LUSTRE tel que défini dans [HCRP91]. Nous introduisons le type tableau, sans pour autant présenter les opérateurs introduits pour manipuler ceux-ci dans LUSTRE-V4. Ces opérateurs seront présentés au chapitre 4. Ensuite (section 2.3), nous nous intéressons à la validation des programmes synchrones : nous introduisons la notion d'observateur synchrone qui permet de décrire les propriétés sur les programmes, puis nous présentons le schéma de preuve basé sur ces observateurs. Nous terminons en présentant les différentes approches de validation utilisée autour de LUSTRE : model-checking, génération automatique de séquences de test, méthodes déductives et débogage.

2.2 LUSTRE

2.2.1 Un bref aperçu du langage

Nous présentons maintenant brièvement la syntaxe du langage et ses principaux opérateurs.

2.2.1.1 Variables, expressions

En LUSTRE, toute variable ou expression désigne un flot, c'est-à-dire une suite infinie de valeurs d'un type donné. Une variable X est la suite de valeurs :

$$x_0, x_1, \ldots, x_n, x_{n+1}, \ldots$$

où x_i est la valeur de X à l'instant i. Si c est une constante, c dénote en LUSTRE la suite infinie de c ; par exemple, la constante entière 1 dénote la suite (1,1,1,...).

Un opérateur classique OP de $\tau_1 \times \tau_2 \times \ldots \times \tau_n$ dans τ (où les τ_i et τ sont des types) opère point à point sur les suites infinies de ces types. Par exemple, l'expression $X+Y$ est définie par :

si $X = (x_1, x_2, \ldots, x_i, \ldots)$
et $Y = (y_1, y_2, \ldots, y_i, \ldots)$
alors $X+Y = (x_1+y_1, x_2+y_2, \ldots, x_i+y_i, \ldots)$

On définit ainsi :

- les opérateurs booléens : or, and, not, le ou exclusif xor et l'implication => ;
- les opérateurs arithmétiques habituels : +, -,*, la division réelle /, la division entière div et le reste de la division entière mod ;
- les opérateurs de comparaison sur les entiers et les réels : =, >=, >, <= et la différence <> (l'égalité = peut aussi être utilisé sur les booléens) ;
- la structure conditionnelle : l'expression if C then E1 else E2 (où C est une expression booléenne et E1 et E2 sont 2 expressions de même type) décrit le flot X tel que $\forall n$. if $C_n$then $X_n = E1_n$else $X_n = E2_n$.

2.2.1.2 Opérateurs sur les suites

Il existe en LUSTRE un opérateur pour faire référence au passé d'une expression. C'est l'opérateur pre, qui est défini comme suit :

$$\forall n > 0, \mathsf{pre}(X)_n = X_{n-1}$$

Autrement dit, si $X = (x_1, x_2, \ldots, x_i, \ldots)$, alors pre(X) = $(\mathsf{nil}, x_1, x_2, \ldots, x_i, \ldots)$, où nil dénote une valeur indéfinie. L'opérateur d'initialisation $\rightarrow$ permet de définir la valeur d'une suite à l'instant initial. Si $X = (x_1, x_2, \ldots, x_i, \ldots)$ et $Y = (y_1, y_2, \ldots, y_i, \ldots)$ sont 2 suites de même type, alors :

$$X \rightarrow Y = (x_1, y_2, \ldots, y_i).$$

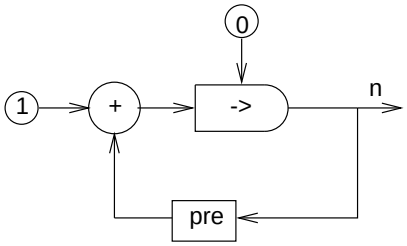


FIG. 2.1 – Représentation graphique du compteur.

2.2.1.3 Équations et nœuds

Une variable "id" d'un programme LUSTRE peut être définie à l'aide d'une équation de la forme : $id = expression$. On peut écrire des équations récurrentes comme n = 0 → pre(n) + 1, qui définit de proche en proche n comme la suite des entiers naturels :

$$\begin{aligned} \text{pre(n)} &= (nil, 0, 1, 2, \ldots) \\ \text{n} &= (0, 1, 2, 3, \ldots) \end{aligned}$$

Un système d'équations LUSTRE peut être facilement représenté graphiquement par un réseau d'opérateurs. La figure 2.1 montre le réseau correspondant à l'équation n = 0 → pre(n) + 1.

Les équations LUSTRE peuvent être regroupées à l'intérieur de programmes appelés *nœuds*. Un nœud possède des entrées, des sorties, des variables locales et des équations définissant ses sorties et variables locales. Les nœuds peuvent être, comme les opérateurs, organisés en réseaux, grâce aux propriétés de compositionnalité des programmes synchrones.

EXEMPLE 1 — Le programme de la figure 2.2 compte le nombre de signaux vrais qu'il reçoit en entrée. A début de l'exécution, le résultat c vaut 1 si un premier signal est détecté, 0 sinon. Ensuite, à chaque instant, le nœud Accumulateur ajoute à la valeur précédente de c (identifiée par l'expression pre(c)), 1 si la nouvelle entrée est vraie, 0 sinon, calculant ainsi la nouvelle valeur de c.

— FIN DE L'EXEMPLE 1

```
node Accumulateur (e : bool) returns (c : int);
let
   c = (if e then 1 else 0)
       -> (pre(c) + (if e then 1 else 0));
tel
```

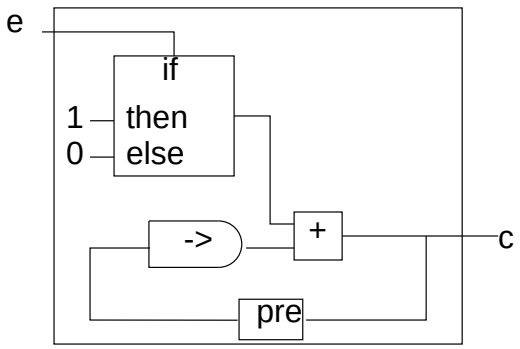


FIG. 2.2 – Le programme Accumulateur.

Remarque 1 LUSTRE *étant un langage* déclaratif, *l'ordre des équations à l'intérieur des nœuds n'a pas d'importance.*

E	=	(	e_1	e_2	e_3	e_4	e_5	...)
C	=	(	$true$	$false$	$true$	$true$	$false$	...)
X = E when C	=	(	$x_1 = e_1$		$x_2 = e_3$	$x_3 = e_4$		...)

FIG. 2.3 – L'opérateur when.

E	=	(	e_1	e_2	e_3	e_4	e_5	...)
C	=	(	$true$	$false$	$true$	$true$	$false$	...)
X = E when C	=	(	e_1		e_3	e_4		...)
Y = current X	=	(	e_1	e_1	e_3	e_4	e_4	...)

FIG. 2.4 – L'opérateur current.

2.2.1.4 Horloges et flots

Pour l'instant, nous n'avons utilisé qu'une seule notion de temps dans les programmes LUSTRE : celle induite par la suite des valeurs (qui définit une horloge dite *globale*, dénotée par la constante true). Pour certaines applications, il est cependant intéressant de faire évoluer des sous-systèmes à des rythmes différents. Pour cela, il existe un opérateur d'*échantillonnage* when et un opérateur de projection (ou de *sur-échantillonnage*) current.

Échantillonnage avec when – Si E est une expression quelconque et C une expression booléenne, E when C dénote la suite de valeurs extraite de E quand C est vraie. Quand l'horloge C est fausse, la suite E when C n'a pas de valeur (figure 2.3).

Sur-échantillonnage avec current – Si X dénote un flot sur l'horloge C, current X dénote un flot sur l'horloge H de C. A chaque instant de l'horloge H, si C est vrai, alors current X porte la même valeur que X, sinon il porte la valeur prise par X au dernier instant où C était vraie (voir figure 2.4).

Si aucune contrainte d'horloge n'est associée (par l'opérateur when) à une variable donnée, alors c'est l'horloge globale qui est utilisée.

2.2.1.5 Types

En LUSTRE, il existe 3 types primaires :
- le type entier, noté int ;
- le type booléen, noté bool ;
- le type réel, noté real.

Il est par ailleurs possible de structurer des valeurs sous la forme de tableaux : si τ est un type quelconque, alors τ^n (où n est une constante connue statiquement) est le type tableau d'éléments de type τ. La taille des tableaux doit nécessairement être connue statiquement, ceci pour éviter toute erreur d'allocation dynamique de mémoire (afin de satisfaire les contraintes de criticité des domaines d'application de LUSTRE)

On peut aussi nommer des types dans le corps d'un fichier lustre. Par exemple, l'instruction suivante nomme par T le type « tableaux d'éléments de types int de taille 10 » :

type T = int^10;

Nous reviendrons plus loin sur l'utilisation des variables de types tableaux.

2.2.1.6 Structures à champs nommés

Nous souhaitons introduire à présent un nouvel élément du langage que nous utilisons plus loin dans les études de cas de cette thèse. Il s'agit des *structures à champs nommés*. Ces structures ne font pas partie du langage dans sa version V4, mais ont été introduites pour la version V6.

Ces structures sont un moyen de rassembler des données de types différents (à la différence des tableaux dont tous les éléments sont de même type). Par exemple, on va pouvoir définir une variable struct à deux champs a de type int et b de type $bool$:

```
var struct : { a : int; b : bool};
```

On peut aussi nommer un type structuré :

```
type TStruct = { a : int; b : bool };
```

et définir des variables de ce type :

```
var struct2 : TStruct;
```

La définition de variables à type structuré peut se faire de deux manières. Soit on associe directement deux variables de type identique :

```
struct2 = struct;
```

Soit on définit chaque champ spécifiquement :

```
struct2 = { a = 0 -> loc + 1; b = true};
```

Pour ce qui est de l'utilisation des champs d'une telle variable on peut sélectionner un champ en utilisant l'opérateur « . » :

```
..... = if struct2.b
        then struct.a
        else if struct.b
             then 0 -> pre(struct2.a)
             else 0
```

2.2.1.7 Exemple de programme

Dans cette partie, nous allons programmer un dispositif de surveillance de délais de réponse, tiré de [HCRP91]. Il nous est utile pour illustrer différents aspects présentés dans la suite du chapitre. Ce programme reçoit en entrée : armer, desarmer (qui sont 2 commandes du système) et date_limite (qui est un événement). Les événements et commandes sont représentés par des variables booléennes. La valeur true dénote la présence d'un événement ou l'exécution d'une commande. La sortie alarme doit être activée à chaque fois qu'une date_limite est atteinte et que la dernière commande reçue est armer.

```
alarme = date_limite and est_arme ;
```

```
node chien_de_garde (armer, desarmer, date_limite : bool)
returns (alarme : bool);
var est_arme : bool;
let
   alarme = date_limite and est_arme;
   est_arme = armer -> if armer
                       then true
                       else if desarmer
                            then false
                            else pre(est_arme);
tel
```

FIG. 2.5 – Le programme du chien de garde

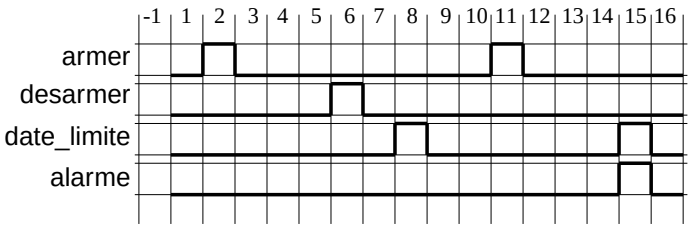


FIG. 2.6 – chronogramme du chien de garde

Reste à définir est_arme qui devient vrai à chaque fois que armer est vrai et reste vrai jusqu'à ce que desarmer soit vrai :

```
est_arme = armer -> if armer
                    then true
                    else if desarmer
                         then false
                         else pre(est_arme);
```

Le programme complet du chien de garde est donné à la figure 2.5. L'exécution d'un tel programme peut être observé sur un chronogramme (voir figure 2.6).

2.2.2 Méthode de compilation

Dans un premier temps, nous abordons brièvement les vérifications statiques qui sont effectuées lors de la compilation d'un programme LUSTRE. Ensuite, nous montrons comment du code impératif est généré à partir du programme LUSTRE.

2.2.2.1 Vérifications statiques

Comme il a déjà été souligné plus haut, LUSTRE est en général utilisé pour programmer des systèmes qualifiés de « critiques », dans lesquels on veut impérativement éviter toute erreur dynamique

x	1	2	3	4	5
b	true	false	true	false	true
x when b	1		3		5

FIG. 2.7 – x + (x when) b est incorrect du point de vue des horloges.

telle que dépassement de la mémoire allouée pour le programme, erreurs dues aux variables non-initialisées, utilisation d'un pointeur erroné.

Nous devons donc nous assurer *statiquement* que le programme ne provoquera pas de telles erreurs d'exécution. Cette vérification concerne les points suivants :

- *Vérification des définitions* : Toute variable locale ou sortie du programme doit être définie par *une et une seule équation* ;
- *Absence d'appels récursifs de nœud* : un nœud ne peut pas s'appeler lui-même (y compris par l'intermédiaire d'un autre nœud), sauf dans le cas très précis de la *récursivité statique*, dans laquelle l'arrêt de la récursivité peut être assuré statiquement (voir 4.1.5). En effet, si l'arrêt de la récursion n'est pas assuré, on peut potentiellement écrire des programmes qui prennent un temps non borné pour calculer une valeur de leur flot de sortie ;
- *Absence d'expressions non-initialisées* : on veut éviter les nil comme dans l'expression pre(X) sans utilisation de -> ;
- *Absence de définitions cycliques* : tout cycle dans un réseau (d'opérateurs ou de nœuds) doit contenir au moins *un* pre. Une équation telle que X = 3*X + 1 est donc interdite ;
- *Définition statique des tailles des tableaux* : aucune allocation dynamique de mémoire n'est autorisée. Les tableaux doivent donc être de taille connue à la compilation. On vérifie aussi que les accès aux éléments des tableaux se font par indice statique ;
- et enfin le *calcul d'horloge* (détaillé ci-dessous).

Calcul d'horloge – Soient les équations suivantes (leur comportement est donné à la figure 2.7) :

```
b = true -> not pre b;
y = x + (x when b);
```

x et x when b n'ont pas la même horloge : x a une valeur à tout instant de l'horloge globale, alors que x when b a une valeur seulement quand b est vrai (c'est-à-dire à un instant global sur 2). Or, on ne sait pas ajouter un entier à une valeur inexistante. Il en résulte une incohérence dans le calcul de l'expression x + (x when b).

Le *calcul d'horloge* consiste donc à associer une horloge à chaque expression d'un programme et à vérifier la contrainte suivante : *n'importe quel opérateur de plus de 1 argument est appliqué à des opérandes qui partagent la même horloge*. L'opération décrite ci-dessus (x + (x when b)) est donc en ce sens interdite en LUSTRE.

Remarquons que les restrictions imposées par le calcul d'horloge peuvent amener à refuser des programmes qui n'aboutiraient pas forcément à des erreurs dynamiques. Si la vérification statique accepte le programme, alors on est sûr que le programme sera sans erreur dynamique. Si le programme est rejeté, il se peut qu'il y ait des erreurs dynamiques.

```
node EDGE (X : bool) returns (Y : bool);
let
  Y = false -> X and not pre(X);
tel
```

```
node FALLING_EDGE (X : bool)
returns (Y : bool);
let
  Y = EDGE(not X);
tel
```

```
node FALLING_EDGE_ec (X : bool) returns (Y : bool);
let
  Y = false -> (not X) and not pre(not X);
tel
```

FIG. 2.8 – Les nœuds EDGE, FALLING_EDGE et FALLING_EDGE_ec.

2.2.2.2 Processus de compilation

Expansion du code LUSTRE – Un programme LUSTRE est tout d'abord compilé en un programme EC (pour *expanded code*). Un programme EC est en fait un programme LUSTRE dans lequel tous les appels de nœuds ont été expansés.

EXEMPLE 2 — Considérons le nœud EDGE décrit à la figure 2.8 et qui détecte les fronts montants d'un flot booléen X. Ce nœud peut-être utilisé dans un nœud FALLING_EDGE qui détecte les fronts descendants d'un flot booléen. Le compilateur lus2ec génère le programme ec FALLING_EDGE_ec.
— FIN DE L'EXEMPLE 2

Remarque 2 *Cette phase d'expansion n'est pas indispensable. Sous certaines contraintes, il est en effet possible de compiler séparément les différents nœuds d'un programme* LUSTRE*. Par exemple, le compilateur de l'outil* SCADE *choisit d'interdire les cycles de définition instantanés (sans* pre*), même si ces cycles contiennent un* pre *à l'intérieur d'un nœud appelé. L'outil donne ensuite le choix au programmeur de compiler séparément les nœuds du programme. Dans le schéma de compilation décrit ici, on fait le choix d'une compilation monolithique : on autorise des cycles de définitions instantanés (à condition que les cycles soient brisés une fois que tous les appels de nœuds ont été expansés), et on expanse tous les nœuds.*

Séquentialisation en boucle simple – Le programme séquentiel que nous voulons obtenir est de la forme dont nous avons parlé plus haut : une boucle infinie (while(true)) représente le *passage du temps*. A chaque instant, le programme lit ses entrées, calcule ses sorties et met à jour ses mémoires. Nous rappelons maintenant le principe de génération de code pour les différents opérateurs du langage:

- Une mémoire est nécessaire pour chaque instance de l'opérateur pre : pour chaque variable (ou expression) v à laquelle pre est appliqué, nous déclarons dans le programme séquentiel 2 variables : v qui représente la valeur de v à l'instant courant et pre_v qui représente la valeur de v à l'instant précédent ;
- L'opérateur d'initialisation -> est traité de la façon suivante : on génère globalement une variable booléenne init qui est vraie seulement au premier instant de l'exécution du système. Le calcul des sorties et variables locales est alors conditionné par la valeur de cette variable ;
- Une expression conditionnelle de type V = if C then E1 else E2 est traduite en un bloc conditionnel : if(C){V = E1}else{V = E2} ;

```
init = true;
while(true){
    read(armer);                                    // lecture des entrées
    read(desarmer);
    read(date_limite);
    if(init){                                       // calcul des sorties
      est_arme = armer;                             // instant initial
      alarme = date_limite and est_arme;
      init = false;
    }else{
      if(armer){                                    // cas général
        est_arme = true
      }else if(desarmer){
        est_arme = false
      }else {est_arme = pre_est_armer;}
    }
    pre_est_arme = est_arme;                        // Mise à jour des mémoires
}
```

FIG. 2.9 – Code séquentiel généré pour le programme chien_de_garde.

- Toutes les autres expressions (arithmétiques/booléennes) sont directement traduites dans leur équivalent impératif.

La figure 2.9 donne le code obtenu pour le programme du chien_de_garde.

La chaîne de compilation LUSTRE – Le compilateur académique développé à Verimag traite les programmes en 2 étapes. La première, effectuée par le compilateur lus2ec, consiste à traduire le code source LUSTRE en un programme au format EC qui ne contient qu'un nœud. Plusieurs outils utilisent ce format intermédiaire directement (l'interpréteur Luciole, le model-checker lesar) ou comme point de départ (format d'entrée du traducteur ec2nbac qui génère un programme au format ba utilisable par l'outil de vérification nbac). Le graphe de compatibilité de ces différents formats est donné à la figure 2.10.

La seconde étape de compilation est effectuée par le compilateur ec2c et consiste à générer le code séquentialisé en langage C. Le code généré ne contient pas la boucle de temps while(true){...} mais seulement une fonction d'initialisation et une fonction step qui réalise une étape de calcul (un instant de l'exécution).

2.3 Validation

Nous nous intéressons maintenant à la validation des programmes décrits en LUSTRE. Nous l'avons vu plus haut, les différentes méthodes peuvent être classées selon leur caractère exhaustif ou automatique. Nous présentons tout d'abord la notion d'observateur qui permet de décrire les propriétés de sûreté dans le langage lui-même. Nous rappelons le principe du model-checking basée sur les observateurs LUSTRE [HLR93]. Nous présentons ensuite une méthode déductive [DC00] utilisant l'outil de validation PVS. Au paragraphe 2.3.4, nous présentons rapidement les méthodes de test utilisées pour LUSTRE [RWNH98]. Enfin, nous présentons le débogueur LUDIC [Gau03].

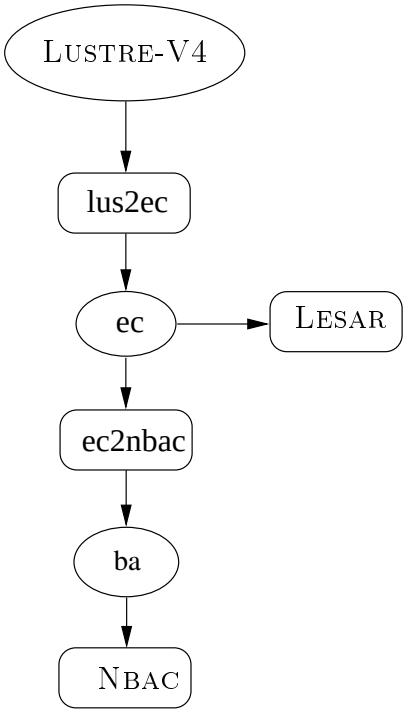


FIG. 2.10 – Les différentes formats et les compatibilités d'outils.

2.3.1 Propriétés et observateurs

La validation des systèmes s'intéresse de manière générale à la vérification de propriétés que l'on peut classer en deux catégories :

- d'un côté les propriétés de *sûreté* qui expriment que quelque chose (en général quelque chose de « *mauvais* ») n'arrivera jamais ;
- de l'autre les propriétés de *vivacité* qui expriment que quelque chose arrivera *à un moment* de l'exécution du système.

Nous nous intéressons à la vérification des propriétés de sûreté sur des programmes LUSTRE car ce sont les plus fréquemment rencontrées dans le domaine des systèmes critiques. Nous décrivons ces propriétés à l'aide d'observateurs [HLR93]. Ce sont des nœuds LUSTRE qui prennent en entrée les entrées/sorties du programme à vérifier et renvoient comme unique sortie une valeur booléenne représentant la valeur de vérité de la propriété à prouver.

Intuitivement, un observateur décrit un ensemble de comportements des entrées/sorties du programme. On peut voir cet ensemble de comportements comme une relation indéterministe entre les entrées et les sorties. Nous y reviendrons plus tard. Les observateurs permettent aussi de décrire des hypothèses faites sur l'environnement.

EXEMPLE 3 — Considérons à nouveau les programmes EDGE et FALLING_EDGE. Nous voulons à présent exprimer le fait qu'il ne peut y avoir plus de fronts descendants que de fronts montants sur un même flot booléen X. Pour cela, il nous faut compter les fronts grâce au nœud Accumulateur de la figure 2.2. Nous exprimons la propriété à l'aide de l'observateur prop donné à la figure 2.11.

— **FIN DE L'EXEMPLE 3**

```
node prop (X : bool) returns (ok : bool);
var edge,fedge : bool
    cedge,cfedge : int;
let
   edge = EDGE(X);
   fedge = FALLING_EDGE(X);
   cedge = Accumulateur(edge);
   cfedge = Accumulateur(fedge);
   ok = cfedge <= cedge;
tel
```

FIG. 2.11 – Observateur pour la propriété « Un flot booléen n'a pas plus de fronts descendants que de fronts montants ».

2.3.2 Vérification par modèles

2.3.2.1 Schéma de preuve

Nous voulons prouver qu'un programme P satisfait une propriété de sûreté φ sachant certaines hypothèses sur son environnement, représentée par une *assertion* Env.

Comme indiqué plus haut, le schéma de preuve des programmes LUSTRE est basé sur la notion d'observateur. Nous utilisons deux observateurs. Le premier, qui sert à modéliser φ, reçoit les entrées et les sorties du programme P et renvoie une sortie booléenne ok vraie tant que la propriété est satisfaite. Le second, qui modélise Env, reçoit les entrées et les sorties de P et renvoie une sortie booléenne relevant, vraie à chaque instant où l'assertion est satisfaite.

Afin de prouver que P satisfait φ sous l'hypothèse Env, le schéma de validation décrit à la figure 2.12 est fourni à un outil de validation par model-checking (Lesar [HLR93], nBac [Jea00]). On dit que P satisfait φ sous l'hypothèse Env si et seulement si, pour toute séquence d'entrées possible soit ok est toujours vrai (la propriété est satisfaite), soit relevant est fausse à un instant donné (l'hypothèse sur l'environnement est violée). Prouver que ok est toujours vrai revient en fait à prouver:

$$\forall \text{ entrées, sorties. } [Env(\text{entrées}, \text{sorties}) \quad \wedge \quad \text{sorties} \; = \; P(\text{entrées}, \text{sorties})] \Rightarrow \varphi(\text{entrées}, \text{sorties}),$$

qui signifie que pour toutes séquences de valeurs d'entrées qui satisfont l'assertion Env, le programme produit des séquences de valeurs de sorties qui satisfont la propriété φ.

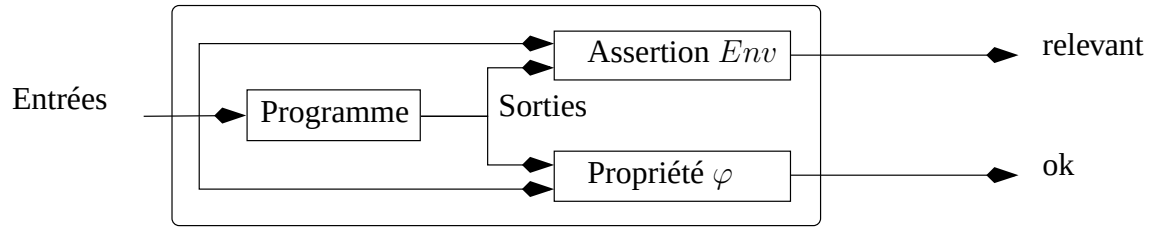


FIG. 2.12 – Schéma de preuve d'un programme LUSTRE.

Dans ce schéma, les hypothèses sur le programme à prouver peuvent aussi parler de l'histoire des sorties : on peut vouloir exprimer des hypothèses de la forme « la nouvelle entrée n'est jamais plus grande que la valeur de sortie précédente ».

Assertions causales – Avant d'évaluer la validité d'une propriété sur un programme, on commence par étudier la validité de *l'assertion associée*. La *causalité* d'une assertion représente le fait que cette assertion décrit un comportement non-bloquant. Une assertion est dite non-causale si elle est susceptible de créer, dans le comportement du système, un ou des états-puits. Par exemple, l'assertion suivante(tirée de la thèse de Pascal Raymond [Ray91]) est non-causale :

```
assert true-> not(pre(a) and pre(false -> pre(a))).
```

Intuitivement, une assertion est *causale* si tout état accessible depuis l'état initial sans jamais voler l'assertion, peut être quitté par une transition qui ne viole pas l'assertion. [Rat92] a proposé une caractérisation de ces assertions non-causales qui sont détectées à la compilation.

2.3.2.2 Model-checking

L'outil Lesar[1] est un model-checker pour la vérification de programme LUSTRE : il explore un modèle fini (un automate) du programme afin de déterminer si un état viole la propriété à prouver. Le modèle manipulé par Lesar est une abstraction représentant une sur-approximation des exécutions possibles du programme. L'abstraction faite sur le programme « *réel* » est conservative : si on arrive à prouver une propriété sur la première, alors la propriété est aussi satisfaite par le programme. Si la vérification échoue sur le modèle, le résultat est inconclusif : soit la propriété n'est effectivement pas satisfaite par le programme réel ; soit la propriété est trop complexe pour l'outil.

Précisément, la partie booléenne du programme est complètement reflétée dans le modèle, mais tout le reste (à savoir variables numériques, types externes, ...) est abstrait. Le processus de vérification se découpe en trois étapes principales :

- Analyse statique : cette partie comprend notamment une analyse de dépendances, des minimisations syntaxiques et d'autres optimisations effectuées sur le code source ;
- Construction de BDDs (diagrammes de décision binaire) : la partie booléenne du programme est transformée en un ensemble de fonctions logiques représentées par des BDDs ;
- Exploration du modèle. Plusieurs algorithmes peuvent être utilisés (algorithme énumératif, symbolique avant, symbolique arrière).

2.3.2.3 Interprétation abstraite

NBac [Jea00] est un outil permettant l'analyse de propriétés numériques, construit sur des méthodes d'interprétation abstraite [CC77].

Un programme est représenté par une combinaison d'un BDD (représentant les contraintes sur les variables booléennes du programme) et d'un polyèdre convexe (représentant une abstraction des valeurs des variables numériques du programme). Les analyses effectuées par NBac sur cette représentation des programmes sont de différentes formes :

- Analyse d'atteignabilité à partir d'un ensemble d'états initiaux, qui permet de calculer les invariants satisfaits par le système ;
- Analyse de co-atteignabilité à partir d'un ensemble d'états initiaux qui permet de calculer des ensembles d'états qui peuvent mener à un état final ;
- Une combinaison de ces deux analyses, qui permet de vérifier des propriétés de sûreté ou de calculer une sur-approximation des ensembles d'états appartenant à une exécution démarrant à un état initial et menant à un état final donné.

[1] Ce paragraphe est directement inspiré de la documentation LUSTRE [Hal93b] que le lecteur est invité à lire pour plus de précision.

2.3.3 Méthodes déductives

Une autre approche consiste à utiliser des méthodes déductives. Dans sa thèse [DC00], Cécile Dumas-Canovas a étudié la possibilité d'utiliser l'outil PVS pour aider à vérifier des propriétés de programmes LUSTRE. Une première solution consistait à encoder la sémantique du langage en PVS et laisser au démonstrateur PVS tout le travail.

Après une présentation succincte de PVS au paragraphe 2.3.3.1, nous rappelons dans 2.3.3.2 une petite partie du travail de C. Dumas-Canovas en donnant la sémantique de LUSTRE en PVS. Durant cette thèse, nous avons tenté d'adopter cette approche en étendant la sémantique aux itérateurs de tableaux. La section 8.4 présente ce travail qui a été abandonné pour les raisons de difficultés de manipulation du langage de PVS alors qu'une grande partie du travail peut-être déchargée à un outil spécifique à LUSTRE afin de n'utiliser un outil comme PVS que pour traiter des sous-buts minimaux (déjà souligné dans [DC00]).

Nous présentons maintenant brièvement le langage de spécification de PVS ainsi qu'un encodage possible de la sémantique de lustre en PVS. Cet encodage a déjà est présenté en détails dans [DC00]. Nous y revenons afin de faciliter la présentation de l'extension pour les itérateurs de tableaux faite à la section 8.4.

2.3.3.1 PVS

PVS [ORS92][2] est un environnement dédié au développement et à l'analyse des spécifications formelles. Il est construit autour d'un système de vérification déductive qui manipule des prédicats de la logique d'ordre supérieur. Dans le langage de spécification de PVS, on manipule des types de base à partir desquels on peut construire de nouveaux types tels que fonctions, ensembles, n-uplets, listes, etc. On peut décrire des spécifications en définissant des *théories* qui contiennent des définitions de types, des axiomes sur ces types. On définit ensuite les *théorèmes* que l'ont cherche à prouver sur ces théories.

Le prouveur de PVS fournit un ensemble de règles d'inférence qu'on applique de manière interactive dans le cadre du calcul des séquents. On peut aussi définir des stratégies de preuve qui sont utiles pour automatiser certaines parties de preuve (voir des preuves entières).

EXEMPLE 4 — la figure 2.13 montre une théorie PVS décrivant le comportement d'une pile . Une pile est un élément de type stack. Le type des éléments de la pile est laissé abstrait (notation [t: TYPE+]). Il existe une pile particulière notée empty qui ne contient aucun élément. La théorie PVS décrit ensuite de manière dénotationnelle les manipulations basiques possibles sur une pile:

- l'empilement (fonction push) permet de rajouter un élément de type t au somme d'une pile. La pile ainsi obtenue est garantie non-vide ;
- l'extraction de la fin de la pile (fonction pop) construit une pile (éventuellement vide) à partir d'une pile non-vide en supprimant l'élément de sommet ;
- la sélection du sommet de pile (fonction top) extrait le sommet d'une pile non-vide.

La théorie comprend ensuite plusieurs axiomes. Le premier stipule que si s est une pile non-vide le fait de dépiler et rempiler le sommet de s conserve s. Un second exprime que le fait d'empiler un élément sur une pile ne modifie pas le reste de la pile, le dernier exprime que l'empilement ne modifie pas l'élément empilé.

Ces axiomes forment la description élémentaire d'une pile. A partir de là, on va pouvoir prouver un certain nombre de théorèmes sur une pile comme celui décrit à la fin de la théorie. Par ce théorème,

[2] Les documentations concernant les différents aspects du système PVS peuvent être téléchargées sur http://pvs.csl.sri.com/documentation.shtml#pvs-bibliography

```
stacks [t: TYPE+] : THEORY
   BEGIN

   stack : TYPE+
   s : VAR stack
   empty : stack
   nonemptystack?(s) : bool = s/= empty

   push : [t,stack -> (nonemptystack?)]
   pop : [(nonemptystack?) -> stack]
   top : [(nonemptystack?) -> t]

   x,y : VAR t

   push_top_pop : AXIOM
   nonemptystack?(s) IMPLIES push(top(s),pop(s)) = s

   pop_push : AXIOM pop(push(x, s)) = s

   top_push : AXIOM top(push(x, s)) = x

   pop2push2= THEOREM pop(pop(push(x, push(y, s)))) = s

END stacks
```

FIG. 2.13 – Une théorie PVS d'une pile.

on veut montrer le fait qu'empiler 2 éléments sur une pile s puis de les dépiler tous les deux restaure la pile s originale. Dans ce cas, la preuve se fait simplement par substitutions consécutives des imbrications pop(push) grâce à l'axiome pop_push. A partir de cet axiome, on déduit facilement que pop(pop(push(x, push(y, s)))) = pop(push(y,s)). En appliquant à nouveau le même axiome, on obtient la propriété désirée : pop(pop(push(x, push(y, s)))) = s.

— FIN DE L'EXEMPLE 4

2.3.3.2 Encodage de la sémantique de LUSTRE en PVS

Dans sa thèse, C. Dumas-Canovas a proposé un encodage de la sémantique de LUSTRE basé sur quelques théories PVS. On commence par coder les flots LUSTRE par des sequences PVS. La première théorie (lustre donnée à la figure 2.14) décrit les opérateurs temporels (->, noté $\sim$ et pre) du langage, les opérateurs sur les horloges (when et current) ainsi que la conditionnelle. Pour définir les opérateurs current et when, on utilise deux fonctions particulières qui s'appellent count et index.

count – Cette fonction permet de compter le nombre de valeurs « vrai » d'un flot booléen depuis l'instant initial jusqu'à l'instant courant. $count(c)(n)$ vaut donc :

$$count(c)(n) = \left\{ \begin{array}{ll} 0 & \text{si } n = 0 \\ count(c)(n-1) & \text{sinon} \end{array} \right\} + \left\{ \begin{array}{ll} 1 & \text{si c(n) est vrai} \\ 0 & \text{sinon} \end{array} \right.$$

La $n^{\text{ème}}$ valeur du flot current(v,y,c) vaut alors la $count(c)(n) - 1^{\text{ème}}$ valeur de y ou bien v si $count(c)(n-1)$ est nul.

index – La fonction $index(c)(k)$ donne l'indice de la $k^{\text{ème}}$ valeur vrai de c. La $n^{\text{ème}}$ valeur du flot x when c vaut alors la $index(c)(n+1)$. La définition de $index$ utilise une fonction récursive $index\_rec(c)(k)(n)$ qui donne le premier indice i à partir de n tel que $count(c)(i) = k$. Nous avons alors $index(c)(k) = index\_rec(c)(k)(k-1)$. $index\_rec$ est définie par :

$$index\_rec(c)(k)(n) = \begin{cases} n & \text{si } count(c)(n) = k \\ index\_rec(c)(k)(n+1) & \text{sinon} \end{cases}$$

Trois théories supplémentaires décrivent les opérations arithmétiques réelles (lustre_reel, figure 2.16) ou entières (lustre_int, figure 2.15) et booléennes (lustre_bool, figure 2.17) classiques sur les flots. Enfin, la théorie lustre_top donnée à la figure 2.18 décrit l'invariance des calculs représentant un nœud lustre au travers d'un opérateur A.

Chaque nœud LUSTRE est représenté par une fonction typée par ses équations et est décrit comme une théorie PVS contenant 3 éléments :

- Un type décrit l'ensemble des comportements des variables de sorties en fonctions des variables d'entrées tels que défini par les équations du nœud LUSTRE ;
- Le nœud LUSTRE lui-même est un élément du type précédent ;
- Un axiome codant la réactivité du nœud LUSTRE, stipulant donc que pour toute séquence d'entrées, il existe toujours une valeur des variables de sorties du nœud considéré.

EXEMPLE 5 — On trouvera à la figure 2.19 un nœud LUSTRE calculant la suite de fibonacci : la valeur de f à l'instant t est égale à la somme de sa valeur aux instants $t-1$ et $t-2$. La figure 2.20 donne une traduction sous forme de théorie PVS du nœud fibo. Cette théorie contient le *type* (fibo_type) des valeurs calculées par le nœud fibo, une instance fibo_node de ce type (qui représente le nœud lui-même) et un axiome fibo_axiom qui signifie que pour toute séquence d'entrées, il existe toujours une séquence de sorties.

Le théorème positive exprime que si les entrées sont des flots de valeurs toujours positives alors la suite de fibonacci calculée est toujours positive.

— FIN DE L'EXEMPLE 5

La première technique étudiée dans C. Dumas-Canovas pour valider des programmes LUSTRE consiste à traduire le programme ainsi que la propriété à prouver en PVS (comme dans l'exemple ci-dessus) et à utiliser les techniques d'induction générale sur les flots et de manipulation logique du prouveur interactif de PVS. Cependant, aucune stratégie n'a été proposée pour la preuve de programmes avec horloges. D'autre part, le retour au source est rendu peu pratique par l'introduction des fonctions count et index qui éloignent le programmeur du formalisme initial (LUSTRE).

Le travail de Cécile Dumas-Canovas (aujourd'hui poursuivi par Jan Mikac à Verimag) s'est donc orienté vers la création d'un outil propre à LUSTRE permettant la simplification du programme, l'analyse de la propriété, et générant des *obligations de preuve* PVS. Ces obligations sont très simples et ne nécessitent pas, en général, d'intervention humaine (elles peuvent être vérifiées par les procédures de décision standard de PVS). Ce Générateur d'Obligations de Preuve pour LUStre (baptisé GLOUPS[3])

[3]une nouvelle version de GLOUPS, développée par Jan Mikac dans le cadre de sa thèse à Verimag est disponible en ligne à l'adresse http://www-verimag.imag.fr/∼mikac/Gloups/Gloups-index.html

```
lustre [T : TYPE+] : THEORY
   BEGIN
   IMPORTING horloge

   nil : T
   const(v : T)(n : nat)  : T = v CONVERSION const

   pre(x : sequence[T])(n : nat) : T = IF n=0 THEN nil ELSE x(n-1) ENDIF;
   ~(x,y : sequence[T])(n : nat) : T = IF n=0 THEN x(0) ELSE y(n) ENDIF;

   when(x : sequence[T], c : (infinie?))(n : nat) : T = x(index(c)(n+1))

   current(v : T, x : sequence[T], c : sequence[bool])(n : nat) : T =
           LET nb = count(c)(n) IN if nb = 0 THEN v ELSE x(nb-1) ENDIF

   ifl(c : sequence[bool], x,y : sequence[T])(n : nat) : T =
     IF c(n) THEN x(n) ELSE y(n) ENDIF;

   ==(x,y : sequence[T])(n : nat) : bool = x(n) = y(n)

END lustre
```

FIG. 2.14 – Définition des opérateurs LUSTRE.

```
lustre_entier : THEORY

   BEGIN

   +(x,y : sequence[int])(n : nat) : int = x(n) + y(n);
   -(x,y : sequence[int])(n : nat) : int = x(n) - y(n);
   *(x,y : sequence[int])(n : nat) : int = x(n) * y(n);
   -(x : sequence[int])(n : nat) : int = -x(n);
   <(x,y : sequence[int])(n : nat) : bool = x(n) < y(n);
   <=(x,y : sequence[int])(n : nat) : bool = x(n) <= y(n);
   >(x,y : sequence[int])(n : nat) : bool = x(n) > y(n);
   >=(x,y : sequence[int])(n : nat) : bool = x(n) >= y(n)

END lustre_entier
```

FIG. 2.15 – Définition des opérateurs arithmétiques entiers.

```
lustre_reel : THEORY
   BEGIN

   +(x,y : sequence[real])(n : nat) : real = x(n) + y(n);
   -(x,y : sequence[real])(n : nat) : real = x(n) - y(n);
   *(x,y : sequence[real])(n : nat) : real = x(n) * y(n);
   /(x : sequence[real], y : sequence[real])(n : nat) : real = x(n) / y(n);
   -(x : sequence[real])(n : nat) : real = -x(n);
   <(x,y : sequence[real])(n : nat) : bool = x(n) < y(n);
   <=(x,y : sequence[real])(n : nat) : bool = x(n) <= y(n);
   >(x,y : sequence[real])(n : nat) : bool = x(n) > y(n);
   >=(x,y : sequence[real])(n : nat) : bool = x(n) >= y(n)

   END lustre_reel
```

FIG. 2.16 – Définition des opérateurs arithmétiques réels.

```
lustre_bool : THEORY
   BEGIN

   NOT(x : sequence[bool])(n : nat) : bool = NOT x(n);
   AND(x,y : sequence[bool])(n : nat) : bool = x(n) AND y(n);
   OR(x,y : sequence[bool])(n : nat) : bool = x(n) OR y(n);
   =>(x,y : sequence[bool])(n : nat) : bool = x(n) => y(n);
   /=(x,y : sequence[bool])(n : nat) : bool = x(n) /= y(n);

   END lustre_bool
```

FIG. 2.17 – Définition des opérateurs booléens.

```
lustre_top : THEORY
   BEGIN
   IMPORTING lustre, lustre_reel, lustre_bool
   A(x : sequence[bool]) : bool = FORALL (n:nat) : x(n)
END lustre_top
```

FIG. 2.18 – La théorie principale.

```
node fibonacci (x,y : int) returns (f : int);
var pf : int;
let
   pf = y -> pre f;
   f = x -> pre (pf + f);
tel
```

FIG. 2.19 – Le nœud LUSTRE fibonacci.

```
fibonacci : THEORY
   BEGIN
   IMPORTING lustre_top

   fibonacci_type(x,y : sequence[int]) : TYPE = { out:[# pf,f:sequence[int] #]
      | out‘pf = y   pre(out‘f) AND out‘f = x   pre(out‘pf + out‘f) }


   fibonacci_axiom : AXIOM FORALL (x, y : sequence[int]) :
      EXISTS (out : fibonacci_type(x,y)) : TRUE

   fibonacci_node(x, y : sequence[int]) : fibonacci_type(x, y)

   positive : THEOREM FORALL (x, y : sequence[int]) :
      A(x >= 0) AND A(y >= 0) IMPLIES A(fibonacci_node(x, y)‘f >= 0)

END fibonacci
```

FIG. 2.20 – La théorie PVS pour le nœud fibonacci.

implémente des règles de preuves basées sur l'induction continue sur les flots LUSTRE.

Une partie du travail accompli durant cette thèse a porté sur la prise en compte des tableaux dans la validation des programmes LUSTRE. La première tentative que nous avons menée (abandonnée par la suite pour des raisons similaires à celles qui ont mené à l'abandon de la traduction systématique de LUSTRE en PVS) a consisté à étendre la sémantique de LUSTRE donnée en PVS par Cécile Dumas-Canovas aux itérateurs et de tenter de prouver des propriétés à l'aide du prouveur interactif de PVS. Nous reviendrons sur cette approche au chapitre 8.

2.3.4 Méthodes de test

Les méthodes de vérification par model-checking présentent une limitation liée à la taille de l'automate du système à vérifier (où au nombre de variables dans le BDD construit) : il n'est pas toujours possible de prouver une propriété. Une alternative depuis longtemps utilisée consiste à tester le programme en l'exécutant avec des séquences d'entrées couvrant au mieux les cas significatifs d'exécution du système à valider.

Dans le cas des systèmes réactifs, l'entrée courante (qui est fournie par l'environnement direct du système) dépend de l'histoire des sorties puisque celles-ci modifient constamment l'environnement.

L'approche retenue [RWNH98] pour LUSTRE (implémentée dans l'outil Lurette d'origine[4]) consiste à générer les valeurs d'entrées *à la volée* étant donnés:

- le programme à tester ;
- une spécification de son environnement. Les entrées ne sont pas choisies complètement au hasard mais toujours de façon à satisfaire la spécification de l'environnement.

Schéma de test – La propriété à vérifier ainsi que l'environnement sont spécifiés par des observateurs LUSTRE. On utilise un schéma de test identique au schéma de preuve de la figure 2.12. Néanmoins, les 3 nœuds LUSTRE Programme, Assertion et Propriété ne sont pas composés : le programme est considéré comme une *boîte noire* dont on ne connaît pas forcément le code lustre. On sait simplement comment l'exécuter *pas à pas*. Le test ainsi réalisé est *fonctionnel* par rapport à un test *structurel* pour lequel on a le code source du programme à tester.

L'algorithme utilisé dans l'outil Lurette peut être résumé comme suit:

```
LongueurCourante := 0;
répéter
  Entrées := Choix(Assertion);
  Sorties := Programme(Entrées)
  TestValide := Propriétés(Entrées,Sorties)
  LongueurCourante := LongueurCourante + 1;
jusqu'à LongueurCourante := LongueurMaximum
```

Cette version est extrêmement simple, mais elle permet de mettre en évidence :

- Une boucle de répétition qui doit être rapprochée de la *boucle de temps* des programmes synchrones. Le nombre de passage dans cette boucle est bornée par la longueur souhaitée pour le cas de test (variable LongueurMaximum) ;
- Une variable LongueurCourante qui mémorise en fait le nombre d'instants écoulés selon l'horloge globale. Cette mémoire permet de gérer la longueur d'une séquence de test ;
- Une fonction de Choix des entrées qui doivent satisfaire l'assertion (qui représente une spécification de l'environnement). Une possibilité consiste à tirer au hasard les nouvelles valeurs d'entrées et à recommencer le tirage jusqu'à ce que les valeurs calculées satisfassent l'Assertion ;
- Une variable TestValide qui vaut vrai tant que la propriété est satisfaite par le test en cours.

La difficulté réside ici dans la description de la fonction Choix qui est chargée de « deviner » des valeurs pour les variables d'entrées qui doivent satisfaire l'assertion.

Dans l'outil Lurette, l'utilisateur fournit une spécification de l'environnement sous la forme d'un nœud LUSTRE. Cette solution présente l'avantage de n'avoir à utiliser qu'un langage pour spécifier à la fois le système sous test et le test lui-même. Par contre, LUSTRE n'est adapté ni pour décrire les comportements indéterministes, ni pour décrire les comportements séquentiels.

Le travail de thèse de Yvan Roux [Rou04] a consisté à proposer un langage de description d'environnements à base d'expressions régulières (et permettant l'utilisation d'informations probabilistes) répondant à ces exigences et à fournir un environnement de test basé sur des environnements décrit à l'aide de ce langage.

2.3.5 Interprétation et débogage algorithmique

La simulation des programmes est un moyen complémentaire de validation par rapport aux méthodes que nous avons présentées jusqu'ici. Il est en général intéressant pour le développeur de pou-

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

voir observer le comportement d'un programme face à des réactions définies par lui-même.

FIG. 2.21 – La fenêtre de commande d'interprétation de Luciole.

L'outil Luciole permet de diriger et d'observer l'exécution d'un système décrit en LUSTRE-V4. La figure 2.21 montre la fenêtre principale de Luciole telle qu'on peut l'utiliser pour le programme du chien_de_garde présenté au paragraphe 2.2.1.7. Les trois cases à cocher à gauche permettent d'assigner une valeur aux entrées correspondantes (armer, desarmer et date_limite). Sur la droite, on peut identifier le passage de la variable alarme à vrai.

Luciole est un interpréteur simple. L'observation du comportement du système interprété se fait seulement par l'intermédiaire des valeurs d'entrées/sorties. L'étape suivante dans l'observation d'un programme est celle du débogage interactif. Le terme « débogage » rassemble des méthodes d'observation de l'exécution des programmes dans le but de déterminer la présence et éventuellement l'origine de bogues, c'est-à-dire des propriétés non désirées d'un programme. La base du débogage des programmes est l'interprétation : on doit pouvoir commander l'exécution du système pas-à-pas. On veut aussi pouvoir déterminer plus rapidement l'origine d'éventuels bogues rencontrés.

L'intérêt du débogage par rapport aux méthodes de vérification traditionnelles (qui appliquent des analyses statiques d'un modèle du programme) est qu'il applique des analyses dynamiques sur le programme pendant son exécution.

L'application de ces techniques aux programmes flots de données (comme ceux décrits en LUSTRE) est assez récente. Fabien Gaucher a étudié dans sa thèse [Gau03, MG00] le débogage des systèmes réactifs décrit en LUSTRE.

Le logiciel Ludic, développé par F. Gaucher[5], propose une interprétation des programmes LUSTRE basée sur une observation fine de leur état et un contrôle pointu de l'exécution (guidage de l'exécution, ré-exécution, points d'arrêts conditionnels, etc). Il propose aussi une méthode de débogage algorithmique pour pallier les difficultés de compréhension des dépendances de données dans un programme flot-de-données. Cette méthode fournit une procédure d'exploration entraînant un recul dans le temps de l'exécution : lorsqu'un bogue est rencontré, son origine est rarement instantanée, mais repose sur le passé plus ou moins lointain de l'exécution du système. Par le débogage algorithmique, on remonte ainsi le temps à partir de l'instant où l'erreur est effectivement détectée afin de déterminer l'instant *origine* de l'erreur.

Parallèlement, des techniques de slicing ont été proposées par Fabien Gaucher pour simplifier les programmes et permettre l'observation et l'interprétation des parties d'un programme réellement concernées par un bogue.

[5] LUDIC est disponible à l'adresse http://www-verimag.imag.fr/SYNCHRONE/LUDIC_Home_page.html

Chapitre 3

Modèles synchrones déterministes et non-déterministes

Il existe de nombreuses formes de spécification utilisées autour de Lustre. *Au chapitre 5, nous introduirons les contrats* assume-guarante. *Ces différents modèles sont destinés à cohabiter : on pourra avoir manipuler des composants décrits dans plusieurs formats. Il nous a donc paru indispensable de pouvoir décrire leur sémantique dans un cadre formel unifié. Le but du présent chapitre est de définir ce cadre formel ainsi que les différentes formes de spécification.*
Au paragraphe 3.2, nous présentons une sémantique de traces pour les composants synchrones. Puis nous nous intéressons à plusieurs formes de spécifications (paragraphe 3.4) de ces composants. Pour chacune, nous donnons sa sémantique de trace et des propriétés caractéristiques. Nous comparons ensuite au paragraphe 3.5 ces formes et essayons d'établir les transformations possibles entre elles. Tout du long, nous utilisons un même exemple (introduit au paragraphe 3.3) qui nous permet de mieux comprendre le sens de chaque forme de spécification. Enfin, nous présentons au paragraphe 3.6 une nouvelle forme de spécification dite par contrat, *qui sera détaillée au chapitre 5.*

3.1 Introduction

Le langage Lustre permet de décrire des comportements complètement déterministes : pour une séquence de valeurs d'entrées, il existe une unique séquence de valeurs de sorties correspondante. Les observateurs utilisés pour l'expression de propriétés de sûreté (voir paragraphe 2.3) sont une autre forme de description de comportements de systèmes réactifs qui peuvent être utilisés pour décrire des systèmes non-déterministes. D'autres formes de spécification sont utilisées par exemple pour le test. Les contrats que nous proposons au chapitre 5 en sont une autre.

Le but de ce chapitre est de définir la sémantique de ces différents modèles de spécifications dans un cadre formel unifié, afin de pouvoir manipuler ensemble des composants décrits dans des formes différentes. Nous allons tout d'abord voir qu'on peut définir un comportement d'un système comme un ensemble de traces des variables qu'il utilise. A l'aide d'un exemple, nous présenterons ensuite les différentes formes de spécification qui sont utilisées autour de Lustre, leurs propriétés, ainsi que les liens qui existent entre ces formes.

3.2 Sémantique de traces

3.2.1 Variables, valuations et traces

De manière générale, nous considérons un ensemble de variables noté V, prenant leur valeur dans un domaine D. D peut être considéré comme une union de types $\mathbb{B} \cup \mathbb{N} \cup \mathbb{R}$ (où $\mathbb{B}$ dénote les booléens, $\mathbb{N}$ les entiers naturels et $\mathbb{R}$ les réels), la distinction n'ayant pas d'importance pour le reste de la présentation.

Définition 1 — Valuation et trace

Étant donné un ensemble de variables V et un domaine de valeurs D, une valuation *est une fonction totale de V dans D (un élément de $V \rightarrow D$). Une* trace *est une séquence arbitrairement longue de valuations. C'est un élément de $(V \rightarrow D)^*$.*

Notations –

- On notera $|V|$ le cardinal d'un ensemble de variables V ;
- On notera $T(V)$ l'ensemble des traces possibles sur les variables V ;
- Dans les exemples, on notera par un tuple $< x_0, ..., x_i, ... >$ une valuation de $V = \{v_0, ..., v_i, ...\}$ où x_i est la valeur de la variable v_i dans la valuation. Une trace sera notée comme une suite de tuples $< x_0, ..., x_i, ... >< y_0, ..., y_i, ... >$;
- Étant donnée une trace $t \in T(V)$: On note $|t|$ la longueur de t ;
- Pour $n \in [0..|t| - 1]$, on note $t[n]$ les valeurs des variables de V au rang n de la trace t. On appellera n-ième *instant* de t, la valuation de rang n dans t. On parlera aussi de n-ième instant d'un ensemble de traces ;
- Une *sous-trace* notée $t[n..m]$ représente la sous-séquence de valeur des variables de V entre les rangs n et m de la trace t inclus ;
- Étant données deux traces s et t, on note par $s\ t$ la trace constituée de toutes les valuations de s suivies de toutes les valuations de t (concaténation de traces) ;
- Enfin, nous noterons les variables *tableaux* comme des vecteurs de variables scalaires. Par exemple, $v = [v_0, \ldots, v_{s-1}]$. Si k est un entier dans $[0..s-1]$, on note $v[k]$ le k-ième élément de v. Nous aurons notamment besoin de cette notation au chapitre 5.

Définition 2 — Projection d'une trace

Étant donnée une trace t sur V et $W \subseteq V$ un sous-ensemble des variables de V, $t\lfloor W \rfloor$ dénote la projection de t sur les variables de W, c'est-à-dire, la séquence de valeurs de V où seules les valeurs des variables de W ont été conservées.

EXEMPLE 6 — Considérons un ensemble de variables $V = \{x, y\}$ et une trace t sur V :

$$t =< 3, 4 >, < 4, 5 >, < 3, 2 >, ..., < 1, 2 >, < 3, 2 >, ...$$

Soit maintenant $W = \{x\}$. Alors, on a :

$$t\lfloor W \rfloor =< 3 >< 4 >, < 3 >, ..., < 1 >, < 3 >, ...$$

— FIN DE L'EXEMPLE 6

3.2.2 Ensembles de traces

Dans la suite nous nous intéressons à des ensembles de traces préfixe-clos.

3.2.2.1 Définitions

Définition 3 — Ordre partiel préfixe

On définit l'ordre partiel préfixe $\leq$ sur les traces d'un ensemble de traces τ par :

$$\forall \nu, \sigma \in \tau.\ \nu \leq \sigma \Leftrightarrow \exists \rho \mid \sigma \equiv \nu\rho$$

Plus simplement, $\nu \leq \sigma$ si et seulement si ν est un préfixe de σ.

Définition 4 — Ensemble de traces préfixe-clos

On dit qu'un ensemble de traces τ est préfixe-clos si et seulement si tout préfixe d'une trace de τ est aussi une trace de τ :

$$\forall t \in \tau.\ t' \leq t \Rightarrow t' \in \tau.$$

3.2.2.2 Opérations sur les ensembles de traces

Soit τ un ensemble de traces sur des variables V. Nous définissons tout d'abord l'opération de projection de τ sur un sous-ensemble de V.

Définition 5 — Projection d'un ensemble de traces

Soit τ un ensemble de traces sur V. Soit $W \subseteq V$. $\tau\lfloor W \rfloor$ dénote la projection *de τ sur W, et est défini par :*

$$\tau\lfloor W \rfloor = \{t \in T(W) \mid \exists t' \in \tau.\ t = t'\lfloor W \rfloor\}$$

Les traces de $\tau\lfloor W \rfloor$ sont les traces de τ où les variables de $V \setminus W$ ont été masquées. La projection de l'ensemble de traces sur W est simplement l'ensemble des traces toutes projetées sur W.

Nous définissons ensuite l'extension d'un ensemble de traces à un sur-ensemble des variables considérées.

Définition 6 — Extension d'un ensemble de traces

Soit τ un ensemble de traces sur V. Soit $W \supseteq V$. $\tau\lceil W \rceil$ dénote l' extension *de τ à W, et est défini par :*

$$\tau\lceil W \rceil = \{t \in T(W) \mid \exists t' \in T(V), t'' \in T(W \setminus V).t\lfloor V \rfloor = t' \wedge t\lfloor W \setminus V \rfloor = t''\}$$

Les variables ajoutées à V ne sont pas contraintes dans $\tau\lceil W \rceil$: elles peuvent prendre n'importe quelle valeur de leur domaine.

EXEMPLE 7 — L'extension d'un ensemble de traces à *une* variable consiste à rajouter autant de traces que nécessaires pour que la variable ajoutée ne soit pas contrainte. Considérons les traces de la figure 3.1 qui portent sur un ensemble de variables $V = \{b : \text{bool}, s : \text{int}\}$. On peut étendre les traces de la figure 3.1 avec l'ensemble $W = \{c : bool\}$. On obtient alors les traces décrites à la figure 3.2 où chaque trace a été démultipliée de manière à conserver à tout instant un choix possible entre les 2 valeurs de c.

— FIN DE L'EXEMPLE 7

$$
\begin{array}{cccc}
< tt, 2 > & < tt, 1 > & < ff, 5 > & \ldots \\
< tt, 3 > & < ff, 4 > & < ff, 4 > & \ldots \\
< ff, 5 > & < tt, 3 > & < tt, 2 > & \ldots
\end{array}
$$

FIG. 3.1 – Un ensemble de traces

$$
\begin{array}{ccc}
< tt, 2, tt > & < tt, 1, tt > & < ff, 5, tt > \\
< tt, 2, tt > & < tt, 1, ff > & < ff, 5, tt > \\
< tt, 2, tt > & < tt, 1, tt > & < ff, 5, ff > \\
< tt, 2, tt > & < tt, 1, ff > & < ff, 5, ff > \\
< tt, 2, ff > & < tt, 1, tt > & < ff, 5, tt > \\
< tt, 2, ff > & < tt, 1, ff > & < ff, 5, tt > \\
< tt, 2, ff > & < tt, 1, tt > & < ff, 5, ff > \\
< tt, 2, ff > & < tt, 1, ff > & < ff, 5, ff >
\end{array}
$$

$$
\begin{array}{ccc}
< tt, 3, tt > & < ff, 4, tt > & < ff, 4, tt > \\
< tt, 3, tt > & < ff, 4, ff > & < ff, 4, tt > \\
< tt, 3, tt > & < ff, 4, tt > & < ff, 4, ff > \\
< tt, 3, tt > & < ff, 4, ff > & < ff, 4, ff > \\
< tt, 3, ff > & < ff, 4, tt > & < ff, 4, tt > \\
< tt, 3, ff > & < ff, 4, ff > & < ff, 4, tt > \\
< tt, 3, ff > & < ff, 4, tt > & < ff, 4, ff > \\
< tt, 3, ff > & < ff, 4, ff > & < ff, 4, ff >
\end{array}
$$

$$
\begin{array}{ccc}
< ff, 5, tt > & < tt, 3, tt > & < tt, 2, tt > \\
< ff, 5, tt > & < tt, 3, ff > & < tt, 2, tt > \\
< ff, 5, tt > & < tt, 3, tt > & < tt, 2, ff > \\
< ff, 5, tt > & < tt, 3, ff > & < tt, 2, ff > \\
< ff, 5, ff > & < tt, 3, tt > & < tt, 2, tt > \\
< ff, 5, ff > & < tt, 3, ff > & < tt, 2, tt > \\
< ff, 5, ff > & < tt, 3, tt > & < tt, 2, ff > \\
< ff, 5, ff > & < tt, 3, ff > & < tt, 2, ff >
\end{array}
$$

FIG. 3.2 – Les traces de la figure 3.1 étendues à $W = \{c\}$.

3.2.3 Composants

Pour toutes les formes de spécifications que nous étudions au paragraphe 3.4, nous considérons des *composants* comme celui de la figure 3.3 manipulant 2 ensembles de variables : un ensemble d'entrées (noté I) et un ensemble de variables de sorties (noté O). La sémantique d'un tel composant est décrite par un ensemble de traces *préfixe-clos* sur $I \cup O$. Ces formes de spécifications décrivent toujours des ensembles de traces préfixe-clos.

FIG. 3.3 – Un composant .

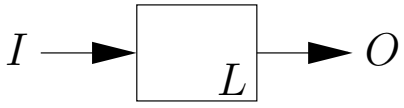


FIG. 3.4 – Un composant avec variables locales.

Nous définissons maintenant la notion de trace et d'ensemble de traces associé à un tel composant ainsi que des opérations de manipulation de ces traces.

Définition 7 — Ensemble de traces d'un composant

Soit un composant P manipulant les variables I (entrées), O (sorties). Les comportements possibles de P sont totalement décrits par un ensemble de traces :

$$\tau(P) \subseteq ((I \cup O) \rightarrow D)*$$

Les traces d'un composant sont les séquences de valeurs que prennent les variables du composant pendant son exécution.

On peut aussi associer à un composant un ensemble de variables locales qui représentent des mémoires utilisées localement (voir figure 3.4). La sémantique est alors décrite par un ensemble de traces sur $I \cup O \cup L$, noté $\tau_{loc}(P)$.

Définition 8 — Ensemble de traces d'un programme à variables locales

Soit un programme P manipulant les variables I (entrées), O (sorties) et L (locales). Les comportements possibles de P sont totalement décrits par l'ensemble de traces :

$$\tau_{loc}(P) \subseteq ((I \cup O \cup L) \rightarrow D)*$$

Les variables locales peuvent être cachées afin d'exprimer la sémantique du programme en terme de traces sur les entrées sorties $I \cup O$. On a alors :

$$\tau(P) = \tau_{loc}(P)\lfloor I \cup O \rfloor.$$

3.2.4 Réactivité et déterminisme par rapport aux entrées

Nous définissons maintenant formellement 2 notions fondamentales qui sont la *réactivité* et le *déterminisme* d'un ensemble de traces. Ces deux notions sont étroitement liées à la distinction qui est faite parmi les variables V entre les entrées I et les sorties O du composant considéré. Informellement, un ensemble de traces τ est dit *réactif* par rapport à I si et seulement si, à tout moment, toute trace de τ est prolongeable : quelle que soit la valuation courante de I, on peut trouver une nouvelle valuation pour O.

$< ff, 2 >$ $< tt, 1 >$ $< ff, 5 >$
$< tt, 3 >$ $< ff, 4 >$ $< tt, 4 >$

FIG. 3.5 – Un ensemble de traces déterministe et réactif.

$< ff, 2 >$ $< tt, 1 >$ $< tt, 5 >$
$< tt, 3 >$ $< ff, 4 >$ $< tt, 4 >$

FIG. 3.6 – Un ensemble de traces non déterministe et non réactif.

Définition 9 — Réactivité d'un ensemble de traces

On dit que τ est réactif par rapport aux entrées I si et seulement si toute trace d de $\tau\lfloor I \rfloor$ de longueur n est telle que pour chaque valuation possible i des I à l'instant n, il existe une trace t identique à d jusqu'au rang $n-1$ et qui associe la valeur i aux entrées à l'instant n.
τ est réactif si et seulement si :

$$\begin{array}{l}\forall n \in \mathbb{N}.\ \forall i \in (I \rightarrow D).\ \forall d \in \tau\lfloor I \rfloor \ \mathsf{tq}\ |d| = n. \\ \quad \exists t \in \tau \mid t[0..n-1]\lfloor I \rfloor = d \\ \qquad \wedge\ t[n]\lfloor I \rfloor = i.\end{array}$$

En d'autres termes, toute trace d du comportement du composant est toujours prolongeable.

τ est dit *déterministe* par rapport à I si et seulement si, à toute séquence de valuations de I, correspond une et une seule séquence de valuations de O.

Définition 10 — Déterminisme d'un ensemble de traces

Soit un ensemble de traces τ. τ est dit déterministe si et seulement si :

$$\forall t1, t2 \in \tau.\ t1\lfloor I \rfloor = t2\lfloor I \rfloor \Rightarrow t1\lfloor O \rfloor = t2\lfloor O \rfloor.$$

Un ensemble τ de traces est dit *déterministe* si et seulement si pour chaque séquence de valeurs d'entrées, il existe une et une seule séquence de sorties correspondante.

EXEMPLE 8 — Considérons $I = \{b : bool\}$ et $O = \{s : int\}$. L'ensemble de traces représenté à la figure 3.1 est réactif vis-à-vis des entrées I : à chaque instant il existe une réaction pour toute valeur possible de b. Mais cet ensemble n'est pas déterministe, puisqu'il est possible (par exemple) d'obtenir, au premier instant, 2 valeurs différentes pour s pour la valeur tt de b. Ce n'est pas le cas pour l'ensemble de traces de la figure 3.5 qui est à la fois réactif *et* déterministe vis-à-vis de l'entrée b. Enfin, l'ensemble décrit à la figure 3.6 n'est ni réactif ni déterministe, et ce à cause du troisième instant : l'ensemble de traces ne réagit pas à l'occurrence d'une valeur ff de b (non-réactivité) et si b prend la valeur tt, on ne sait pas quelle valeur produire pour s (non-déterminisme).

— FIN DE L'EXEMPLE 8

On peut aussi définir la réactivité et le déterminisme d'un ensemble de traces par rapport à des entrées qui sont contraintes. Cela nous permet d'exprimer le fait qu'une contrainte qu'on exige sur les entrées d'un composant (spécifiée par un ensemble de traces) ne rend pas le composant non-réactif ou non-déterministe.

Définition 11 — Réactivité par rapport à un ensemble contraint d'entrées

Soit τ_C un ensemble de traces sur I spécifiant une contrainte sur les entrées du composant. On dit que τ est réactif par rapport aux entrées I contraintes par τ_C si et seulement si toute trace d de $\tau\lfloor I \rfloor$ de longueur n satisfaisant la contrainte représentée par τ_C est telle que pour chaque valuation possible i des I à l'instant n, il existe une trace t identique à d jusqu'au rang $n-1$, qui associe la valeur i aux entrées à l'instant n et qui satisfait aussi τ_C.
τ est réactif par rapport à I contraint par τ_C si et seulement si :

$$\begin{array}{l}\forall n \in \mathbb{N}.\ \forall i \in (I \rightarrow D).\ \forall d \in \tau\lfloor I \rfloor \ \textit{tq}\ |d| = n. \\ \quad \exists t \in \tau \ \textit{tq}\ (d \in \tau_C \\ \qquad\qquad \Rightarrow (t[0..n-1]\lfloor I \rfloor = d \\ \qquad\qquad\quad \wedge\, t[n]\lfloor I \rfloor = i \\ \qquad\qquad\quad \wedge\, t[0..n]\lfloor I \rfloor \in \tau_C)).\end{array}$$

EXEMPLE 9 — Soit $I = \{a\}$. Considérons l'ensemble de traces τ suivant portant sur les variables $\{a, b\}$:

$$\begin{array}{cccc} < 12,2 > & < 11,1 > & < 13,5 > & < 12,4 > \\ < 11,3 > & < 12,4 > & < 11,4 > & < 13,1 > \\ < 13,3 > & < 13,4 > & < 12,4 > & < 11,15 > \end{array}$$

Cet ensemble de traces est réactif par rapport à I contraint avec τ_C :

$$\begin{array}{cccc} < 11 > & < 11 > & < 11 > & < 11 > \\ < 12 > & < 12 > & < 12 > & < 12 > \\ < 13 > & < 13 > & < 13 > & < 13 > \end{array}$$

qui définit les traces où $10 < a \leq 13$. En effet, ici $\tau\lfloor I \rfloor$ est exactement égal à τ_C. Par contre, τ n'est pas réactif par rapport à I contraint avec τ'_C qui définit les traces où $10 \leq a \leq 13$.
— FIN DE L'EXEMPLE 9

Remarque 3 *Si τ est réactif par rapport à un ensemble I contraint par τ_C et que $\tau_{C'} \subseteq \tau_C$ alors τ est réactif par rapport à $\tau_{C'}$. En d'autres termes, si un ensemble de traces est réactif par rapport à un ensemble de traces donné, il est aussi réactif par rapport à un sous-ensemble de cet ensemble de traces.*

EXEMPLE 10 — Si l'on revient sur l'exemple précédent, et que l'on considère $\tau_{C'} \subseteq \tau_C$ défini comme :

$$\begin{array}{cccc} < 11 > & < 11 > & < 11 > & < 11 > \\ < 12 > & < 12 > & < 12 > & < 12 > \end{array}$$

alors τ est réactif par rapport à I contraint par $\tau_{C'}$.
— FIN DE L'EXEMPLE 10

Remarque 4 *Si τ n'est pas réactif par rapport à τ_C, il peut être réactif par rapport à $\tau_{C'}$ si $\tau_{C'} \subset \tau_C$.*

EXEMPLE 11 — Toujours dans le même exemple, τ n'est pas réactif par rapport à $\tau_{C''}$ si $\tau_{C''}$ est défini comme l'ensemble des traces dans lequel $10 < a \leq 16$. Il est par contre réactif par rapport à τ_C qui est tel que $\tau_C \subset \tau_{C''}$.
— FIN DE L'EXEMPLE 11

$$\begin{array}{cccc} <0,\underline{1},2> & <2,\underline{4},10> & <7,\underline{7},9> & <4,\underline{7},12> \\ <0,\underline{1},2> & <2,\underline{5},10> & <7,\underline{8},9> & <4,\underline{9},12> \\ <0,\underline{1},2> & <2,\underline{6},10> & <7,\underline{8},9> & <4,\underline{9},12> \end{array}$$

FIG. 3.7 – Des traces produites par le composant C.

Définition 12 — Déterminisme par rapport à un ensemble contraint d'entrées

Soit un ensemble de traces τ. Soit τ_C un ensemble de traces sur I spécifiant une contrainte sur les entrées du composant. τ est dit déterministe par rapport à I contraint par τ_C si et seulement si :

$$\begin{array}{r} \forall t1, t2 \in \tau.\ (t1\lfloor I \rfloor \in \tau_C \\ \wedge\ t2\lfloor I \rfloor \in \tau_C \\ \wedge\ t1\lfloor I \rfloor = t2\lfloor I \rfloor) \\ \Rightarrow t1\lfloor O \rfloor = t2\lfloor O \rfloor. \end{array}$$

3.3 Exemple

Nous présentons maintenant un exemple de composant que nous décrirons par la suite sous toutes les formes de spécifications que nous introduisons.

On considère un composant C qui reçoit deux entrées $I1$ et $I2$ et qui produit une sortie O. Son comportement est décrit par l'ensemble de traces suivant :

$$\begin{array}{rl} \tau(C) = & \{t \in T(\{I1, I2, O\}) \mid \\ & t\lfloor I1 \rfloor[0] \ \leq\ t\lfloor O \rfloor[0]\} \\ & \wedge\ \forall n \in [1..|t|-1]. \\ & \quad t\lfloor I1 \rfloor[n] \ \leq\ t\lfloor O \rfloor[n] \ <\ t\lfloor I2 \rfloor[n-1]\} \end{array}$$

À tout instant n, la valeur de O est comprise entre la valeur courante de $I1$ et la valeur qu'avait $I2$ à l'instant précédent ($t\lfloor I1 \rfloor[n] \ \leq\ t\lfloor O \rfloor[n] \ <\ t\lfloor I2 \rfloor[n-1]\}$). A l'instant initial, nous exigeons seulement que O soit plus grand que $I1$ puisque toute comparaison avec la valeur précédente n'a aucun sens ($t\lfloor I1 \rfloor[0] \ \leq\ t\lfloor O \rfloor[0]\}$).

Nous reviendrons sur cet exemple pour illustrer chacun des modèles que nous présentons dans la section suivante. Pour l'instant, nous donnons à la figure 3.7 quelques exemples de traces appartenant à $\tau(C)$. Les triplets dénotent <la valeur courante de $I1$, la valeur courante de O (soulignée), la valeur précédente de $I2$>. Remarquons simplement que $\tau(C)$ est indéterministe par rapport aux entrées de C, puisque pour la même séquence de valeurs de $I1, I2$, plusieurs séquences différentes de O sont possibles.

3.4 Différentes formes de spécifications

On considère des composants manipulant 3 ensembles de variables : des entrées I, des sorties O et des variables locales L. Les comportements d'un composant sont décrits par un ensemble de traces sur $I \cup O$.

Nous étudions maintenant différentes formes de spécifications. Tout d'abord, les *nœuds* LUSTRE « simples » (comme présentés en 2.2.1.3) décrivent des comportements réactifs et déterministes. Les

observateurs que nous avons présentés en 2.3.1, permettent d'établir des relations entre les entrées et les sorties d'un programme, décrivant ainsi des accepteurs de propriétés de sûreté. Les comportement décrits par un observateur sont potentiellement non-déterministes.

Les *nœuds à variables locales* indéfinies permettent eux aussi de décrire des comportements non-déterministes, en laissant *non-définies* les valeurs d'une partie de leurs variables locales. On suppose alors que ces variables prennent leur valeurs de manière aléatoire. À partir de la catégorie précédente, il est possible de « sortir » les variables locales indéfinies et de les considérer comme des entrées un peu particulières du composant. Leur valeur n'est alors ni fixée par l'environnement, ni choisie aléatoirement mais fixée par un *oracle* extérieur au programme. On parle alors de *nœuds à oracles*.

Nous décrivons ensuite la sémantique des automates *Lucky*, introduits dans [Rou04] pour la spécification de comportements non-déterministes dans le cadre du test. Nous étudions aussi une forme générale de spécifications appelée *step-relation*. On décrit l'évolution d'un composant pas à pas : étant donnée la valeur v des variables du programme à un instant t, on définit les valeurs v' prises par les variables à l'instant $t + 1$. Nous donnons pour chaque forme de spécification sa syntaxe dérivée de LUSTRE et sa sémantique de trace. Nous illustrons chaque forme à l'aide de l'exemple présenté en 3.3. Nous conclurons cette partie par une courte discussion sur les connexions qui existent entre ces différentes formes de spécification.

3.4.1 Nœuds LUSTRE

Un nœud LUSTRE N est un programme réactif et déterministe décrit par un quadruplet (I, O, L, Eq). I, O et L sont les ensembles, respectivement d'entrées, de sorties et de variables locales. Eq est un ensemble d'équations de la forme $var = expr$ où var est une variable de $O \cup L$ et $expr$ une expression définie par la grammaire suivante :

$$expr ::= c \mid x \mid op(e, ..., e) \mid pre(c, y).$$

dans laquelle c représente les constantes de D, x dénote une variable de $I \cup O \cup L$, op dénote les opérateurs combinatoires, et $pre(c, y)$ représente la valeur précédente de y, initialisée par une constante c. Cet opérateur pre est une forme dégénérée des opérateurs pre et -> de LUSTRE : on ne peut appliquer $pre(c, y)$ qu'à des variables, et on donne immédiatement leur initialisation.

À partir des équations Eq d'un composant, le compilateur sait construire 2 fonctions sur des flots : f calcule la valeur courante des sorties en fonction de l'histoire des entrées et des variables locales ; g calcule la mise à jour des variables locales (de l'état du composant) en fonction de l'histoire des entrées et des variables locales. On a :

$$f : ((I \cup L) \to D)^* \to (O \to D)$$
$$g : ((I \cup L) \to D)^* \to (L \to D)$$

Les n-ièmes valeurs de O et L sont définies comme suit :

$$\forall n \in \mathbb{N},\ O[n] = f(I[0], \ldots, I[n], L[0], \ldots, L[n])$$
$$\text{et } \forall n \in \mathbb{N},\ L[n] = g(I[0], \ldots, I[n], L[0], \ldots, L[n])$$

Comme il a été souligné plus haut (2.2), le compilateur interdit tout cycle de dépendance instantanée entre les variables. Ces deux définitions sont donc sans cycle. La valeur d'une variable de L à un instant donné ne peut dépendre d'elle-même. Par contre, elle peut tout à fait dépendre des valeurs à cet instant d'autres variables locales, à condition qu'on puisse établir un ordre d'évaluation strict des valeurs de ces variables.

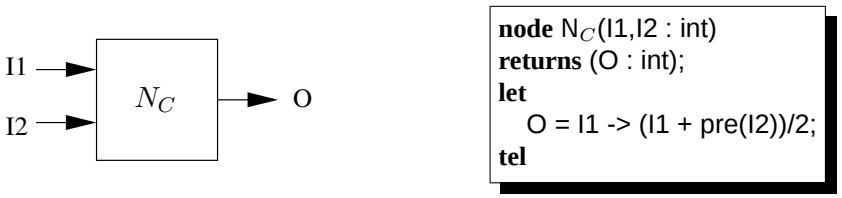


FIG. 3.8 – Un nœud LUSTRE implémentant la spécification de C.

Mémoire bornée – Dans les langages séquentiels, la valeur courante des sorties (et des variables locales) d'un programme peut dépendre de l'histoire complète des entrées, comme c'est le cas dans la description donnée de f et g. Dans le cas des langages synchrones, toutefois, la taille de la mémoire utilisée par le programme est connue statiquement.

Le calcul de la valeur courante des sorties et variables locales se fait donc sur le valeur courante des entrées et sur une abstraction de l'histoire des entrées (représentée par une fonction Abs) qui caractérise la mémoire bornée du programme. À un instant donné, la valeur des sorties et des variables locales d'un programme peut donc être calculée en fonction de Abs :

$$\forall n \in \mathbb{N},\ O[n] = f'(Abs(I[0], \ldots, I[n-1]), I[n], L[n])$$
$$\text{et } \forall n \in \mathbb{N},\ L[n] = g'(Abs(I[0], \ldots, I[n-1]), I[n], L[n])$$

Notons que Abs peut être calculée incrémentalement grâce à une fonction de mise à jour de la mémoire, notée h, définie par : $Abs(I[0], \ldots, I[n]) = h(Abs(I[0], \ldots, I[n-1]), I[n])$. En d'autres termes, si on note $Abs[n]$ l'état de la mémoire à l'instant n, on a : $Abs[n] = h(Abs[n-1], I[n])$. On en déduit que O et L peuvent être calculées par :

$$\forall n \in \mathbb{N},\ O[n] = f'(Abs[n], I[n], L[n])$$
$$\text{et } \forall n \in \mathbb{N},\ L[n] = g'(Abs[n], I[n], L[n])$$

Par abus de notation, et parce qu'on sait toujours calculer f' et g', on utilisera le plus souvent f et g directement dans les définitions.

3.4.1.1 Sémantique

N définit un ensemble de traces sur (I, O), noté $\tau_{\text{Lustre}}(N)$ et tel qu'à chaque instant, la valeur des variable de O est entièrement définie par la fonction f et les variables locales sont aussi entièrement définies par la fonction g. Cette sémantique correspond au modèle de programme entrées/sorties présenté plus haut (voir figure 3.3) : les variables locales ne sont pas visibles de l'extérieur du composant.

$$\begin{aligned}\tau_{\text{Lustre}}(N) = \quad & \{t \in T(I \cup O) \mid \exists t_L \in T(L) \\ & \text{t.q } \forall n \in [0..|t|-1].\ t(n)\lfloor O \rfloor = f(t[0..n]\lfloor I \rfloor,\ t_L[0..n]) \\ & \qquad \text{et } \ t_L(n) = g(t[0..n]\lfloor I \rfloor,\ t_L[0..n])\}\end{aligned}$$

EXEMPLE 12 — À partir de la spécification du composant donnée au paragraphe 3.3, on peut décrire un nœud LUSTRE qui implémente cette spécification. La figure 3.8 décrit une instanciation simple possible de cette spécification : O est calculée comme la moyenne de I1 et I2. L'ensemble des traces décrit par N$_C$ est un sous-ensemble de l'ensemble de traces décrit par la spécification que nous

avons donnée plus haut pour C (on a $\tau_{\text{Lustre}}(\mathsf{N}_C) \subseteq \tau(C)$). La figure 3.9 donne une trace appartenant à $\tau_{\text{Lustre}}(\mathsf{N}_C)$.

— FIN DE L'EXEMPLE 12

$$< 0, \underline{1}, 2 > \quad < 2, \underline{6}, 10 > \quad < 7, \underline{8}, 9 > \quad < 4, \underline{9}, 12 >$$

FIG. 3.9 – Une trace du nœud N_C.

3.4.1.2 Propriétés

Comme nous l'avons déjà remarqué un nœud LUSTRE est

- déterministe par rapport à I : pour n'importe quelle valeur des entrées, on peut calculer *au plus* une valeur des sorties. Dans l'exemple ci-dessus la valeur de O est complètement déterminée par les valeurs de I1 et de pre(I2) ;
- réactif par rapport à I : pour n'importe quelle valeur des entrées, on peut calculer *au moins* une valeur des sorties. Dans l'exemple ci-dessus, quelles que soient les valeurs de I1 et pre(I2), on sait toujours calculer I1 -> (I1 + pre(I2))/2.

3.4.2 Observateurs

Un observateur (voir section 2.3.1) est un nœud LUSTRE avec une sortie booléenne. C'est une façon de représenter un accepteur d'une propriété sur les entrées/sorties d'un composant.

Soit un programme P portant sur un ensemble de variables d'entrées I et un ensemble de sorties O. On note $X = I \cup O$. Un observateur Obs est un nœud LUSTRE qui permet la description d'une propriété de sûreté Φ sur les variables de X de P. Obs possède pour entrée les variables de X, et émet une seule sortie booléenne ok vrai tant que les traces de X satisfont Φ. Un observateur est donc un quintuplet (X, ok, L, f, g) où g est une fonction de l'abstraction de l'histoire des I et des O qui calcule la valeur des variables locales et f est une fonction de l'abstraction de l'histoire des I et des O qui calcule la valeur de vérité de la propriété. Comme dans le cas des nœuds LUSTRE, les n-ièmes valeurs de ok et de L sont calculées à l'aide de 2 fonctions f et g :

$$\begin{array}{c} \forall n \in \mathbb{N},\ ok[n] = f(I[0], \ldots, I[n], L[0], \ldots, L[n]) \\ \text{et } \forall n \in \mathbb{N},\ L[n] = g(I[0], \ldots, I[n], L[0], \ldots, L[n]) \end{array}$$

3.4.2.1 Sémantique

O décrit donc un ensemble de traces $\tau_{\text{Obs}}(O)$ sur les variables X, défini comme suit :

$$\begin{array}{rl} \tau_{\text{Obs}}(O_C) = & \{t \in T(X) \mid \exists t_L \in T(L) \\ & \text{t.q. } \forall n \in [0..|t|-1].\ t_L(n) = g(t(0..n)\lfloor I \rfloor,\ t_L(0..n))\} \\ & \text{et } \ g(t(0..n), t_L(0..n)) = vrai\} \end{array}$$

A chaque instant, les variables de L sont totalement définies par la fonction f et dépendent de l'histoire des entrées et des variables locales. La validité de la propriété Φ est vérifiée par la fonction g et dépend elle aussi de l'histoire des entrées et des variables locales.

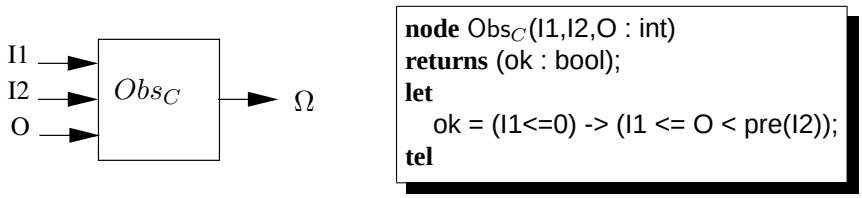


FIG. 3.10 – Un observateur tiré de la spécification de C.

EXEMPLE 13 — L'observateur de la figure 3.10 décrit exactement l'ensemble de traces du composant C.

— FIN DE L'EXEMPLE 13

3.4.2.2 Propriétés

Un observateur est un nœud LUSTRE habituel calculant la valeur d'une variable booléenne ok. Le comportement de ok est donc déterministe et réactif par rapport à X. En se rappelant que $X = I \cup O$ et que l'observateur est utilisé pour décrire des traces d'un composant à entrées I et à sorties O, le comportement décrit pour les variables de O est non-déterministe par rapport aux entrées I. Dans l'exemple ci-dessous. Si I1 vaut 1 et pre(I2) vaut 5 alors O peut prendre n'importe quelle valeur parmi 2,3 ou 4.

L'ensemble de traces décrit par un observateur peut ne pas être réactif. C'est le cas des assertions non-causales que nous avons présenté au paragraphe 2.3.2.

3.4.3 Nœuds à variables locales indéfinies

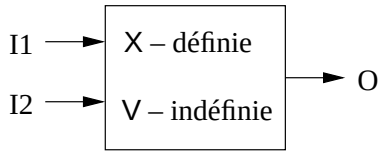
Les nœuds LUSTRE dans lesquels certaines variables locales (les Ω) ne sont pas définies peuvent également servir de description de comportements non-déterministes. Ces variables locales peuvent prendre n'importe quelle valeur, elles ne sont pas contraintes. On pourrait néanmoins imaginer compiler ces nœuds en du code où la valeur de chaque variable indéfinie est calculée à l'aide d'une fonction aléatoire « random ».

Un tel nœud peut posséder des variables locales définies (les L). C'est donc un tuple (I, Ω, O, L, f, g) où f calcule la valeur des variables locales L et g calcule la valeur des variables de O.

3.4.3.1 Sémantique

Nous retrouvons la définition d'un ensemble de traces d'un nœud LUSTRE standard, mis à part le fait que la trace t_Ω des variables Ω n'est pas contrainte. La trace t_L des L est entièrement définie par g. Nous donnons ci-dessous une sémantique des nœuds à variables locales indéfinies en terme de traces sur $I \cup O$.

$$\begin{aligned}\tau_{\text{Indef}}(Nindet_C) = \quad & \{t \in T(I \cup O) \mid \exists t_\Omega \in T(\Omega), t_L \in T(L) \\ & \text{t.q } \forall n \in [0..|t|-1].\ t(n)\lfloor S \rfloor = f(t[0..n]\lfloor I \rfloor,\ t_\Omega[0..n],\ t_L[0..n]) \\ & \qquad \text{et} \quad t_L(n) = g(t[0..n]\lfloor I \rfloor,\ t_\Omega[0..n],\ t_L[0..n])\}\end{aligned}$$




```
node Nindet_C(I1,I2 : int) returns (O : int);
var X : int;
indef V : int;
let
   X = I1 + V;
   O = I1 -> if (X>=pre(I2))
             then pre(I2)-1
             else X
tel
```

FIG. 3.11 – Un nœud à variables locales indéfinies implémentant la spécification de C.

EXEMPLE 14 — La figure 3.11 montre une implémentation possible de C à l'aide d'un nœud à variables locales non-définies. Le comportement de la variable V n'est pas définie : elle peut prendre n'importe quelle valeur entière. V représente la distance qui existe entre $I1$ et la valeur recherchée (X). On tire cette distance au hasard. Si, par chance, la valeur ainsi obtenue pour X est inférieure à $pre(I2)$, alors on donne à O la valeur trouvée pour X. Sinon, on lui donne la valeur de $pre(I2) - 1$. Dans tous les cas, on a bien $I1 \leq O < pre(I2)$.

— FIN DE L'EXEMPLE 14

Remarque 5 *La répartition des valeurs de la variable O est plutôt mauvaise : O prendra fréquemment la valeur $pre(I2) - 1$. Néanmoins, il est difficile de définir un critère de couverture d'une contrainte quelconque et cela ne fait pas partie des objectifs de ce travail.*

3.4.3.2 Propriétés

Le comportement d'un nœud à variables locales indéfinies est non-déterministe par l'absence de définition des variables Ω. Dans l'exemple de la figure 3.11, V n'est pas défini. Elle peut prendre n'importe quelle valeur sur $\mathbb{N}$ indépendamment des valeurs associées à I1 et I2. Donc X, qui dépend directement la valeur de V peut prendre un nombre quelconque de valeurs pour chaque valeurs des entrées I1 et I2. Il en va de même pour O.

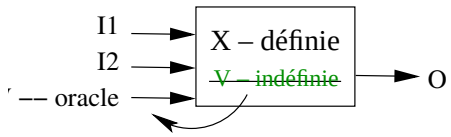
Le comportement d'un nœud à variables locales indéfinies est réactif vis-à-vis de I : Il peut réagir à n'importe quelle séquence d'entrées (les I ne sont pas contraints), seule change la façon dont son calculés les variables locales et les sorties.

3.4.4 Nœuds à oracles

Les nœuds à variables indéfinies ne sont pas autorisés dans la syntaxe du langage LUSTRE. Il est néanmoins possible d'exprimer des comportements similaires en utilisant des nœuds dit *à oracles*.

Un nœud à oracles est un nœud à variables locales indéfinies dans lequel on considère les variables locales indéfinies comme des entrées du programme. C'est donc un n-uplet (I, Ω, O, L, f, g) où f et g ont le même rôle que précédemment. La construction de $Noracle$ à partir de $Nindet$ est purement syntaxique.

L'interprétation de $Noracle$ comme un nœud à oracles est laissée aux outils (ou au lecteur le cas échéant) : rien ne les distingue des nœuds LUSTRE standard.




```
node Noracle_C(I1,I2,V : int) returns (O : int);
var X : int;
let
   X = I1 + V;
   O = I1 -> if (X>=pre(I2))
             then pre(I2)
             else X
tel
```

FIG. 3.12 – La variable locale V a été sortie en *oracle*.

3.4.4.1 Sémantique

La sémantique de $Noracle$ peut être définie en termes de traces sur les $I \cup \Omega \cup O$ comme pour un nœud LUSTRE normal. On a alors :

$$\begin{aligned}\tau_{\text{Lustre}}(Noracle) = \quad & \{t \in T(I \cup \Omega \cup S) \mid \exists t_L \in T(L) \\ & \text{t.q } \forall n \in [0..|t|-1].\ t(n)\lfloor O \rfloor \ = \ f(t[0..n]\lfloor I \rfloor,\ t_L[0..n]) \\ & \qquad \text{et} \ \ t_L(n) \ = \ g(t[0..n]\lfloor I \rfloor,\ t_L[0..n])\}\end{aligned}$$

Cependant, les variables Ω ne doivent pas être considérés comme de véritables entrées du programme, leur valeur étant en fait fixée par un « oracle » extérieur. On défini alors la sémantique d'un tel programme par un ensemble de traces sur $I \cup O$, noté τ_{Oracle} et définit par :

$$\tau_{\text{Oracle}}(Noracle_C) = \tau_{\text{Lustre}}(Noracle_C)\lfloor I \cup O \rfloor,$$

On a en fait :

$$\tau_{\text{Oracle}}(Noracle_C) = \tau_{\text{Indef}}(Nindef_C),$$

où $Noracle_C$ est obtenu en faisant apparaître toutes les variables locales indéfinies de $Nindef_C$ comme des entrées supplémentaires du composant. Seule diffère la façon dont les valeurs des variables « indéfinies » sont calculées.

EXEMPLE 15 — La figure 3.12 montre le nœud à oracle obtenu à partir du nœud de la figure 3.11 en « sortant » simplement la variable indéfinie V comme une nouvelle entrée du composant.
— FIN DE L'EXEMPLE 15

3.4.4.2 Propriétés

En tant qu'ensemble de traces sur $I \cup O$, un nœud à oracle possède les même propriétés que le nœud à variables locales indéfinies correspondant.

3.4.5 Automates « Lucky »

Dans le cadre du test, les travaux de thèse de Yvan Roux [Rou04] ont fourni un nouveau mode d'expression des comportements non-déterministes. A haut niveau, un langage a été proposé (Lutin) pour permettre de décrire ces comportements. Ce langage permet de maîtriser finement l'indéterminisme par l'intermédiaire d'opérateurs inspirés des expressions régulières : le choix non-déterministe, la répétition d'un sous-comportement (itération). Une notion de poids associée aux branches d'un choix

non-déterministe permet de rendre compte de caractéristiques stochastiques d'un comportement (on peut exprimer qu'un sous-comportement est « *x fois plus probable* » qu'un autre).

Au niveau de la simulation, [Rou04] a proposé un modèle d'exécution à base d'automates (*WTS* pour « *Weight-labelled Transition Systems* ») dans lequel il est possible d'exprimer des contraintes sur les transitions entre états (condition à laquelle la transition peut avoir lieu) ainsi qu'un rapport de probabilité entre deux transitions sortant d'un état. Les constructions principales sur ces WTSs sont

- un parallélisme top-level (« *un comportement est décrit par un ensemble d'automates concurrents, produisant chacun ses propres contraintes sur le comportement global* ») ;
- des états transitoires qui sont utiles pour améliorer la structuration des descriptions ;
- des poids infinis qui permettent de décrire qu'une transition de l'automate est plus probable qu'une autre sans préciser de rapport de probabilité exact entre les deux.

Pour la suite de la présentation, nous oublierons les informations stochastiques (poids sur les transitions) qui ne présentent pas d'intérêt pour nous. Les automates Lucky manipulent des variables V parmi lesquelles on distingue les entrées I, les sorties O et les variables locales L. Pour une variable v de V, v dénote la valeur de la variable à l'instant courant, $\bullet v$ dénote la valeur à l'instant précédent.

Un programme est décrit par un n-uplet :

$$(I, O, L, Q, J \subseteq Q, U \subseteq Q \times R(V, \bullet V) \times Q, C(V)).$$

Q est l'ensemble des états de l'automate, J dénote l'ensemble des états initiaux (le programme est non-déterministe, il peut donc y avoir plusieurs états initiaux). T est la relation de transition de l'automate, C représente une contrainte sur les valeurs initiales des variables du programme. Les transitions sont éventuellement pondérées par une valeur qui représente la probabilité relative de choisir une transition parmi plusieurs possibles à partir d'un état donné.

3.4.5.1 Sémantique

Comme nous l'avons fait pour les autres formes, nous donnons une sémantique d'un automate $Lucky = (I, O, L, Q, J \subseteq Q, U \subseteq Q \times R(V, \bullet V) \times Q, C(V))$ en terme de traces sur les variables du programme ainsi que sur ses états (traces de type τ_{loc}). On définit $V = I \cup O \cup L$.

$$\begin{aligned} \tau_{loc}(Lucky) = \quad & \{t \in T(V \cup Q) \mid \\ & t\lfloor V \rfloor[0] \in C \\ & \forall k \in [0..|t|-1].\ \exists (q_k, \phi, q_{k+1}) \in U \mid \\ & \phi(t[k+1], t[k])\} \end{aligned}$$

On peut ensuite donner une sémantique en terme de traces sur les entrées/sorties du composants décrit par l'automate Lucky, en masquant les variables d'états Q et les variables locales L :

$$\tau(Lucky) = \tau_{loc}(Lucky)\lfloor I \cup O \rfloor.$$

3.4.5.2 Propriétés

Les automates Lucky peuvent représenter des comportements déterministes ou non. Ces comportements sont toujours réactifs. L'utilisation principale est la description des environnements des systèmes dans un but de simulation.

3.4.6 Step-relations

Les step-relations permettent de décrire « pas à pas » l'évolution des variables d'un composant. On définit une relation entre la valeur des variables à l'instant t et leur valeur à l'instant suivant $t+1$.

Définition 13 — Step-relation

Une step-relation sur V relie des valuations entre elles. C'est donc un sous-ensemble de $\mathsf{Vals}(V,D) \times \mathsf{Vals}(V,D)$. *L'ensemble* $\mathsf{Step\text{-}Rels}(V,D)$ *des step-relations sur V est défini par :* $\mathsf{Step\text{-}Rels}(V,D) = \mathcal{P}(\mathsf{Vals}(V,D) \times \mathsf{Vals}(V,D))$.

EXEMPLE 16 — Pour noter de telles relations, on peut par exemple utiliser un langage de contraintes simple. Étant donné un ensemble de variables V, $v \in V$ la notation v représente la valeur de v à un instant donné et v' représente la valeur de v à l'instant suivant. La step-relation $v' = v$ décrit par exemple un composant où la valeur de la variable v est maintenue constante tout au long de l'exécution du composant. Pour $v = v' + 1$, v croît exactement de 1 à chaque instant. Enfin, pour $v' > v$, on spécifie que v croît strictement au cours du temps. Les deux premières step-relations décrivent des comportement déterministes, alors que le comportement de v dans $v' > v$ est non-déterministe.

— FIN DE L'EXEMPLE 16

Définition 14 — Relation combinatoire

Une relation combinatoire i *est un ensemble de valuations dans* $\mathsf{Vals}(V,D)$. *L'ensemble des relations combinatoires sur V est* $\mathsf{Comb\text{-}Rels}(V,D) = \mathcal{P}(\mathsf{Vals}(V,D))$.

Les relations combinatoires sont utilisées pour décrire les valeurs d'ensembles de variables à un instant donné (par exemple à l'initialisation d'une exécution). De manière générale le comportement d'un composant qui manipule des variables V peut être décrit par un couple (r,i) où $i \in \mathsf{Comb\text{-}Rels}(V,D)$ décrit les valeurs initiales des variables de V et $r \in \mathsf{Step\text{-}Rels}(V,D)$ décrit leur évolution pas-à-pas après l'instant initial.

EXEMPLE 17 — Un composant synchrone peut-être décrit comme un quintuplet $NStepRel = (I,O,L,r,c)$ où $r \in \mathsf{Step\text{-}Rels}(I \cup O \cup L, D)$ et $c \in \mathsf{Comb\text{-}Rels}(V,D)$. Le composant C que nous avons introduit plus haut peut être décrit à l'aide de :

$$(\{I1, I2, O\},\ r\ :\ I1' \leq O' < I2,\ c\ :\ I1 \leq O)$$

— FIN DE L'EXEMPLE 17

3.4.6.1 Sémantique

Comme nous l'avons fait pour les autres formes de spécification, nous donnons à présent la sémantique d'un couple (r,c) (où r est une step-relation sur $(I \cup O \cup L)$ et c est une relation combinatoire sur $(I \cup O \cup L)$) en terme d'ensemble de traces sur les variables de $I \cup O$.

$$\begin{aligned}\tau_{StepRel}(r,c) = \quad & \{t \in T(I \cup O) \mid \\ & \forall t' \in T(I \cup O \cup L) \text{ tel que } t' \lfloor I \cup O \rfloor = t. \\ & t'[0] \in c \\ & \wedge \forall n \in [1..|t'|-1].\ (t'[n-1], t'[n]) \in r\}\end{aligned}$$

Les traces de $\tau_{StepRel}(r,c)$ sont telles que :

- leur première valuation satisfait la contrainte initiale donnée par c et ;
- tout couple de valuations successives satisfait la relation r.

3.4.6.2 Propriétés

Les step-relations sont la forme la plus générale pour représenter les comportements d'un composant synchrone. Elles peuvent être déterministes ou non, réactives ou non. Nous verrons plus loin la traduction de toute les formes de spécification vers des step-relations.

3.4.6.3 Notations

Dans le chapitre 5, nous décrirons les composants à contrats à l'aide de step-relations. Nous introduisons dans ce paragraphe quelques notations qui nous seront utiles à ce moment-là.

Nous avons défini plus haut la projection et l'extension d'ensembles de traces par rapport à un sur-ensemble ou un sous-ensemble des variables concernées. Par extension de notation, on confondra un couple step-relation/relation combinatoire (r, c) avec l'ensemble de traces qu'il décrit $\tau_{StepRel}(r, c)$. On pourra utiliser les notations $(r, c)\lfloor W \rfloor$ ou $(r, c)\lceil W \rceil$, $(r, c) \cap (r', c')$ ou encore $(r, c) \subseteq (r', c')$.

Par la suite nous allons décrire des itérations, qui sont des formes de composants où il faut composer plusieurs fois le même composant avec lui-même. Nous aurons alors besoin de pouvoir renommer des relations (cette définition s'applique aussi bien aux step-relations qu'aux relations combinatoires).

Définition 15 — Renommage d'une relation

> *Soit une relation r portant sur des variables d'un ensemble V. Soit un ensemble de variables W, de même taille que V, et en bijection avec V. On définit le renommage de r dans W, la relation notée $r_{[V \leftarrow W]}$ où toutes les occurrences des variables de V ont été remplacées par les variables correspondantes de W.*

3.5 Comparaison des formes de spécifications

Nous venons de présenter plusieurs formes de spécifications qui permettent de décrire des composants synchrones. Nous nous intéressons maintenant aux différentes correspondances entre ces formes. Sont-elles équivalentes entre elles ? Est-il possible de traduire un programme décrit dans une forme donnée en un programme sémantiquement équivalent décrit dans une autre forme ?

3.5.1 Catégories de spécifications

Tout d'abord, il est possible de répartir les formes de spécifications selon deux critères. Le premier est le *déterminisme*. Les nœuds LUSTRE sont des programmes déterministes : a tout moment la valeur des sorties et des variables locales est entièrement déterminée par la valeur courante des entrées et l'histoire des mémoires du programme. Toutes les autres forment permettent de décrire des systèmes non-déterministes.

Le second critère est la distinction faite entre *générateurs* et *reconnaisseurs* de séquences de valeurs. Un *générateur* est un programme qui calcule des valeurs pour certaines variables (en l'occurrence les sorties O et des variables locales L) d'après les valeurs de ses entrées I. Cette catégorie comprend pour nous les nœuds LUSTRE ainsi que les nœuds à oracles ou tout autre forme dérivée. Un *reconnaisseur* est un programme qui reçoit des séquences de valeurs et détermine si ces séquences satisfont un critère particulier. Un observateur est un reconnaisseur associé à une propriété temporelle d'un programme.

Un troisième critère qui peut être avancé est celui d'exécutabilité d'une spécification. Dans la littérature, on trouve notamment deux travaux qui traitent de cet aspect. Dans [Hay89], Jones et Hayes avancent que les spécifications (notamment non-déterministes) ne devraient pas être exécutables, car cela rend leur description trop précise et donc trop difficile à utiliser. Un de leurs arguments est que

```
node N_COBS(I1,I2,O : int) returns (ok : bool);
let
    ok = O = N_C(I1,I2);
tel
```

FIG. 3.13 – Un observateur trivial obtenu à partir de N_C.

« les spécifications sont destinées à une consommation humaine [...]. Pour ce rôle les programmes donnent beaucoup trop de détails sur *comment* résoudre un problème, plutôt que de donner des détails sur *quel* problème doit être résolu ». Ce point de vue est intéressant car il prône une abstraction des spécifications par rapport aux implémentations, ce que nous recommandons également. En revanche, le fait de réserver les spécifications à un usage humain nous paraît trop contraignant. Si l'on permet d'exprimer des spécifications comme des programmes non-déterministes (dans un langage formellement défini), donc exécutable si l'on sait résoudre ce non-déterministe (comme par des tirages aléatoires des variables indéfinies) on peut envisager de manipuler ces programmes pour la simulation ou la validation. Nous rejoignons en cela les idées que Fuchs décrit dans [Fuc92]. Cela n'enlève rien à la nécessité d'abstraction des spécifications par rapport aux implémentations, mais élargies les possibilités d'utilisation des spécifications.

Ce critère d'exécutabilité peut être utilisé pour distinguer les différentes formes de spécifications. Les programmes LUSTRE sont exécutables puisque déterministes. Les autres formes peuvent aussi être considérées comme exécutables si l'on admet l'utilisation d'une certaine forme de résolution de contraintes, comme avancé dans [Fuc92]. Cette résolution de contraintes (qui a pour but de déterminer les valeurs des variables locales indéfinies, ou bien des variables de sorties dans le cadre des observateurs synchrones) est techniquement limitée (il existe de nombreuses formes de contraintes pour lesquelles la résolution est indécidable). A l'heure actuelle on sait, par exemple, simuler des programmes non-déterministes dont les comportements sont définis par des relations linéaires entre les variables (comme dans Lurette).

3.5.2 D'un nœud LUSTRE à un observateur

Il est assez évident de traduire un nœud LUSTRE N en un observateur N_{Obs} trivial qui code le fait que les sorties de N sont bien calculées par N. Prenons par exemple le nœud N_C de la figure 3.9. Nous pouvons construire l'observateur de la figure 3.13, qui code la propriété suivante : « la sortie O est toujours égale à la moyenne des entrées I1 et I2 ».

La transformation inverse n'est que rarement possible. Dans le cas où la sortie de l'observateur est calculée par une conjonction d'égalités chacune définissant une des sorties du composant, une analyse syntaxique de l'observateur peut permettre d'isoler ces égalités qu'on extrait comme équations définissant les sorties. Par exemple, l'observateur de la figure 3.13 peut être transformé en le nœud N_C. Dans le cas général, l'expression définissant ok est quelconque et l'on ne peut pas construire de nœud déterministe correspondant.

Ceci illustre la différence fondamentale entre reconnaisseurs et générateurs : il est toujours possible de traduire un générateur en un reconnaisseur en spécifiant que ce dernier doit reconnaître des valeurs qu'on peut spécifier à l'aide de *définitions* extraites directement du générateur. L'opération inverse est presque toujours impossible.

3.5.3 D'un nœud à oracle à un nœud à variables locales indéfinies

Comme nous l'avons souligné plus haut, il existe une transformation syntaxique entre les nœuds à variables locales indéfinies et les nœuds à oracles. Pour traduire le nœud $Noracle_C$ à partir du nœud $Nindet_C$ (et inversement), il suffit de changer la déclaration des variables indéfinies Ω.

3.5.4 D'un nœud à oracle à un observateur

δ On peut toujours écrire un observateur à partir d'un nœud à oracle. Il suffit pour cela de considérer les oracles comme des entrées de l'observateur. La transformation revient alors à celle entre un noeud LUSTRE et un observateur. Si on reprend le nœud à oracle de la figure 3.12, on peut facilement obtenir l'observateur suivant de la figure 3.14.

```
node Nobs_C(I1, I2, V, O : int)
returns (ok : bool);
var X : int;
let
   X = I1 + V;
   ok = (O = I1 -> if (X>=pre(I2))
                   then pre(I2)
                   else X)
tel
```

FIG. 3.14 – Observateur correspondant au nœud à oracle de la figure 3.12.

3.5.5 Vers les step-relations

Comme nous l'avons déjà indiqué plus haut, les step-relations restent la forme la plus générale pour décrire des ensembles de traces. Toutes les formes peuvent être traduites en step-relations.

EXEMPLE 18 — Tout composant C décrit sous la forme d'un nœud LUSTRE, d'un nœud à variable indéfinies, d'un nœud à oracle ou d'un observateur peut être traduit en un couple (c_C,r_C) où c_C est une relation combinatoire décrivant l'état initial du composant et r_C une step-relation décrivant son comportement au court du temps. Par exemple, le nœud N_C peut être décrit par le couple :

$$(c_{NC} \ : O = I1, r_{NC} \ : O' = (I1' + I2)/2)$$

Le composant C et l'observateur Obs_C sont décrits par le même couple :

$$(c_C \ : I1 <= O, r_C \ : I1' <= O' < I2)$$

Enfin, les programmes $Nindet_C$ et $Noracle_C$ sont décrit par :

$(c_C \ : I1 = O, r_C \ : (\text{ if } (X' >= I2) \text{ then } O' = I2 - 1 \text{ else } O' = X' \wedge X' = I1' + V')\lfloor I1, I2, O \rfloor)$

— **FIN DE L'EXEMPLE 18**

Un composant décrit par une step-relation s est un reconnaisseur. La step-relation décrit quelles valeurs des variables sont valides tout au long de l'évolution du comportement du composant. Elle

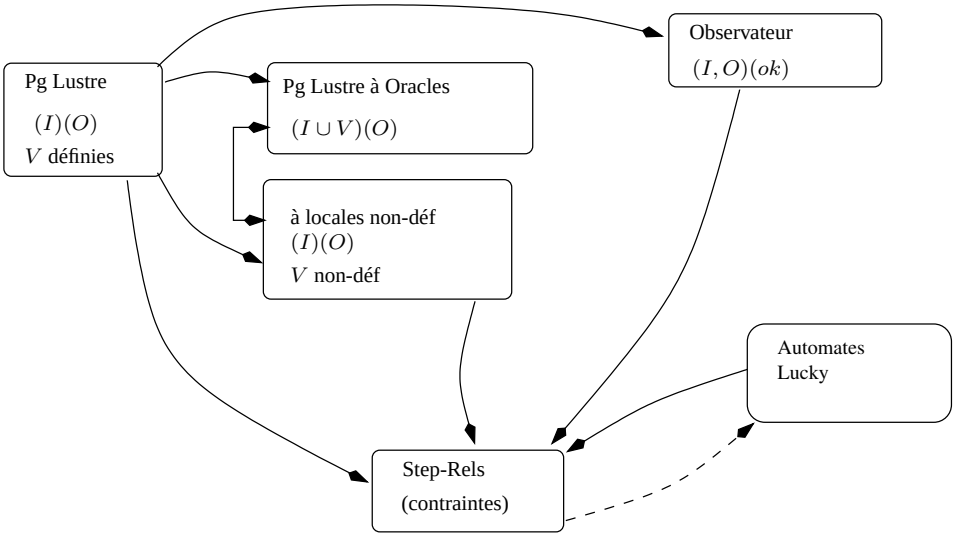


FIG. 3.15 – Les transformations possibles entre formes de spécification.

n'exprime pas forcément comme calculer *exactement* ces valeurs. On peut ainsi traduire un nœud LUSTRE, un observateur ou un nœud à oracles en une step-relation (augmentée d'une relation combinatoire décrivant l'état initial du composant). Pour des raisons identiques à celle évoquées dans le cas des observateurs, la transformation inverse n'est pas toujours possible.

3.5.6 Step-relations et automates Lucky

Les step-relations sont très proches des automates Lucky : les deux représentent des relations entre deux valeurs successives des variable du composant. En fait, les step-relations peuvent être vues comme une restriction des automates Lucky au cas où l'ensemble d'état Q ne contient qu'un seul élément. La traduction step-relations $\rightarrow$ Lucky est donc automatique.

Dans l'autre sens, et dans le cas où on ne considère pas les pondérations des transitions, on peut traduire un automate Lucky en une step-relation (accompagnée d'une relation combinatoire spécifiant l'état initial du programme) en encodant les états de l'automate par des variables booléennes.

3.5.7 Une vue générale

La figure 3.15 synthétise les transformations possibles entre les différentes formes de spécification. Les flèches pleines représentent les transformations toujours possibles. Les flèches en pointillés représentent les transformations qui ne sont pas toujours possibles.

3.6 Contrats

Un contrat est un couple de A et G qui décrit le comportement d'un composant portant sur ses variables d'entrée I et de sortie O. A spécifie une *assertion*, hypothèse que le composant fait sur son environnement et est décrit par un ensemble de traces sur I noté $\tau(A)$. G spécifie une propriété *garantie* sur les valeurs des sorties O dans le cas ou les entrées ne violent pas l'assertion A. G est décrit par un ensemble de traces sur $I \cup O$ noté $\tau(G)$. Nous considérerons plus loin qu'un contrat est associé à un nœud LUSTRE. Pour l'instant, et de manière générale (par exemple dans le cas où

le nœud associé au contrat n'est pas encore spécifié), nous pouvons étudier un contrat *en tant que tel* et donner sa sémantique. Le comportement d'un composant spécifié par un contrat (A, G) est un ensemble de traces sur $I \cup O$ dont chaque trace t est telle que :

- Soit la projection $t\lfloor I \rfloor$ de t sur I ne satisfait pas A (n'est pas une trace de $\tau(A)$) ;
- Soit la projection de t sur I satisfait A et alors t doit aussi satisfaire G.

La première condition signifie que toute entrée du composant doit satisfaire son assertion. La seconde signifie que pour toute histoire d'entrées qui satisfait l'assertion, le composant produit des sorties qui satisfont la garantie du contrat.

Le couple (A, G) *est* en lui-même une description du comportement du composant. Sa sémantique (en termes de traces sur $I \cup O$) est donnée par la définition suivante.

Définition 16 — Contrat

Le contrat (A, G) définit un ensemble de traces $\tau_{Contrat}(A, G)$ sur I, O par :

$$\begin{aligned}\tau_{Contrat}(A, G) = \quad & \{t \in T(I \cup O) | \\ & (t\lfloor I \rfloor \notin \tau(A)) \\ & \vee (t\lfloor I \rfloor \in \tau(A) \wedge t \in \tau(G))\}\end{aligned}$$

La spécification des clauses A et G d'un contrat peut se faire à l'aide de n'importe laquelle des formes que nous avons étudié plus haut. On verra plus loin qu'en pratique on les spécifie à l'aide d'observateurs, ou d'observateurs à variables locales indéfinies (utiles lors des compositions de contrats notamment). La sémantique qui en est donnée au chapitre 5 et les règles de composition associées sont exprimées à partir d'une définition de forme step-relation.

Le lecteur aura noté que l'assertion A ne porte pas sur les sorties du composant. Ce choix peut se justifier par la complexité de l'analyse des programmes impliquées par l'utilisation des sorties dans A et par le fait que cette utilisation n'est en fait pas indispensable. Nous reviendrons sur cette notion de contrat et sur celle de composant l'incluant au chapitre 5. Nous commenterons plus en détail dans ce même chapitre notre choix de ne pas utiliser les sorties dans A.

Partie II

Tableaux et contrats

Chapitre 4

Tableaux et itérateurs

Dans ce chapitre, nous montrons l'évolution de la partie du langage LUSTRE *qui permet de manipuler des tableaux. Dans un premier temps (paragraphe 4.1), nous rappelons brièvement les opérateurs tableaux introduits pour* LUSTRE-V4 *dans la thèse de Frédéric Rocheteau [Roc92]. Nous motivons ensuite, au paragraphe 4.2, l'introduction de nouveaux opérateurs appelés* itérateurs*, avant de les présenter (4.3) et de montrer les schémas de compilation (4.4) et d'optimisation (4.5) utilisés.*
L'introduction de tels opérateurs doit permettre de rendre plus claires les spécifications en en montrant les régularités. Par ailleurs, la nature même de ces opérateurs doit rendre évident le traitement des régularités de structure de programmes qu'ils impliquent, d'abord d'un point de vue de la compilation (c'est ce que décrit la section 4.2) mais aussi d'un point de vue validation (nous y reviendrons dans le chapitre 8).
Une version condensée de ce chapitre a été publié dans [Mor02].

4.1 Historique des tableaux en LUSTRE - Motivations

Les tableaux ont été introduits dans LUSTRE lors du travail de thèse de Frédérique Rocheteau [Roc92]. L'objectif général de ce travail était l'extension du langage dans un but de conception de circuits. L'introduction des opérateurs de traitements des tableaux n'avait pas pour but d'augmenter l'expressivité du langage, mais uniquement de « *simplifier l'écriture de programmes de grande taille* ». Les extensions apportées au langage sont :

- le type tableau ;
- des opérateurs de sélection (d'un élément ou d'une tranche) ;
- un mécanisme de *récursivité statique*.

Nous détaillons maintenant la syntaxe de ces opérations.

4.1.1 Type tableau

Si τ est un type et si n est une constante entière *connue statiquement*, alors $\tau\hat{}n$ est le type tableau d'éléments de type τ. Par exemple, les définitions suivantes sont acceptées pour déclarer un tableau de taille 42, un type registre, "tableau de booléens de taille 16" et une variable R du type registre.

```
T : intˆ42;
const TAILLE = 16;
type registre = boolˆTAILLE;
R : registre;
```

4.1.2 Accès aux éléments d'un tableau

Soit A un tableau de taille n. Ses éléments sont : A[0], A[1], ..., A[n-1]. L'expression A[e] est acceptée, du moment que e est une constante connue à la compilation dont la valeur est comprise entre 0 et n-1.

4.1.3 Expressions

On peut utiliser les constructeurs suivants :

- [0,2,3] représente le tableau d'éléments 0, 2 et 3 ;
- trueˆ3 = [true,true, true] ;
- constructions par tranches. On a, de manière générale :

$$A[i..j] = \begin{cases} [A[i], A[i+1], \ldots, A[j]] \text{ si } i \leq j \\ [A[i], A[i-1], \ldots, A[j]] \text{ si } j < i \end{cases} ;$$

- Concaténation : si A est de taille n et B de taille m alors $A|B = [A[0], A[1], \ldots, A[n-1], B[0], B[1], \ldots, B[m-1]]$.

Tous les opérateurs polymorphes de LUSTRE (if ... then ... else ..., pre, $\rightarrow$) s'appliquent au tableaux. Par exemple, on peut écrire :

$$A = \text{trueˆ4} \rightarrow \text{if cˆ4 then B[4..7] else pre(A);}$$

Le tableau A vaut [true,true,true,true] au premier instant et ensuite si c est vrai alors [B[4],B[5],B[6],B[7]] sinon [pre A[0],pre A[1],pre A[2],pre A[3]].

4.1.4 Généricité sur la taille

La taille d'un tableau T peut être un paramètre générique du nœud dans lequel T est défini, à condition que pour chaque appel du nœud, ce paramètre soit instancié par une constante statique.

EXEMPLE 19 — Considérons par exemple le nœud générique N déclaré comme suit :

```
node N(const n :int; A : intˆn) returns (B : intˆn);
```

On va pouvoir instancier N en donnant au paramètre n une valeur constante connue statiquement. Par exemple, on écrira :

```
const size = 10;
...
T' = N(size,T);
```

— FIN DE L'EXEMPLE 19

Grâce à ce mécanisme, on peut écrire des algorithmes manipulant des tableaux de taille quelconque et les instancier sur les tableaux que l'on manipule réellement. La généricité permet la *réutilisabilité* des programmes.

4.1.5 Récursivité statique

Il existe en LUSTRE un mécanisme de récursivité statique mis en place par l'opérateur with qui autorise l'appel d'un nœud par lui-même à condition que l'arrêt de la récursivité puisse être assuré statiquement.

EXEMPLE 20 — Imaginons que l'on veuille programmer un algorithme qui prenne en entrées 2 tableaux de n booléens A et B et qui rende une matrice carrée AB (de taille $n \times n$) définie par l'équation :

$$\forall i, j \in [0, n], AB[i, j] = A[i] \text{ and } B[j]$$

Le nœud LUSTRE correspondant (issu d'un programme calculant un produit d'entiers représentés par des vecteurs de booléens) est le suivant :

```
node prod(const n,i : int; a,b : bool^n) returns(ab : bool^n^(n-i));
let
   ab = with i=n-1
        then [b and a[i]^n]
        else [b and (a[i])^n]|prod(n,i+1,a,b);
tel
```

A chaque appel du nœud prod, i est incrémenté exactement de 1 (car on appelle prod(n,i+1,a,b)). La condition i=n-1 est alors forcément assurée lors des appels récursifs de prod. Lorsqu'elle l'est, la récursion s'arrête et ab prend la valeur de l'expression [b and a[i]^n]. La condition d'arrêt de la récursion peut donc être évaluée statiquement. Le compilateur accepte un programme utilisant l'opérateur with à la condition qu'il puisse déterminer la satisfaisabilité de la condition d'arrêt.

— FIN DE L'EXEMPLE 20

4.1.6 Exemple

La description de systèmes matériels grâce aux opérateurs définis ci-dessus est assez satisfaisante et élégante. On peut, par exemple, définir un additionneur N bits par le programme de la figure 4.1.

4.1.7 Avantages et inconvénients

Le compilateur V4 expanse les tableaux en variables indépendantes. Par exemple, on peut compiler le programme ADD avec la commande lus2ec et on obtient le code intermédiaire (format ec) de la figure 4.2. Remarquons que les appels de nœuds ont aussi été expansés (on n'a donc plus qu'un seul nœud LUSTRE).

Comme on peut l'observer à la figure 4.2, on a perdu la structuration des données sous forme de tableaux. A la place du tableau de booléens A (qui était de taille n), on a maintenant n variables booléennes indépendantes. Le code impératif (le compilateur produit du code en langage C) généré pour ce programme aura donc aussi des variables indépendantes à la place des tableaux.

Cette méthode est tout à fait adaptée à la génération de matériel puisqu'au bout du compte, une variable (ou un élément de tableau) sera représentée par un *fil* sur une carte et qu'on aura besoin d'une définition immédiate de chaque fil. Par ailleurs, cette approche nous permet d'utiliser les outils

```
const n=10;

node FULL_ADD(ci,a,b : bool) returns (co,s : bool);
let
   s = a xor (b xor ci);
   co = (a and b) xor (b and ci) xor (a and ci);
tel

node ADD(A,B : bool^n) returns (S : bool^n ; overflow : bool);
var CARRY : bool^n;
let
   (CARRY,S) = FULL_ADD([false] | CARRY[0..n-2],A,B);
   overflow = CARRY[n-1];
tel
```

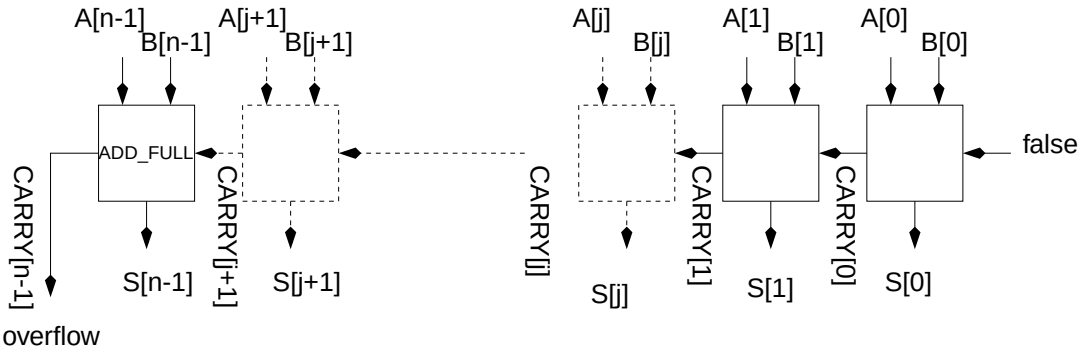


FIG. 4.1 – Un additionneur n bits.

```
node ADD (A_0: bool; A_1: bool; A_2: bool; A_3: bool; A_4: bool;
          A_5: bool; A_6: bool; A_7: bool; A_8: bool; A_9: bool;
          B_0: bool; B_1: bool; B_2: bool; B_3: bool; B_4: bool;
          B_5: bool; B_6: bool; B_7: bool; B_8: bool; B_9: bool)
returns (S_0: bool; S_1: bool; S_2: bool; S_3: bool; S_4: bool;
         S_5: bool; S_6: bool; S_7: bool; S_8: bool; S_9: bool;
         overflow: bool);
var
    V59_CARRY_0: bool; V60_CARRY_1: bool; V61_CARRY_2: bool;
    V62_CARRY_3: bool; V63_CARRY_4: bool; V64_CARRY_5: bool;
    V65_CARRY_6: bool; V66_CARRY_7: bool; V67_CARRY_8: bool;
let
    S_0 = A_0 xor B_0 xor false;
    S_1 = A_1 xor B_1 xor V59_CARRY_0;
    ...
    S_9 = A_9 xor B_9 xor V67_CARRY_8;
    overflow = (A_9 and B_9) xor (B_9 and V67_CARRY_8)
               xor (A_9 and V67_CARRY_8);
    V59_CARRY_0 = (A_0 and B_0) xor (B_0 and false)
                  xor (A_0 and false);
    V60_CARRY_1 = (A_1 and B_1) xor (B_1 and V59_CARRY_0)
                  xor (A_1 and V59_CARRY_0);
    ...
    V67_CARRY_8 = (A_8 and B_8) xor (B_8 and V66_CARRY_7)
                  xor (A_8 and V66_CARRY_7);
tel
```

FIG. 4.2 – Code LUSTRE intermédiaire produit pour le programme ADD.

de validation LUSTRE (dans la limite liée à l'explosion combinatoire impliquée par l'utilisation de tableaux de grande taille) sans avoir à augmenter ceux-ci avec les opérateurs tableaux.

Mais pour la génération de code *logiciel*, l'expansion de code est inutile et le code obtenu n'est pas satisfaisant. En effet, il est :

- lent. LUSTRE est conçu pour programmer des systèmes *réactifs* dont le temps de réponse est, nous l'avons vu plus haut, imposé par l'environnement. En expansant un tableau, on obtient dans le code produit, autant d'affectations qu'il y a d'éléments dans le tableau, et donc autant d'accès mémoire. En conservant une structure de tableaux, avec boucles de parcours, on peut n'obtenir qu'un seul accès mémoire pour le début du tableau plus des décalages plus rapides à effectuer pour chacun des éléments ;
- trop gros, et c'est le point le plus important. LUSTRE est conçu pour programmer des systèmes *embarqués* pour lesquels la taille mémoire est souvent limitée. La manipulation de tableaux, dans le code impératif produit, permettrait l'utilisation de boucles les parcourant, ce qui réduirait considérablement la taille du code. On pourrait compiler le programme ADD en un programme C comme celui de la figure 4.3[1].

```
for(i=0;i<n;i++){
    S[i] = A[i] xor (B[i] xor C[i]);
    CARRY[i] = (A[i] && B[i])
                    || (B[i] && CARRY[i-1])
                    || (A[i] && CARRY[i-1]);
}
overflow = CARRY[n-1];
```

FIG. 4.3 – Boucle for souhaitable comme traduction d'un algorithme tableau.

```
X[0] = Y[0];
X[1..2] = Y[4..5];
X[3..5] = Y[1..3];
```

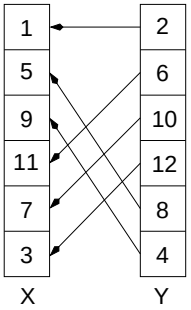


FIG. 4.4 – Calculs de tableaux par tranches.

Une approche possible consiste à tenter de générer du code à boucle à partir des opérateurs de LUSTRE-V4. Mais la possibilité de définition « arbitraire » d'éléments ou de tranches de tableaux (comme à la figure 4.4) rend difficile cette transformation.

D'autre part, lorsqu'on souhaite développer un système logiciel manipulant des tableaux, on a souvent en tête des algorithmes sur les tableaux utilisant des boucles dont l'écriture avec les opérateurs

[1]Nous n'avons pas décrit la boucle temporelle inhérente à tout programme réactif

V4 n'est pas aisée (une solution consiste à transformer l'algorithme original itératif en un algorithme récursif que l'on peut décrire avec l'opérateur with).

4.2 Vers des itérateurs de tableaux

Pour résoudre les problèmes évoqués ci-dessus, nous proposons d'introduire des itérateurs de tableaux dans le langage. Ces opérateurs sont largement inspirés de ceux que l'on trouve pour la manipulation des listes dans les langages fonctionnels. Il s'agit du map, du red, du fill et du map_red, décrits au paragraphe 4.3. Des méthodes de compilation adaptées permettent de générer du code C avec tableaux et boucles. Les avantages attendus sont les suivants :

- **Taille du code C produit :** dans tous les cas où l'on applique n fois un calcul Calc (potentiellement identifié comme un nœud), on réduit n copies du code de Calc à une seule (voir exemple ADD ci-dessus). Le code sans itérateurs de tableaux est beaucoup plus gros, aussi bien en C que dans tout assembleur que l'on obtient par compilation du programme C.
- **Temps d'exécution du code C produit :** considérons un programme C $P1$ constitué de n affectations explicites en séquence ; un programme C $P2$ constitué d'une boucle comportant une seule affectation ; la boucle est exécutée n fois. $P1$ est en général un peu plus lent que $P2$. Cela dépend beaucoup du processeur cible, et des optimisations du compilateur C (dans n affectations, il y a n calculs d'adresses en mémoire, alors que dans une boucle d'accès à un tableau, le code assembleur produit peut en général ne contenir qu'un vrai calcul d'adresse, plus des décalages plus rapides à effectuer).
- **Taille mémoire nécessaire à l'exécution du code C produit :** il n'y a pas de différence intrinsèque entre n variables et un tableau de taille n. Toutefois, si les traitements de tableaux sont exprimés en itérateurs, il est possible de repérer les cas de variables intermédiaires inutiles. Par exemple, deux « map » enchaînés nécessitent un tableau intermédiaire (le résultat du premier map et la donnée du second). On sait transformer un tel programme, en gardant son comportement intact, de manière à faire disparaître cette structure de donnée intermédiaire (voir sec 4.5). On obtient un gain de place et un certain gain en temps (il faut bien affecter et relire chacun des éléments de cette structure intermédiaire, quand elle est présente).
- **Généricité sur la taille :** les programmes sont naturellement écrits sous forme de manipulations de tableaux de taille n, et il n'est pas indispensable de connaître la valeur de n pour écrire le programme. Il faut simplement fixer une valeur pour obtenir un programme exécutable.
- **« structuration », prouvabilité :** Identifier explicitement des tableaux partout où un ensemble de données vont être traitées de manière très similaire (même s'il y a quelques exceptions), permet de mieux structurer les spécifications. Cela a pour conséquences entre autres : une meilleure lisibilité du code, une meilleure prise en compte de la structure par les outils avals comme un simulateur, un débogueur ou des outils de preuve. Nous étudions ces possibilités au chapitre 8.

4.3 Itérateurs de tableaux - Syntaxe et sémantique

Nous présentons maintenant les itérateurs de tableaux. Une méthode de compilation de ces itérateurs pour générer du code impératif à boucle for comme vu plus haut est proposée ainsi qu'un ensemble d'optimisations basées sur la suppression de tableaux intermédiaires lors d'enchaînements d'itérations.

Nous donnons ici une description syntaxique et sémantique des 4 *itérateurs* LUSTRE. Cette description est celle présentée dans [Mor02]. Pour chaque itérateur, nous donnons une description indé-

pendante de la syntaxe qui manipule non pas des nœuds mais des fonctions purement mathématiques. Cette notation sera aussi utilisée pour présenter les axiomes d'optimisation de la section 4.5. Puis le lecteur trouvera la syntaxe et la sémantique de l'implémentation LUSTRE. Enfin, nous donnons, pour chaque cas, le code C qui devrait être généré pour l'équation donnée en exemple.

Notations – Par la suite, n dénote un entier dont la valeur doit être, en LUSTRE, connue statiquement.T et T' dénotent des tableaux de taille n. Les τ dénotent n'importe quel type LUSTRE. Rappelons que si τ dénote un type quelconque, alors τ^n est le type tableau de τ. Nous manipulons des fonctions sous la forme de λ-termes à plusieurs paramètres. Par exemple, soit le λ-terme suivant : $\lambda x \lambda y.x + y$. Pour représenter un tel terme, nous écrirons, $\lambda x, y.x+y$, notation qui correspond à la syntaxe LUSTRE. *Par extension un nœud* LUSTRE *(même s'il n'est pas purement combinatoire) est représenté à l'aide d'un λ-terme*. Si un nœud possède plusieurs sorties, on les sépare par une virgule dans l'expression du λ-terme associé. Par exemple, $\lambda x, y.x + y, x * y$ dénote un nœud prenant deux entrées x et y et produisant deux sorties, l'une valant $x + y$ et l'autre $x * y$.

Soit un nœud N. On pourra noter par N(I) les sorties produites par N lorsqu'on l'applique aux entrées I. Les itérations sont notées comme des nœuds qui prennent toujours pour premier paramètre le nœud itéré.

Si l'on souhaite distinguer les sorties produites par un nœud on peut le faire en utilisant la notation « ‘ ». Par exemple, N(I)‘o dénote la valeur de la variable de sortie o obtenue par application du nœud N aux entrées I.

4.3.1 map

L'opération de mapping consiste à appliquer la même fonction g $= \lambda t.t'$ à tous les éléments d'un ou plusieurs tableaux (T, dans notre exemple). Les résultats sont les éléments d'autres tableaux (T'). La syntaxe abstraite de l'opérateur map est la suivante :

$$T' = map(g, T).$$

Les syntaxe et sémantique exactes LUSTRE sont données ci-dessous. Nous ne parlons plus de fonction, mais de nœud ou d'opérateur. Si N (resp., O) est un nœud LUSTRE (resp., un opérateur) de profil :

$$\tau_1 \times \tau_2 \times \ldots \times \tau_l \rightarrow \tau'_1 \times \tau'_2 \times \ldots \times \tau'_k,$$

Et si n est une *constante statique*, alors map≪N;n≫ (resp. map≪O;n≫) est considéré comme un nœud (resp. un opérateur) [2] de profil :

$$\tau_1\hat{}\text{n} \times \tau_2\hat{}\text{n} \times \ldots \times \tau_l\hat{}\text{n} \rightarrow \tau'_1\hat{}\text{n} \times \tau'_2\hat{}\text{n} \times \ldots \times \tau'_l\hat{}\text{n}.$$

Le schéma 4.5 montre un map simple, avec *un* tableau en entrée et *un* tableau en sortie. Le schéma 4.6 montre plus en détail le nœud mappé.

4.3.1.1 Exemple

Considérons le nœud N suivant :

[2] Dorénavant, nous ne ferons plus la différence entre nœud et opérateur

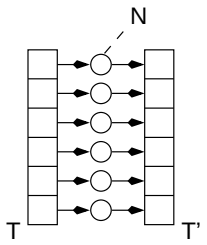


FIG. 4.5 – L'opération map.

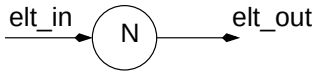


FIG. 4.6 – Le nœud itéré.

```
node N(elt_in : int) returns (elt_out : int);
let
    elt_out = elt_in -> elt_in + pre(elt_out);
tel
```

L'itération de ce nœud grâce à un map à l'intérieur d'un nœud test (ci-dessous) permet de calculer le tableau T' tel que $\forall i \in [0..n-1].\ T'[i] = N(T[i])$.

```
node test(T : int^n) returns (T' : int^n);
let
    T' = map<<N;n>>(T);
tel
```

4.3.1.2 Code séquentiel correspondant

Le code que l'on produit pour l'équation LUSTRE T' = map ≪N,n≫(T) est :

```
for(i=0;i<n;i++){
    T'[i] = N(T[i]);
}
```

4.3.2 red

La réduction, fréquemment appelée *foldl* ou *foldr* [GLP93], permet de calculer un accumulateur en parcourant un ou plusieurs tableau(x). Soit la fonction $g = \lambda t, acc.acc'$. La réduction res d'un tableau T à l'aide de cette fonction peut être calculée de la manière suivante :

$$res = red(init, T, g),$$

où init est une expression d'initialisation de la réduction. En LUSTRE, on utilise l'opérateur red qui a la syntaxe suivante. Si N est un nœud de profil :

$$\tau \times \tau_1 \times \tau_2 \times \ldots \times \tau_l \rightarrow \tau'$$

et si n est une constante statique, alors red≪N;n≫ est un nœud de profil :

$$\tau \times \tau_1\hat{}\mathrm{n} \times \tau_2\hat{}\mathrm{n} \times \ldots \times \tau_l\hat{}\mathrm{n} \rightarrow \tau'$$

Le schéma 4.7 montre un red simple, avec *un* tableau en entrée. Le schéma 4.8 montre plus en détail le nœud itéré.

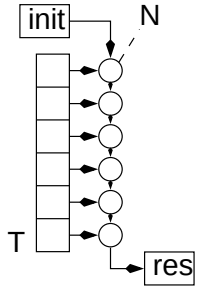


FIG. 4.7 – L'opération de réduction.

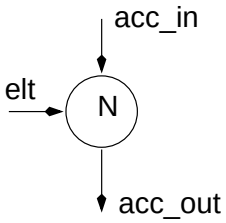


FIG. 4.8 – Le nœud itéré.

4.3.2.1 Exemple

Voici un nœud qui calcule la somme de ses 2 entrées entières:

```
node plus(acc_in, elt : int) returns (acc_out : int);
let
   acc_out = acc_in + elt;
tel
```

L'itération de ce nœud avec l'opérateur red permet de calculer la somme des éléments d'un tableau :

```
node redPlus(T : int^n) returns (res : int');
let
    res = red<<plus;n>>(0,T);
tel
```

4.3.2.2 Code séquentiel correspondant

Le code que l'on produit pour l'équation LUSTRE T' = red ≪plus,n≫(0,T) est :

```
int res = 0;
for(i=0;i<n;i++){
   res = plus(res,T[i]);
}
```

Remarque 6 (Concernant le nœud itéré) *Un nœud itéré par un* red *aura toujours la structure suivante :*

- *Au moins 2 entrées : la première correspondant à l'accumulateur, les suivantes correspondant aux éléments des tableaux donnés en paramètre au* red *;*
- *Toujours une seule sortie : celle correspondant à l'accumulateur.*

D'autre part, on doit s'assurer que les horloges des accumulateurs de sortie et d'entrée du nœud itéré sont compatibles puisque ces variables se retrouvent « branchées » entre deux instances du nœud itéré.

L'unicité de l'accumulateur ne limite en rien les possibilités offertes : en effet, on peut utiliser des accumulateurs à type structuré, avec autant de champs que désiré. Nous verrons cela dans les exemples en 4.3.5.

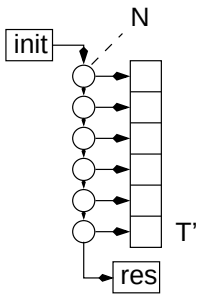


FIG. 4.9 – L'opération de remplissage.

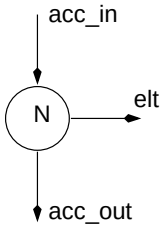


FIG. 4.10 – Détail.

4.3.3 fill

L'opérateur fill permet de remplir un ou plusieurs tableaux et de définir l'accumulateur de sortie correspondant. Cet opération consiste à calculer itérativement la valeur des éléments du ou des tableaux à remplir, en prenant en compte une valeur initiale. Le i-ème élément du tableau est donc le résultat de la fonction itérée, appliquée i fois à la valeur initiale de l'« accumulateur ».

Soit la fonction g= $\lambda acc.acc', elt$. Le tableau T peut être rempli et l'accumulateur de sortie res défini avec l'équation suivante :

$$res, T = fill(init, g),$$

où a est une expression d'initialisation du remplissage. En LUSTRE, on utilise l'opérateur fill qui a la syntaxe suivante : si N est un nœud de profil :

$$\tau \rightarrow \tau \times \tau'_1 \times \tau'_2 \times \ldots \times \tau_l$$

et si n est une constante statique, alors fill≪N;n ≫ est un nœud de profil :

$$\tau \rightarrow \tau \times \tau'_1\hat{}\mathrm{n} \times \tau'_2\hat{}\mathrm{n} \times \ldots \times \tau_l\hat{}\mathrm{n}$$

Le schéma 4.9 montre un fill simple, avec *un* tableau en sortie. Le schéma 4.10 montre plus en détail le nœud itéré. Le code du programme correspondant est donné ci-dessous.

4.3.3.1 Exemple

Considérons le nœud N suivant :

```
node N(acc_in : int) returns (acc_out : int ; elt : int);
let
   acc_out = acc_in + 1;
   elt = acc_in;
tel
```

L'itération de ce nœud par un fill initialisé à 0 permet de calculer un tableau de taille n qui contient la suite des entiers [0,..,n-1]. Le nœud suivant test réalise ce calcul.

```
node test(init : int) returns (T : int^n);
var res : int;
let
   res, T = fill≪N;n≫(0);
tel
```

4.3.3.2 Code séquentiel correspondant

Le code que l'on produit pour l'équation LUSTRE res,T = fill ≪N,n≫(init) est :

```
res = init;
for(i=0;i<n;i++){
   res,T[i] = N(res);
}
```

Notons l'utilisation dans le code généré d'une variable res non définie dans le programme LUSTRE qui sert d'accumulateur durant le remplissage.

Remarque 7 (Concernant le nœud itéré) *Comme pour le* red*, le nœud utilisé par un* fill *doit respecter certaines contraintes :*

- *Une seule entrée : celle correspondant à l'accumulateur ;*
- *Au moins 2 sorties : la première correspondant à l'accumulateur, les suivantes correspondant aux éléments des tableaux que l'on remplit par le* fill*.*

Les contraintes d'horloges pour les accumulateurs d'entrée et de sortie doivent être respectées.

4.3.4 map_red

L'opération de map_red constitue l'opérateur général duquel tous les autres peuvent être déduits. Si T et T' sont deux tableaux et si g est une fonction définit par : g $= \lambda acc, t.acc', t'$, alors on peut avoir :

$$(res, T') = map\_red(init, T, g)$$

En LUSTRE, les syntaxe et sémantique exactes sont données ci-dessous. Si N est un nœud lustre :

$$\tau \times \tau_1 \times \tau_2 \times \ldots \times \tau_l \rightarrow \tau \times \tau'_1 \times \tau'_2 \times \ldots \times \tau'_k,$$

et si n est une *constante statique*, alors map_red≪N;n≫ est considéré comme un nœud de profil :

$$\tau \times \tau_1\hat{}\mathrm{n} \times \tau_2\hat{}\mathrm{n} \times \ldots \times \tau_l\hat{}\mathrm{n} \rightarrow \tau \times \tau'_1\hat{}\mathrm{n} \times \tau'_2\hat{}\mathrm{n} \times \ldots \times \tau'_l\hat{}\mathrm{n}.$$

Le schéma 4.11 montre un map_red simple, avec *un* tableau en entrée et *un* tableau en sortie. Le schéma 4.12 montre plus en détail le nœud itéré.

4.3.4.1 Exemple

Considérons le nœud sumCumul suivant :

```
node sumCumul(acc_in, t_in : int) returns (acc_out, t_out : int);
let
    t_out = acc_in;
    acc_out = acc_in + t_in;
tel
```

L'itération de ce nœud grâce à un map_red à l'intérieur du nœud suivant test permet de calculer les sommes cumulées des i premiers éléments du tableau T. On a $\forall i \in [0, .., n-1].T'[i] = init + \Sigma_{k=0}^{i-1} T[k]$.

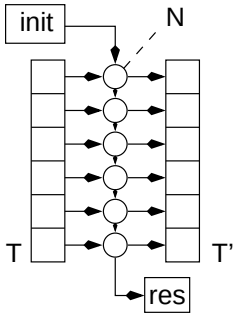


FIG. 4.11 – L'opération map_red.

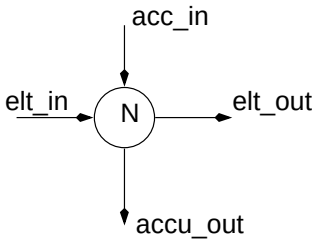


FIG. 4.12 – Détail.

```
node test(init : int ; T : int^n)
returns (res : int ; T' : int^n);
let
   res,T' = map_red≪sumCumul;n≫(init,T);
tel
```

4.3.4.2 Code séquentiel correspondant

Le code que l'on produit pour l'équation LUSTRE T' = map_red ≪N,n≫(T) est le suivant :

```
acc_out = acc_in;
for(i=0;i<n;i++){
   acc_out,T'[i] = N(acc_out,T[i]);
}
```

Remarque 8 (Concernant le nœud itéré) *Au moins 2 entrées et 2 sorties :*

- *la première entrée et la première sortie correspondent à l'accumulateur ;*
- *les autres entrées/sorties correspondent aux éléments des tableaux consommés/produits par l'itération.*

Les contraintes d'horloges pour les accumulateurs d'entrée et de sortie doivent être respectées.

4.3.5 Exemples

4.3.5.1 Sélection d'un élément par son indice

Comme nous l'avons dit plus haut, il est impossible en LUSTRE de sélectionner un élément d'un tableau directement par son indice si celui-ci est donné par une expression dont la valeur n'est pas connue statiquement.

Nous proposons, comme premier exemple d'utilisation des itérations, de coder un programme qui sélectionne le i-ème élément d'un tableau array, la valeur de i étant donnée par une expression quelconque. Lorsque la valeur de i n'est pas valide (hors des bornes du tableau) le programme retourne une *valeur par défaut* (une constante statique dans notre cas, mais on pourrait choisir de renvoyer

par exemple la valeur du premier élément de array). Les éléments du tableau array sont d'un type quelconque elementType.

L'accumulateur de l'itération est une variable de type iteratedStruct :

```
type iteratedStruct = {currentRank : int;
                       rankToSelect : int;
                       elementSelected : elementType};
```

À chaque étage de l'itération, les informations contenues dans l'accumulateur représentent :

- le rang de l'élément courant (initialisé à 0). C'est le champ currentRank de la structure qui est incrémenté de 1 à chaque étage de l'itération ;
- le rang de l'élément à sélectionner (initialisé à rankToSelect). Ce champ n'est jamais modifié par l'itération ;
- la valeur de l'élément sélectionné, nommé elementSelected. La valeur de ce champ ne sera significative qu'à partir du moment où l'itération aura atteint le i-ème élément du tableau. Si i n'est pas valide, elementSelected prend une valeur par défaut donnée dans l'initialisation de l'itération (ici la constante 0).

Un étage de l'itération est décrit à l'aide du nœud selectOneStage donné ci-dessous.

```
node selectOneStage(acc_in : iteratedStruct;
                    currentElt : int)
returns (acc_out : iteratedStruct)
let
   acc_out = {currentRank = acc_in.currentRank+1;
              rankToSelect = acc_in.rankToSelect;
              elementSelected = if(acc_in.currentRank=acc_in.rankToSelect)
                                then currentElt
                                else acc_in.elementSelected};
tel
```

Pour déterminer la valeur du i-ème élément, on itère le nœud selectOneStage sur le tableau array. On calcule ainsi une variable de type iteratedStruct. La valeur de l'élément sélectionné est obtenue très simplement avec l'expression iterationResult.elementSelected. Le nœud selectElementOfRank_inArray_ réalisant la sélection est donné ci-dessous :

```
node selectElementOfRank_inArray_(i : int;
                                  array : arrayType)
returns (elementSelected : elementType)
var iterationResult : iteratedStruct;
let
   iterationResult = red≪selectOneStage;size≫({currentRank = 0,
                                               rankToSelect = rankToSelect,
                                               elementSelected = 0},
                                              array);
   elementSelected = iterationResult.elementSelected;
tel
```

4.3.5.2 Mémoire à n instants

Nous décrivons maintenant un programme qui, à partir d'une variable booléenne x construit un tableau Tab de taille n qui contient les valeurs de x aux n derniers instants. Si on note t l'instant courant, pour toute valeur i comprise dans [0..n-1], T[i] contient la valeur qu'avait x à l'instant t - i. Nous utiliserons ce programme au paragraphe 8.2.3.1 pour faciliter l'expression d'une propriété portant sur les n dernières valeurs d'une variable. On pourrait décrire une telle mémoire pour n'importe quel type de données.

```
node memoire1(acc_in : bool) returns (acc_out, elt : bool);
let
   elt = true -> pre(acc_in);
   acc_out = true -> pre(acc_in);
tel

node memoireNInstants(x : bool) returns (Tab : bool^10);
var acc_out : bool;
let
   acc_out, Tab = fill<<memoire1;10>>(x);
tel
```

Le fait d'utiliser une taille statique des itérations implique ici que la taille mémoire nécessaire pour ce programme à l'éxecution est statique.

4.4 Compilation

Nous présentons maintenant le principe général de l'algorithme de compilation des itérations en du code séquentiel avec boucles. Nous donnons ensuite quelques exemples et indiquons comment ce principe s'étend à la compilation d'itérations sur des tableaux à plusieurs dimensions. L'algorithme de génération de code est présenté en annexe, chapitre A.

4.4.1 Principe général

L'objectif est de proposer un schéma de compilation pour traduire des programmes LUSTRE avec itérateurs en du code C avec boucle et tableaux. Nous ne nous sommes pas intéressé à des problèmes de vérification statique habituellement nécessaires (voir 2.2.2.1). Nous supposons donc que le programme LUSTRE à compiler est syntaxiquement correct et qu'il ne contient pas d'expressions sémantiquement incorrectes du point de vue de la manipulation des types. Nous supposons aussi que les appels de nœuds sont expansés, ce qui est effectivement le cas en pratique, un algorithme étant appliqué en amont de la génération de code pour traiter ce problème. Lors de la génération de code, on ne parcourt qu'un seul nœud (le nœud principal). Les seuls autres nœuds que nous inspectons sont les nœuds itérés dans le nœud principal.

La boucle de réactivité habituellement nécessaire à l'exécution du programme n'est pas produite. Nous avons choisi ce processus de compilation simplifié pour étudier directement les problèmes de générations de code avec boucles et tableaux.

La génération de code se fait en deux phases distinctes. Dans un premier temps, on génère le code correspondant au calcul des variables locales et des sorties du programme. Ce code comprend pour chaque itération une boucle de la forme décrite dans les paragraphes précédents. La seconde phase

correspond à la génération du code de mise à jour des mémoires. Pour chaque mémoire (occurrence de l'opérateur pre) utilisée dans un nœud itéré, on génère une boucle for de mise à jour. On doit aussi générer le code de mise à jour des mémoires utilisées directement dans le nœud principal.

L'algorithme de génération de code est donné dans l'annexe A. Nous donnons ci-dessous quelques exemples qui permettent de mettre en évidence notamment les deux phases que nous venons de présenter.

4.4.2 Quelques exemples

Nous donnons ci-dessous des exemples de programmes LUSTRE et leur traduction en code impératif avec boucle pour les itérations. Le premier exemple est purement combinatoire (il s'agit de la version itérateur de l'additionneur que nous avons montré plus haut). Le second est temporel : l'opérateur pre est utilisé dans le nœud itéré ainsi que dans le nœud utilisant l'itération. Enfin, un troisième exemple est donné, illustrant la génération de code pour des itérations de tableaux à plusieurs dimensions.

4.4.2.1 Additionneur n-bits

Soit le programme ADD décrit plus haut. On donne à la figure 4.13 une version utilisant l'itérateur map_red. Le code séquentiel produit pour ce programme est donné à la figure 4.14. Ce programme n'utilise pas de mémoire, il n'est donc pas nécessaire de distinguer l'instant initial. L'algorithme donné à l'annexe A générerait deux parties distinctes (init et step) pour ce programme, même si ces deux parties sont strictement identiques.

```
node FULL_ADD(ci,a,b : bool) returns (co,s : bool);
let
   co = (a and b) or (b and ci) or (a and ci);
   s = a xor (b xor ci);
tel

const n=10;

node ADD(A,B : bool^n) returns (S : bool^n; overflow : bool);
let
   overflow,S = map_red<<FULL_ADD;n>>(0,A,B);
tel
```

FIG. 4.13 – Additionneur n bits avec itérateurs.

4.4.2.2 Exemple à mémoire

On donne à la figure 4.15 un exemple d'itération où à la fois le programme principal mais aussi un nœud itéré utilisent l'opérateur pre. La figure 4.16 donne le code impératif produit pour ce programme. On trouve cette fois-ci la distinction nette entre instant initial (partie *init*) et instants suivants (partie *step*) qu'on pourrait isoler dans des fonctions.

```
typedef int Boolean;
typedef int Integer;

void _ADD(Boolean _A[10],
          Boolean _B[10],
          Boolean *_S[10],
          Boolean *_overflow){
    Integer i0;
    Boolean mem_overflow;
    *_overflow = 0;
    for(i0=0;i0<10;i0++){
       mem_overflow = *_overflow;

       *_S[i0] = _A[i0] ^ (_B[i0] || mem_overflow);
       *_overflow = (_A[i0] && _B[i0])
                    ^ (_B[i0] && mem_overflow)
                    ^ (_A[i0] && mem_overflow);
    }
}
```

FIG. 4.14 – Code obtenu pour le programme ADD.

```
node M(acc_in,t1,t2 : int) returns (acc_out : int);
var w : int;
let
   w = 0 -> pre(w) + pre(acc_in);
   acc_out = 0 -> (acc_in + w + t1 + pre(t2));
tel

node main (e1,e2 : int^10) returns (s,v : int);
var v : int;
let
   v = 0 -> pre(v)+1;
   s = red<<M;10>>(0,e1,e2);
tel
```

FIG. 4.15 – Un programme itératif à mémoire.

4.4.2.3 Itération à plusieurs dimensions

Les itérateurs LUSTRE permettent de travailler facilement avec des tableaux à plusieurs dimensions. Nous allons voir un exemple d'un tel calcul, puis nous verrons le code que l'on peut produire pour ce programme. Le code obtenu est constitué d'imbrications de boucles de type for et manipule des tableaux à plusieurs dimensions.

On commence par implémenter le nœud qui multiplie par m son entrée (on fixe ici i=2) (figure

```
void _main(Integer _e1[10],
           Integer _e2[10],
           Integer *_s;
           Integer *_v;{
    Int pre_v;                                   // mémoires du nœud principal

                                                 // mémoires des nœud itérés
    Int pre_w[10], _w[10];
    Int pre_acc_in[10], _acc_in[10];
    Int pre_e2[10];

    if(init){                                    // Instant initial
      _v = 0;                                    // définition de v
      _s = 0                                     // définition de s
      for(int j=0; j<10; j++){
        _acc_in[j] = s;
        _w[j] = 0;
        _s = 0;
      }
    }else{                                       // Instants suivants
      v = pre_v + 1;                             // définition de v
      s = 0;                                     // définition de s
      for(int j=0; j<10; j++){
        _acc_in[j] = s;
        _w[j] = pre_w[j]
                + pre_acc_in[j];
        _s = _s + _w[j]
             + _e1[j] + pre_e2[j];
      }
    }
                                                 //Mise à jour des mémoires
    pre_v = _v;                                  // mémorisation de v (pour pre(v))
    for(int j=0; j<10; j++){                     // mémorisations liées à l'itération
      pre_acc_in[j] = acc_in[j];
      pre_w[j] = _w[j];
      pre_e2[j] = _e2[j];
    }
}
```

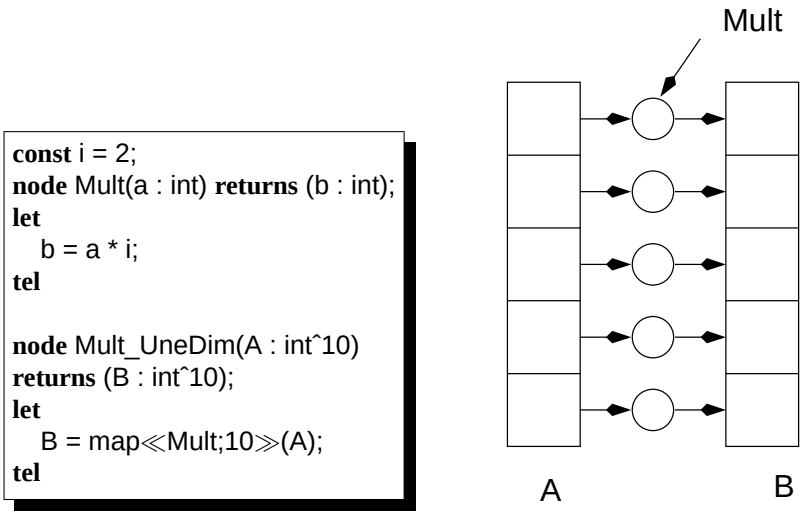
FIG. 4.16 – Code généré pour une itération d'un nœud à mémoire.

4.17). Ensuite, on itère ce nœud (voir figure 4.17) sur un vecteur d'une taille quelconque (disons n, qui sera un paramètre générique du nœud Mult_UneDim, fig. 4.17). Ainsi, le tableau B est défini par l'équation :

$$\forall i \in [0..n], B[i] = A[i] * 2$$

Incrémentalement, on construit le programme Mult_DeuxDim (voir figure 4.18), qui effectue la même opération, mais sur des tableaux à 2 dimensions (n et l). Et, enfin (figure 4.19), le programme qui effectue le calcul sur des matrices à 3 dimensions (n, l et m) est le nœud Mult_TroisDim.

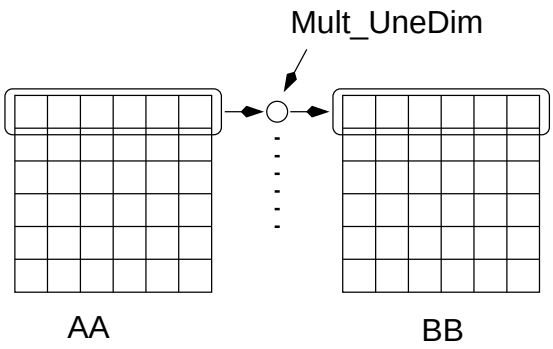
L'algorithme de l'annexe A permet de traiter des tableaux et itérations à plusieurs dimensions. Pour le nœud Mult_ToisDim on obtient le code de la figure 4.20.




```
const i = 2;
node Mult(a : int) returns (b : int);
let
   b = a * i;
tel

node Mult_UneDim(A : int^10)
returns (B : int^10);
let
   B = map<<Mult;10>>(A);
tel
```

FIG. 4.17 – Le nœud Mult_UneDim.




```
node Mult_DeuxDim(AA : int^10^11)
returns (BB : int^10^11);
let
   BB = map<<Mult_UneDim;11>>(AA);
tel
```

FIG. 4.18 – Le nœud Mult_DeuxDim.

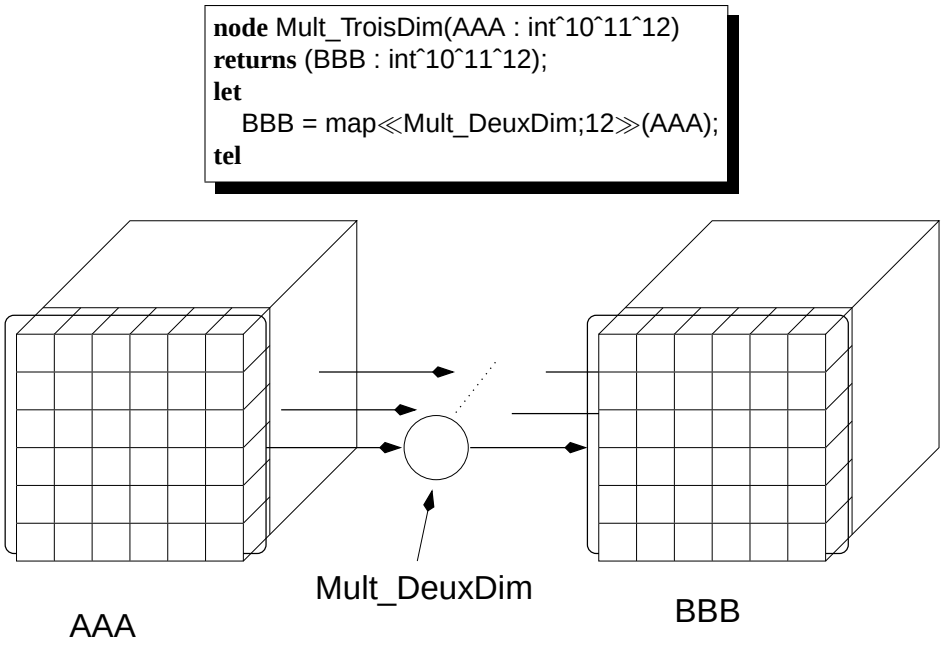


FIG. 4.19 – Le nœud Mult_TroisDim.

```
void main(int Tab_in[10][11][12],
          int *Tab_out[10][11][12]){
    int i0;
    for(i0=0;i<12;i0++){
        int i1;
        for(i1=0;i1<11;i1++){
            int i2;
            for(i2=0;i2<10;i2++){
                Tab_out[i1][i2][i0] = Tab_in[i2][i1][i0] * 2;
            }
        }
    }
}
```

FIG. 4.20 – Code obtenu pour l'itération de Mult_DeuxDim sur un tableau à 3 dimensions.

4.5 Optimisation des enchaînements d'itérateurs

4.5.1 Problématique

Considérons le programme de la figure 4.21. T′ est défini par un map du nœud f appliqué à T et T″ par un map de g appliqué à T′. La définition exacte de f et g a peu d'importance : dans l'exemple les deux nœuds sont combinatoires, mais ils pourraient aussi bien avoir des mémoires locales ou utiliser des horloges particulières.

D'après la définition de l'opérateur map, T′ et T″ sont définis par :

$$\forall i \in [0..n], \mathsf{T}'[\mathrm{i}] = \mathsf{f}(\mathsf{T}[\mathrm{i}]), \tag{4.1}$$

$$\forall i \in [0..n], \mathsf{T}''[\mathrm{i}] = \mathsf{g}(\mathsf{T}'[\mathrm{i}]). \tag{4.2}$$

D'après ces définitions, on peut redéfinir facilement chaque élément T″[i] par une composition de f et g appliquée à T[i].

$$\forall i \in [0..n], \mathsf{T}''[\mathrm{i}] = \mathsf{g}(\mathsf{f}(\mathsf{T}[\mathrm{i}])). \tag{4.3}$$

Il faut remarquer aussi que dans le programme décrit ci-dessus, la seule utilité du tableau T' est de rendre le programme plus clair. T' n'est en effet pas une sortie du programme et n'est utilisée nulle part ailleurs dans le nœud main. On peut alors concevoir de remplacer ce programme par le programme de la figure 4.22, sémantiquement équivalent, où h est un nouveau nœud, inexistant auparavant, correspondant à la *composition* des nœuds f et g.

```
node main(T : int^10) returns (T// : int);
var T/ : int^10;
let
   T/ = map<<f;n>>(T);
   T// = map<<g;n>>(T/);
tel
node f(in_f : int) returns (out_f : int);
let
   out_f = in_f + 1;
tel
node g(in_g : int) returns (out_g : int);
let
   out_g = in_g + 2;
tel
```

FIG. 4.21 – Un exemple d'enchaînement d'itérations.

4.5.2 Conditions d'applicabilité

On ne peut appliquer cette transformation que si le tableau T' est inutile pour le reste du programme : on ne peut pas supprimer des variables qui seraient des sorties d'un programme ni qui permettraient de calculer directement ou indirectement des sorties. On peut facilement définir cette notion d'*inutilité* :

```
node main(T : int^10)returns(T// : int);
let
   T// = map<<a_0;n>>(T );
tel
node h(in_f : int)returns(out_g : int);
var
    in_g : int ;
    out_f : int ;
let
    out_f = in_f + 1;
    in_g = (out_f);
    out_g = in_g + 2;
tel
```

FIG. 4.22 – Version optimisée du programme main.

Définition 17 — Variable inutile vis-à-vis de l'enchaînement des itérateurs

On dit qu'une variable V est inutile *dans un nœud* LUSTRE, vis-à-vis de l'enchaînement des itérateurs, *si* toutes *les conditions suivantes sont vérifiées :*

- *V est locale au nœud considéré ;*
- *les 2 itérations considérées portent sur des tableaux de tailles identiques ;*
- *V n'apparaît* que *dans 2 équations :*
  - *dans la première comme résultat d'une itération ;*
  - *dans la seconde comme paramètre d'une autre itération ;*
- *Ces 2 itérations sont optimisables par les axiomes introduits dans ce chapitre.*

4.5.3 Avantages et inconvénients

Optimisations du code généré – Cette optimisation représente, comme nous l'avons dit plus haut, une double source d'optimisation :

- On économise la place mémoire normalement allouée à la variable T' ;
- On produit un code plus rapide puisque ne contenant qu'un seule boucle for, correspondant à l'unique map du programme résultat.

Les figures 4.23 et 4.24 montrent les 2 codes produits pour le programme main ci-dessus : avec ou sans optimisation.

On peut imaginer que la mise en place des itérateurs dans les environnements (comme l'atelier SCADE) se fasse sous la forme de bibliothèques. Ces bibliothèques contiendront des programmes génériques implantant des algorithmes classiques sur les tableaux (comme les calculs de maximum, minimum, tris, etc.). L'utilisateur final n'aura généralement pas à se poser les questions d'optimisation que nous soulevons ici : il manipulera les programmes fournis, en les arrangeant pour programmer ses propres algorithmes.

C'est cette méthode d'utilisation qui nous permet d'affirmer que ces optimisations seront utiles. L'utilisateur créera sans même sans rendre compte, en utilisant les programmes fournis dans la bibliothèque, des enchaînements d'itérations. Le compilateur LUSTRE doit donc prendre lui-même ces optimisations en charge.

```
void _main(Integer _T[10],
           Integer *_T//[10]){
    Integer _T/[10];
    Integer i0;
    for(i0=0;i0<n;i0++){
        _T/[i0]=_T[i0] + 1;
    }

    Integer i0;
    for(i0=0;i0<n;i0++){
    *_T//[i0]=T/[i0] + 2;
    }
}
```

FIG. 4.23 – Code produit pour l'enchaînement de 2 maps *sans* optimisation des enchaînements.

```
void  _main(Integer _T[10],
            Integer *_T//[10]){
    Integer i0;
    Integer _ing;
    Integer _outf;
    for(i0=0;i0<n;i0++){
        _outf=_T[i0] + 1;
        _ing=_outf;
        *_T//[i0]=_ing + 2;
    }
}
```

FIG. 4.24 – Code produit pour l'enchaînement de 2 maps *avec* optimisation des enchaînements.

Perte de traçabilité – La mise en place de l'optimisation des enchaînements d'itérateurs entraîne une perte de traçabilité du programme. On supprime des variables et on modifie la structuration fine du programme. Dans sa globalité, il garde la même sémantique, mais la méthode programmée par l'utilisateur n'est plus apparente. Cela peut avoir une incidence importante sur la correction des programmes (vérification, débogage).

Pour cette raison, ces optimisations ne doivent pas être implémentées dans les outils de correction de programme (interprètes, logiciels de test et débogueur), mais uniquement au moment de la compilation, c'est-à-dire là où l'utilisateur n'est plus censé devoir lire le code.

4.5.4 Axiomatisation

L'enchaînement montré dans le paragraphe précédent est trivial. Nous avons identifié en tout une dizaine de cas où l'on peut appliquer une technique similaire. Le tableau ci-dessous synthétise les optimisations possibles. La première case de la deuxième ligne du tableau se lit comme suit : « l'enchaînement map(fill) est optimisable ».

Dans cette partie, nous considérons chacun des cas possibles et nous en détaillons l'optimisation.

↗	map	fill	red	map-red
map	⋆		⋆	⋆
fill	⋆		⋆	⋆
red				
map-red	⋆		⋆	⋆

4.5.4.1 map **suivi de** map

Soient 2 fonctions $f = \lambda t.t'$ et $g = \lambda t.t''$. t' et t'' représentent en fait des expressions qui dépendent de t. L'enchaînement (voir figure 4.25) des map de ces 2 fonctions est tout à fait habituel, et correspond à l'exemple que nous avons présenté plus haut.

$$map(map(T, f), g)$$
$$\equiv$$
$$map(T, \lambda t.\ let\ x = f(t)\ in\ g(x)).$$

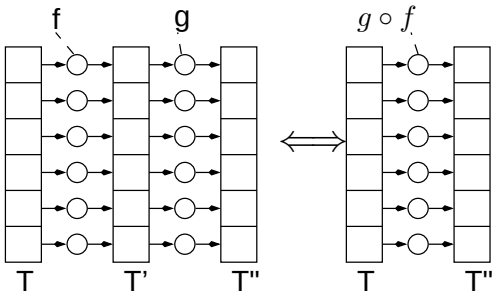


FIG. 4.25 – map(map).

4.5.4.2 map **suivi de** red

Supposons que l'on veuille mapper une fonction $f = \lambda t.t'$ sur un tableau T. Le résultat obtenu (voir figure 4.26) est alors réduit par le nœud $g = \lambda a, < t.a' >$ grâce à l'opérateur red. Cet enchaînement est optimisable vers une unique réduction, et ce grâce à l'équivalence de la figure 4.26

$$red(i, map(T, f), g)$$
$$\equiv$$
$$red(i, T, \lambda a, t\ .\ let\ x = f(t)\ in\ g(a, x)).$$

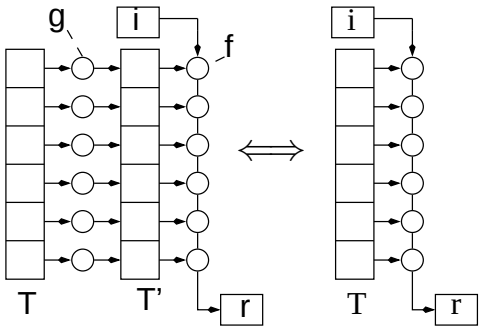


FIG. 4.26 – red(map).

4.5.4.3 map **suivi de** map_red

Si la fonction itérée par le map est $f = \lambda t.t'$ et celle itérée par le map_red est $g = \lambda a, t. < a', t'' >$, alors, on peut appliquer l'axiome suivant de la figure 4.27.

$$map\_red(i, map(T, f), g)$$
$$\equiv$$
$$map\_red(i, T, \lambda a, t\ .\ let\ x = f(t)\ in\ g(a, x)).$$

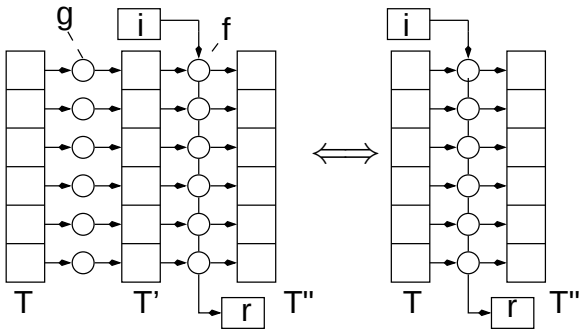


FIG. 4.27 – mapred(map).

4.5.4.4 fill **suivi de** map

Soit $f = \lambda a.a'$ la fonction utilisé dans le map et soit $g = \lambda a. < a', t >$ la fonction de remplissage. L'enchaînement fill(map) peut être optimisé en le remplaçant par un *remplissage* à l'aide d'un nœud, combinaison de f et de g (voir figure 4.28).

$$map(fill(i, g)`T, f)$$
$$\equiv$$
$$fill(i, T, \lambda a\ .\ let < x, y > = g(a)\ in\ let\ z = f(y)\ in$$
$$< x, z >).$$

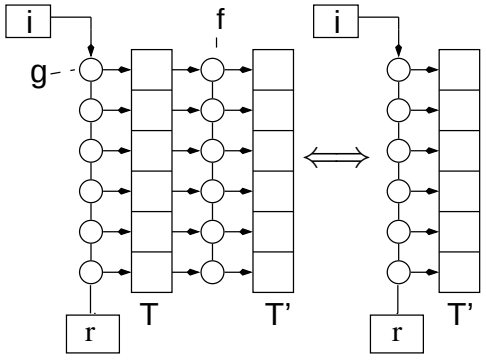


FIG. 4.28 – map(fill).

4.5.4.5 fill **suivi de** red

Soit le programme LUSTRE suivant :

```
node main(i,j : int) returns (r : int);
var T : intˆ10;
      s : int;
let
   s,T = fill≪f;n≫(i);
   r = red≪g;n≫(j,T);
tel

node f(Acc_in_f : int) returns (Acc_out_f : int ; Elt_out_f : int);
node g(Acc_in_g : int ; Elt_in_g : int) returns (Acc_out_g : int);
```

Cet enchaînement n'est pas optimisable en une autre forme itérée (avec tableau). L'algorithme de remplissage (représenté par l'itérateur fill) produit des éléments dans le tableau T ; l'algorithme de réduction (l'itérateur red) consomme ces éléments.

Dans le programme ci-dessus, on remplit d'abord T pour ensuite le parcourir à nouveau. On s'aperçoit alors que le tableau T est *inutile* (on sens de la définition 4.5.2). En effet, si l'on se réfère au schéma 4.29, on voit bien que que le calcul de r pourrait se faire sans mémoriser *tous* les éléments de T.

Soit le code produit pour l'équation T = fill≪f;n≫(i); :

```
s = i;
for(i0=0;i0<n;i0++){
   x,y = f(s);
   T[i0] = y;
   s = x;}
```

Et soit le code produit pour l'équation r = red≪g;n≫(T) :

```
r = j;
for(i0=0;i0<n;i0++){
   r = g(r,T[i0]);}
```

On peut remplacer le code obtenu précédemment par le code ci-dessous (équivalent) à la condition suivante qu'il n'existe pas de dépendance instantanée entre s et j.

```
r = j; s = i;
for(i0=0;i0<n;i0++){
   x,y = f(s);
   r = g(r,y);
   s = x;}
```

On ne peut réaliser cette optimisation que lors de la génération de code impératif.

4.5.4.6 fill **suivi de** map_red

Soit $f = \lambda a.(a', t)$ la fonction utilisée dans le fill et soit $g = \lambda a, t.(a', t')$ la fonction utilisée dans le map_red. Si j ne dépend pas de s dans l'instant, on peut appliquer la règle de la figure 4.30.

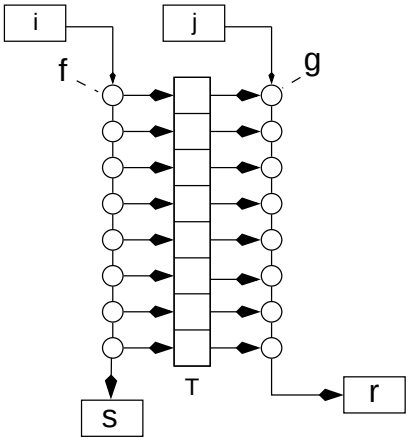


FIG. 4.29 – Le cas particulier de l'enchaînement red(fill).

$$map\_red(i, fill(j, f)`T, g)$$
$$\equiv$$
$$fill(\{i, j\}, \lambda a_1, a_2\ .\ let < x, y >= f(a_1)\ in\ let$$
$$< x', y' >= g(a_2, y)\ in < \{x, x'\}, y' >)$$

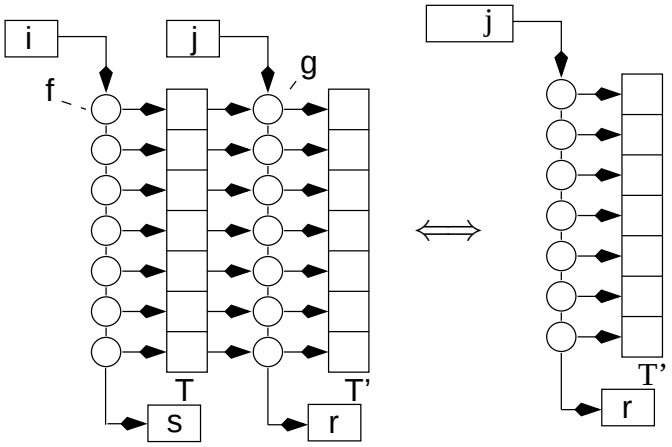


FIG. 4.30 – map_red(fill).

4.5.4.7 map_red **suivi de** map

Soit $f = \lambda(a,t).(a',t')$ la fonction utilisée dans le premier map_red et soit $g = \lambda(a,t).(a',t')$ la fonction de remplissage. On applique alors la règle de la figure 4.31.

$$map(map\_red(i,T,f),g) \equiv map\_red(i,T,\lambda\, a,t\ .\ let < x,y > = f(a,t)\ in < x,g(y) >).$$

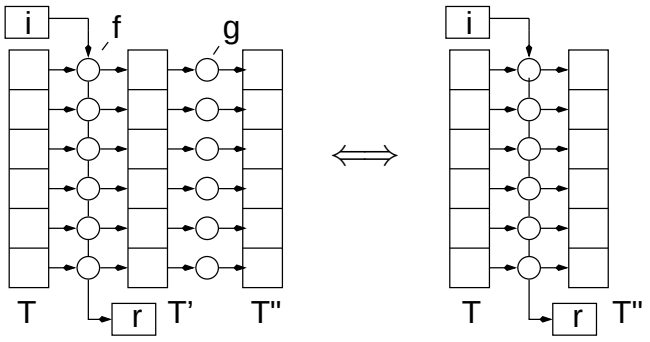


FIG. 4.31 – map(map_red).

4.5.4.8 map_red **suivi de** red

Soient $f = \lambda a,t.a',t'$ le nœud itéré par map_red et $g = \lambda a,t.a'$ le nœud itéré par red. Une condition nécessaire pour que cet enchaînement soit optimisé est :

Si s est le résultat du map_red, et si j est la valeur d'initialisation du red, alors il *faut* que j ne dépende pas de s, dans l'instant. En effet, dans le cas où j dépend de s, il faut appliquer le map_red de f avant de pouvoir appliquer le red de g. Si cette condition est vérifiée on peut appliquer la règle de la figure 4.32.

4.5.4.9 map_red **suivi de** map_red

Pour optimiser cet enchaînement il faut vérifier que :

$f = \lambda a,t.(a',t')$

et $g = \lambda a,t.(a'',t'')$,

Alors on peut appliquer la règle de la figure 4.33. On doit vérifier la même condition que précédemment quant à une dépendance instantanée entre s et j.

4.6 Travaux connexes

4.6.1 Sur les itérations

La notion même d'itérateurs est très étroitement liée à celle de fonctions d'ordre supérieur et plus généralement au style fonctionnel. Parmi les premières apparitions de ces idées, on trouve les travaux de John Backus [Bac78]. Celui-ci donne déjà des opérateurs pour manipuler des listes tels que les

$$red(j, map\_red(i, T, f))$$
$$\equiv$$
$$red(\{i, j\}, T, \lambda\ \{a_1, a_2\}, t\ .\ let < x, y > = f(a_1, t)\ in$$
$$let\ x' = g(a_2, y)\ in\ \{x, x'\})$$

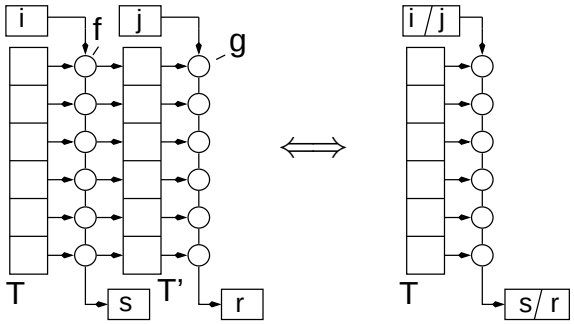


FIG. 4.32 – red(map_red).

$$map\_red(j, map\_red(i, T, f), g)$$
$$\equiv$$
$$map\_red(\{i, j\}, T, \lambda\{a_1, a_2\}, t\ .\ let < x, y > =$$
$$f(a_2, t)\ in\ let < x', y' > = f(a_2, y)\ in$$
$$< \{x, x'\}, y' >).$$

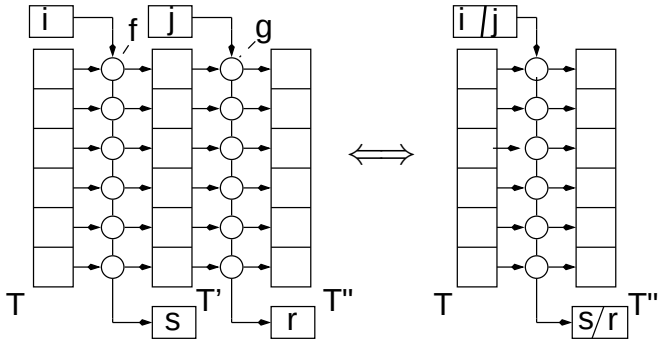


FIG. 4.33 – map_red(map_red).

Insert ou *Apply to all* (correspondant respectivement à la réduction gauche et au mapping), et des idées sur la simplification des compositions de telles fonctions.

Dès les origines, le but annoncé était de fournir des outils puissants, permettant une réflexion « *haut-niveau* » plus mathématique que celle habituellement trouvée dans les méthodes de programmation impérative.

Ces points représentent exactement une des motivations pour l'introduction d'itérateurs en LUSTRE : proposer au programmeur des constructions *simples*, *claires* et pour lesquelles on sait produire un code efficace et sûr.

Il est important de souligner que dans la majorité des travaux présentés ici, les objets manipulés sont essentiellement des listes (ou plus généralement des structures arborescentes). D'autre part, dans les travaux où l'on parle explicitement de tableaux ([Bir88] par exemple), les opérations fournies (notamment les possibilités de *construction* de tableaux) sont beaucoup plus puissantes que les possibilités de LUSTRE.

Il faut comparer les techniques présentées dans ces articles et nos propres propositions en gardant toujours à l'esprit ces quelques remarques :

- nos tableaux sont *toujours* de taille fixe et connue statiquement. Ce n'est pas le cas des listes qui peuvent être *infinies* ;
- il ne nous est pas possible de construire des tableaux par *concaténation* ;
- en LUSTRE, aucune création dynamique de tableau n'est possible.

Le formalisme de Bird-Meertens – Le formalisme de Bird-Meertens (BMF) a été introduit par *Richard Bird* dans [Bir88]. Le but est de fournir une *théorie mathématique* complète pour l'utilisation de l'approche fonctionnelle dans la spécification des programmes. Cette théorie est basée sur des notations qui permettent de manipuler plus facilement des listes, des tableaux et des arbres. Des opérations telles que le 'mapping', la réduction gauche et droite, les accumulations (qui correspondent à nos map_red) sont définies.

Bird propose aussi différentes méthodes de compilation pour BMF. Une d'entre elles consiste à traduire ces programmes directement en code impératif. Par exemple, une réduction gauche peut être traduite en une boucle for.

On retrouve aussi dans [Bir88] (ainsi que dans [JV00]) la notion de *catamorphisme* qui est une forme générale de la réduction que nous proposons[3] :

Définition 18 — Catamorphisme

> *A function on finite data structures having a tree-like data type is a catamorphism if it can be computed bottom-up by replacing the constructors systematically by an evaluation function.*

Par exemple, la somme des éléments d'une liste d'entiers est un catamorphisme : on remplace les éléments de la liste par leur valeur et les constructeurs de listes (concaténation) par le + (l'addition sur les entiers).

De telles opérations (map, reduce, ...) ont été incluses dans des langages plus spécialisés tels que $8_{1/2}$ [OM92], ALPHA [Mau89] ou ID [Nik91]. Les deux premiers implémentent des approches flots de donnée, mais dans le but de résoudre des problèmes d'architecture matérielle (parallélisme de calcul). ID est un langage fonctionnel général.

[3] Cette définition est donnée par Eelco Visser, sur
http://losser.st-lab.cs.uu.nl/~visser/cgi-bin/twiki/view/Transform/CataMorphism

Liens avec les itérateurs LUSTRE – En ce qui nous concerne, BMF est une bonne base pour comprendre les mécanismes fonctionnels que nous manipulons. Les schémas de compilation vers des boucles de type for que suggère Bird sont identiques à ceux que nous avons adoptés dans la génération de code C pour nos itérateurs. Enfin, les règles d'optimisation que nous proposons en section 4.5 sont comparables au règles de *promotion* de Bird.

Les opérateurs proposés pour manipuler les tableaux sont cependant plus complets que ceux que nous proposons pour LUSTRE. Cela vient principalement du fait que Bird travaille avec une sémantique purement fonctionnelle. La sémantique de LUSTRE est plus réduite : on n'a typiquement pas de lambda explicite, d'ordre supérieur ni d'évaluation partielle.

4.6.2 Sur les optimisations des enchaînements d'itérateurs

Les optimisations que nous présentons au paragraphe 4.5 sont issues du domaine de la programmation fonctionnelle. Les principales références portent sur la notion de *listlessness* et de *déforestation* de Wadler [Wad90][4] et sur les travaux de Richard Waters.

4.6.2.1 Transformation automatique d'expressions "sérielles" en boucles

Dans [Wat91], Waters souligne les avantages de la programmation à l'aide d'expressions "sérielles", et des techniques d'optimisations qui peuvent être utilisées dans ce cadre. Les objets manipulés dans ce travail sont des listes, et la différence est assez importante car, comme nous l'avons déjà remarqué, les contraintes qu'apporte l'aspect "infini" des listes n'apparaissent pas pour nos tableaux.

Waters présente les avantages considérables qu'il y a à utiliser des fonctions d'ordre supérieur. Cette technique favorise une séparation claire des calculs, de potentielles vérifications et modifications étant ainsi plus faciles à effectuer sur le programme.

De manière orthogonale, ces méthodes ne sont pas beaucoup utilisées, et ce pour deux raisons :

- les constructions proposées au programmeur ne sont pas faciles d'emploi ;
- les méthodes de compilation utilisées ne sont que très rarement efficaces.

D'après Waters, la principale source d'inefficacité réside dans la création de structure intermédiaires lors d'enchaînements de fonctions sérielles, alors que cela pourrait être évité dans de nombreux cas.

Il propose à la fois des schémas de compilation de ces itérations vers du code séquentiel avec boucle et des optimisations qui visent à diminuer le nombre de ces listes intermédiaires. Ces optimisations sont comparables aux axiomes que nous proposons en page 87. Les schémas de compilation proposés sont à peu près ceux que nous avons adoptés.

4.6.2.2 Listlessness et déforestation

La déforestation a été introduite en 1990 par Philip Wadler dans [Wad90]. L'auteur y déclare que les *structures intermédiaires sont le fardeau de la programmation fonctionnelle*. Le but du travail présenté dans cet article est de fournir des techniques pour *déforester* les programmes, c'est-à-dire *supprimer les structures intermédiaires lorsqu'elles sont inutiles*. Wadler définit 2 formes de programmes, dont la seconde, appelée *tree-less form,* est une forme dans laquelle on garantit l'absence de structures intermédiaires inutiles dans les programmes. Tout programme écrit dans la première forme est traduisible en un programme de la seconde forme et toute composition de fonction "*tree-less*" peut être traduite en une seule fonction "*tree-less*". L'algorithme de déforestation donné est un parcours de structure qui isole les enchaînements optimisables et applique les axiomes donnés.

[4] une bonne introduction peut aussi être trouvée dans [GLP93]

La notion de déforestation est une version étendue de celle de *listlessness* ([Wad84, Wad85]) puisqu'on ne considère plus simplement des listes intermédiaires, mais tout type de structures (notamment les arbres abstraits dans un processus de compilation).

Une implémentation des techniques de déforestation peut être trouvée dans [GLP93]. Une technique dérivée, appelée *warm fusion* est présentée dans [LS95].

Dans les travaux de Wadler, on traite n'importe quelle fonction récursive. Cette généralité pose certains problèmes notamment quant à la preuve de terminaison des algorithmes de déforestation. Dans les travaux cités ci-dessus, l'étude est restreinte à l'étude des catamorphismes.

Les optimisations d'enchaînements d'itérations peuvent aussi être rapprochées des techniques de fusion de boucles (une introduction peut être trouvée dans [Muc00]) qui permettent de transformer des séquences de boucles impératives de types for en une seule boucle. Nos itérations sont des boucles de types for. Nous pourrions envisager d'appliquer ces techniques fusion de boucles sur le code impératif généré par le compilateur. Nous trouvons plus intéressant d'appliquer ces techniques au niveau du langage LUSTRE lui-même et, de manière générale, le plus tôt possible. En effet ces transformations ne sont pas forcément disponibles dans les compilateurs standards. Même si elles sont prévues, il n'est pas évident qu'elles soient aussi efficaces que si on les effectue au niveau LUSTRE. L'analyse nécessaire est assez simple à mettre en place au niveau LUSTRE : il s'agit d'un simple parcours du graphe de flot-de-données.

4.6.2.3 Remarques et liens avec les itérateurs LUSTRE

Tous ces travaux proposent des techniques pour optimiser des enchaînements de programmes itératifs (afin de supprimer des structures intermédiaires). Mais ils permettent aussi de faire apparaître un lien intéressant entre plusieurs formes de calcul itératifs qui cohabitent dans LUSTRE.

De l'optimisation des enchaînements d'itérations – L'élimination des tableaux intermédiaires a été un des points importants du travail autour des itérateurs (voir section 4.5). L'optimisation des enchaînements d'itérations est une application de la déforestation de Wadler. Bien que d'application très générale (les langages fonctionnels), ces travaux nous ont servi de référence pour l'écriture des axiomes d'optimisations présentés plus haut.

Approche flot-de-donnée / Itérations – Par ailleurs, il est intéressant de faire le lien entre les enchaînements d'itérations et le fondement même des langages flots de données. La sémantique de LUSTRE est inspirée, entre autres, de la sémantique de Kahn et de la notion de listlessness de Wadler. Les conditions proposées par Waters afin de rendre possible l'élimination de flots (listes) intermédiaires sont des propriétés basiques des programmes LUSTRE (synchronisme, réactivité...). On pourra notamment, pour s'en convaincre, se référer à [CP96].

Dans notre travail, au lieu de manipuler des listes, nous manipulons des tableaux. Au lieu de travailler dans le temps, nous travaillons dans l'espace, puisqu'au lieu d'avoir à attendre les éléments les uns après les autres dans le temps, nous les avons tous en même temps, mais « étalés » dans l'espace sous forme de tableaux. Mais les même techniques s'appliquent avec le même but : limiter les structures intermédiaires dans des programmes « répétitifs » (au sens où les opérations sont « *répétées* » au fil du temps pour LUSTRE, et qu'on « *répète* » l'opération itérée sur les éléments d'un tableau).

4.6.3 Itération de nœuds à mémoire - *Retiming*

Le travail autour de la compilation et l'optimisation des itérations s'est déroulé en deux grandes étapes. Nous avons commencé à étudier les itérations de nœuds combinatoires. Ce n'est qu'une fois que nous avons clarifié cette question que nous nous sommes intéressé au cas où le nœud itéré est un nœud LUSTRE quelconque, avec potentiellement une mémoire (des occurrences de l'opérateur pre).

Nous avons tout d'abord étudié la possibilité d'« *éliminer* » les occurrences de l'opérateur pre du code du nœud itéré, en les faisant sortir de celui-ci vers le niveau de l'itération.

Dans ce but, nous avons étudié le *retiming*, qui est une technique définie par *Leiserson* et *Saxe* dans [LS90]. Ces travaux ont été beaucoup utilisés dans le domaine de la programmation de VLSI. Le *retiming* permet de modifier le nombre de « délais » (tampons qui permettent de mémoriser des valeurs en vue d'utilisations ultérieures) à certains endroits dans un circuit synchrone afin de minimiser le temps de parcours du chemin d'exécution critique. On veut bien sûr à conserver la sémantique du programme considéré.

Dans ce travail, les circuits sont représentés sous la forme de graphes de flots de données synchrones (équivalents à des programmes LUSTRE).

Les Graphes Flots de Donnée synchrones – Un DFG synchrone G est un triplet $< V, E, \delta, \omega >$ où V est l'ensemble des sommets de G (qui sont des sous-programmes), E est l'ensemble des arcs de G et δ est la fonction qui associe à tout sommet v de V son délai de propagation $\delta(v)$ qui est égal au temps nécessaire au sous-programme V pour fournir ses sorties, une fois qu'il a reçu toutes ses entrées. G contient aussi un certain nombre de registres (sortes de tampons mémoires : un registre fournit en sortie la valeur précédente de son entrée) répartis sur les arcs de G. Le registre est l'équivalent de l'opérateur pre de LUSTRE. En LUSTRE, chaque sommet v est tel que $\delta(v) = 0$.

EXEMPLE 21 — Soit un programme, appelé corrélateur, à 1 entrée x, qui la compare avec k constantes a_0, a_1, ... a_{k-1} Après chaque nouvel x_i ($i \geq k$), le corrélateur produit une sortie y qui est le nombre de correspondances défini par :

$$y_i = \sum_{j=0}^{k} \rho(x_{j-i}, a_j)$$

où ρ est la fonction de comparaison suivante :

$$\rho(x, y) = \begin{cases} 1 & : \text{ si } x = y \\ 0 & : \text{ sinon} \end{cases}$$

La figure 4.34 montre le circuit correspondant et la figure 4.35 montre le DFG synchrone. On supposera, comme cela est fait dans [LS90] que le délai de propagation de chaque ρ est de 3 unités de temps et que celui des $+$ est de 7 unités de temps. $k = 3$.

— FIN DE L'EXEMPLE 21

Soit e un élément de E (donc un arc du graphe G). On définit une valeur, notée $\omega(e)$ qui est le nombre de registres présents sur l'arc e.

On définit des chemins dans G, et on définit la fonction ω sur les chemins :

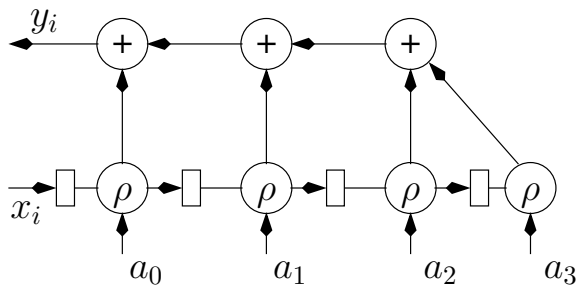


FIG. 4.34 – Le circuit du corrélateur.

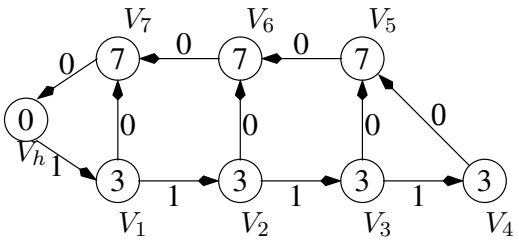


FIG. 4.35 – DFG synchrone.

Définition 19 — Nombre de registres

$\omega(p)$ est le nombre de registres sur le chemin p. Si $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{k-1}} v_k$ alors :

$$\omega(p) = \sum_{i=0}^{k-1} \omega(e_i)$$

De la même façon, on peut définir le délai de propagation sur un chemin d'opérateur.

Définition 20 — Délai de propagation

$\delta(p)$ est le délai de propagation du chemin p. Si $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{k-1}} v_k$ alors :

$$\delta(p) = \sum_{i=0}^{k} \delta(v_i)$$

EXEMPLE 22 — Dans l'exemple du corrélateur, on a : $\omega(V_1, V_4) = 3$ et $\delta(V_1, V_4) = 12$.
— FIN DE L'EXEMPLE 22

Le but des techniques proposées dans [LS90] est de minimiser le temps d'exécution d'un parcours du graphe. La seule modification que l'on peut faire consiste à déplacer ou supprimer des registres. En effet, on ne peut pas modifier le temps de calcul d'un nœuds du graphe. C'est ce qu'on appelle le *retiming*.

La *période* d'un graphe donné est le délai de propagation maximum pour les chemins de ce graphe qui ne contiennent aucun registre. C'est en quelque sorte la borne supérieure du temps d'exécution optimal, en dessous de laquelle on ne pourra descendre, en modifiant la répartition des registres. La période $\Phi(G)$ est définie par :

$$\Phi(G) = max\{\delta(p) \ : \omega(p) = 0\}$$

La période du graphe de la figure 4.35, est : $\Phi(G) = 24$.

***Retiming* d'un DFG synchrone** – Soit G un DFG. On cherche à optimiser le temps d'exécution du système représenté par G. Le *retiming* réalise cette optimisation en *rajoutant* ou en *enlevant* des registres sur certains arcs de G.

On définit r la *fonction de retiming* de V dans $\mathbb{Z}$ qui va nous permettre d'évaluer les modifications de placement des registres par rapport aux sommets du graphe. A chaque sommet v du graphe G, r

fait correspondre le nombre de registres qu'on enlève de chaque arc entrant dans v et qu'on place sur chaque arc sortant de v. On note G_r le graphe équivalent à G obtenu par le *retiming* r.

Ainsi, si ω_r est la fonction qui associe à chaque chemin du graphe le nombre de registres qu'il contient *après retiming*, on a :

$$\forall u \xrightarrow{p} v,\ \omega_r(p) = \omega(p) + r(v) - r(u).$$

Propriété 1 *Si p est un cycle, on a $r(v) = r(u)$ (puisque $u = v$), et donc $\omega_r(p) = \omega(p)$.*

L'algorithme de *retiming* donné dans [LS90] consiste, étant donné un DFG G, à minimiser $\Phi(G)$. Pour cela, on va se donner 2 nouvelles fonctions Ω et Δ qui vont nous permettre d'évaluer Φ. Soient :

$$\Omega(u, v) = min\{\omega(p) : u \xrightarrow{p} v\}$$

et

$$\Delta(u, v) = max\{\delta(p)\ : u \xrightarrow{p} v \text{ et } \omega(p) = \Omega(u, v)\}$$

Ω représente le nombre minimum de registres sur les chemins reliant les sommets u et v. Δ est le temps d'exécution maximal non-optimisable, c'est-à-dire le temps nécessaire pour exécuter un parcours du chemin entre 2 sommets u et v tels que $\omega(u, v) = \Omega(u, v)$.

On donne maintenant le théorème sur l'existence d'un *retiming* d'un graphe G :

Théorème 1 *Soit $G = < V, E, \delta, \omega >$ un DFG synchrone, soir c un nombre réel positif quelconque, et soir r une fonction de* retiming*, de V dans $\mathbb{Z}$. Alors r est un* retiming *de G tel que $\Phi(G_r) \leq c$* si et seulement si *:*

- $r(u)$ - $r(v) \leq \omega(e)\ \forall u \xrightarrow{e} v \in G$ *;*
- $r(u)$ - $r(v) \leq \Omega(u, v) - 1\ \forall u, v \in V$ *tels que $\Delta(u, v) > c$.*

La première condition stipule en fait que, par *retiming*, on ne doit pas augmenter le nombre de registres entre 2 sommets quelconques de G. La seconde condition signifie que l'on diminue forcément le temps minimum nécessaire à *relier 2 sommets du graphe G, en utilisant n'importe quel chemin les reliant*. Les inéquations de ce théorème peuvent être résolues grâce à l'algorithme de Bellman-Ford (voir [Law76]).

L'algorithme présenté dans [LS90] se base sur le théorème ci-dessus. Il permet de donner la fonction r pour un circuit G donné de façon à ce que la période du graphe obtenu par r soit minimale.

Algorithme 1 (Minimisation de la période d'un DFG synchrone) *Soit un DFG synchrone $G =< V, E, \delta, \omega >$, cet algorithme détermine un* retiming *r tel que $\Phi(G_r)$ est aussi petite que possible.*

- *calculer Ω et Δ ;*
- *trier les couples $< u, v >$ de sommets du graphe selon Δ ;*
- *Rechercher la période minimale atteignable parmi les $\Delta(u, v)$. Pour tester si une période c est atteignable, appliquer l'algorithme de* Bellman-Ford *pour vérifier si les conditions dans le théorème 1 sont vérifiées ;*
- *pour la période minimale atteignable trouvée en étape 3, le* retiming optimal *est la valeur minimale trouvée par l'algorithme de* Bellman-Ford*.*

À partir de l'exemple de la figure 4.35, on obtiendrait le DFG synchrone de la figure 4.36.

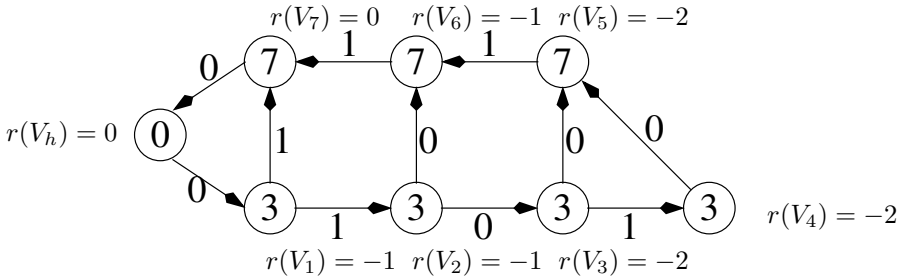


FIG. 4.36 – Graphe *retimé* du programme.

Liens avec les itérateurs LUSTRE – Comme il a été indiqué plus haut, on peut considérer les programmes LUSTRE comme des réseaux d'opérateurs, des DFG synchrones, où les registres sont les occurrences de l'opérateur pre. L'application du *retiming* à LUSTRE revient à appliquer les équivalences sur les pre : pre(a+b) est équivalent à pre(a) + pre(b). Dans le premier cas, on n'a à mémoriser qu'une seule valeur, celle de a+b, au lieu de mémoriser les valeurs pour a et pour b.

Dans les cas d'itérations de nœuds à mémoires, le but n'est pas d'optimiser le temps d'exécution, mais de « sortir » les occurrences de pre en dehors des nœuds itérés (donc au niveau des programmes *itérants*). On pourrait alors appliquer les techniques qu'on a développées pour les cas d'itérateurs de nœuds sans mémoires.

EXEMPLE 23 — Soit une itération d'un nœud N contenant des appels à l'opérateur pre (voir figure 4.37) :

```
node N(acc_in : int ; elt_in : int) returns (acc_out : int);
let
   acc_out = 0 -> elt_in + pre(acc_in);
tel

node main(init : int^10; Tab_in : int^10) returns (res : int);
let
   res = red<<N;10>>(init,Tab_in);
tel
```

En retimant N, on peut repousser les appels à l'opérateur pre à l'extérieur du nœud pour être appliqué sur les arguments de l'itération. La figure 4.38 montre le *retiming* effectué sur 2 étages du programme N. Si on peut « sortir » les opérateurs temporels, alors itérer le nœud N sur un tableaux T revient à itérer le nœud N' sur des mémoires du tableau T (voir la figure 4.39). On se ramène alors à des cas d'itérations de nœud sans mémoire, pour lesquelles le code est plus simple à générer.

— **FIN DE L'EXEMPLE 23**

Malheureusement, cette technique est assez lourde à appliquer. Il faut en effet parcourir tout le graphe d'opérateurs obtenu par mise à plat des instances du nœud itéré, afin de déterminer les occurrences de pre et les faire 'remonter' en dehors de ces instances.

D'autre part, cette méthode nécessite une restructuration totale des programmes. Non seulement on n'est pas certain de savoir automatiser cette restructuration mais dans la plupart des cas on obtient un résultat qui est moins efficace en termes de mémoire utilisée. Dans la version de la figure 4.39,

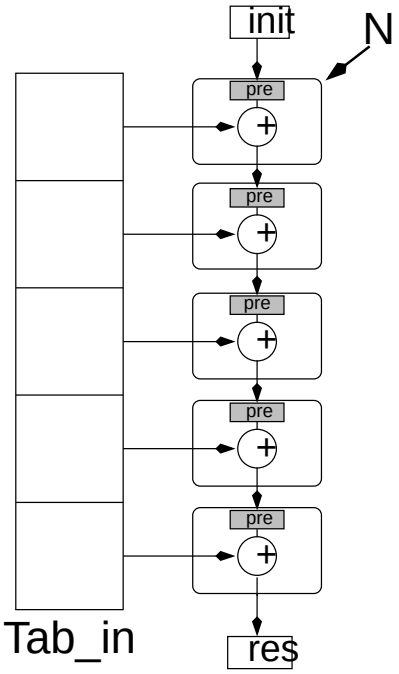


FIG. 4.37 – Itération d'un nœud à mémoire.

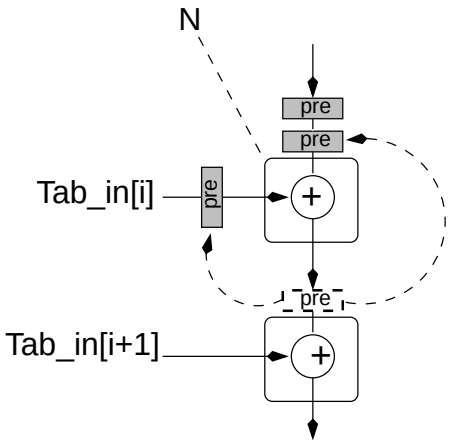


FIG. 4.38 – Première étape du *retiming*.

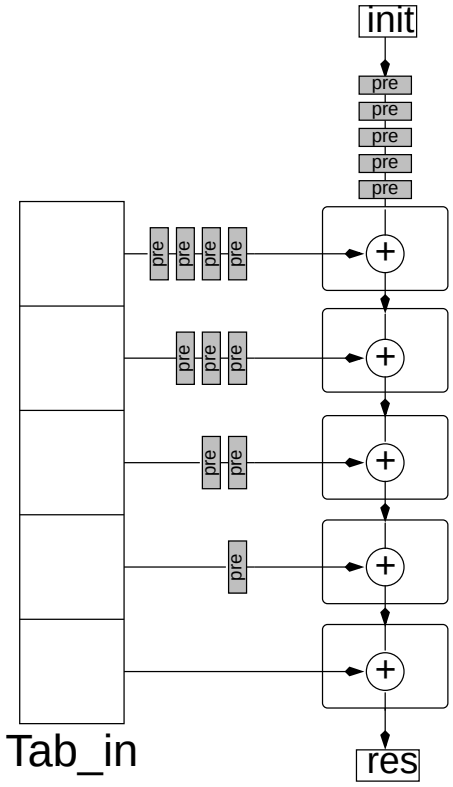


FIG. 4.39 – Résultat du *retiming* d'un red.

on voit bien qu'on a besoin de plus de cases mémoires (il y a plus d'occurrences de l'opérateur pre, exactement 15) que dans la version de la figure 4.37 (où l'on ne décompte que 5 occurrences de pre).

Qui plus est, l'itération obtenue par *retiming* manipule des tableaux de mémoires. Vraisemblablement, ces tableaux doivent être créés par des itérations de nœuds à mémoires. On ne fait donc que repousser le problème. En fait, on applique les équivalences pre(a+b)≡pre(a)+pre(b), mais dans le mauvais sens : on augmente le nombre de mémoires nécessaires. Le *retiming* ne semble donc pas représenter une solution raisonnable pour la compilation d'itérateurs à mémoire.

Nous n'avons finalement pas retenu cette approche pour la compilation d'itérations à mémoire. Nous avons développé l'algorithme mentionné plus haut et donné dans l'annexe A, qui traite le cas général d'itérations de nœuds LUSTRE quelconques.

4.7 Conclusions et perspectives

4.7.1 Transfert

Les travaux que nous avons présentés dans ce chapitre ont été menés dans le but de répondre à de réels besoins de la part de certains partenaires industriels. L'étude de cas du ELMU (décrite au chapitre 7) a permis de mettre en évidence la nécessité des itérateurs. L'étude du système de traitement de donnée gyroscopique (présenté au même chapitre) a permis de montrer que la symétrie impliquée par les itérations s'adaptait bien à la description des systèmes redondants tolérants aux pannes.

Le travail autour de la compilation des itérations et des optimisations d'enchaînements s'est fait en parallèle avec l'étude de cas du ELMU, et en collaboration avec la société *Esterel Technologies* qui développe SCADE, outil basé sur LUSTRE et utilisé pour l'étude de cas. Un compilateur adoptant un schéma de compilation identique à celui décrit plus haut a été développé au sein d'Esterel Technologies par M. Colaço [CP00]. Dans ce même rapport est également décrite une extension du langage Lucid Synchrone[] avec des itérateurs.

Cette méthodologie de programmation permet de réduire non seulement la taille des spécifications mais aussi la taille du code produit par le compilateur.

4.7.2 Vers un nouveau LUSTRE

Depuis le courant de l'année 2003, une nouvelle version du langage LUSTRE académique (appelée sobrement LUSTRE-*V6*) est en gestation. Elle incorpore les itérateurs tels que nous les avons décrits ici. L'équipe synchrone a aussi voulu introduire dans le langage deux nouveaux niveaux de conception appelés package et model. Un *package* est un regroupement de types, constantes et composants qui peut être paramétré. Un modèle est un package *paramétré*.

Par exemple, on peut décrire un modèle de package fournissant un seul composant implémentant l'opération *followed by* (initialisation et opération pre). Ce modèle est paramétré par le type des flots auxquels est appliqué le *followed by*. Les paramètres génériques du modèles sont donné après le mot-clef needs. On indique après le mot-clef provide la signature des fonctionnalités (composants et types) exportées par le package/model. Dans notre cas, le modèle modSimple fournit un seul nœud fby1 qui prend deux entrées de type t et produit une sortie de type t.

Entre les mots-clés body et end, on donne la définition des fonctionnalités énumérées après le mot-clef provide. On peut aussi trouver des définitions de composants utilisées seulement localement (par exemple pour aider à la définition des composants exportés).

```
model modSimple
  needs
    type t;
  provides
    node fby1(init, fb : t) returns (next : t);
  body

node fby1(init, fb : t) returns (next : t);
let
  next = init -> pre fb;
tel

end
```

On peut instancier le modèle modSimple en spécifiant la valeur des paramètres génériques. On peut ainsi définir 3 nouveaux packages pint, pbool et preal qui fournissent le composant *followed by* pour des flots entiers, booléens et réels :

```
package pint is modSimple(int)
package pbool is modSimple(bool)
package preal is modSimple(real)
```

Un programme LUSTRE contient nécessairement un package principal. On propose par exemple un package dont la seule fonctionnalité consiste en un opérateur *followed by* pour des variables d'un type structuré selType avec un champ entier, un champ booléen et un champ réel. Le package principal fournit donc d'un côté le type selType, de l'autre le nœud preced implémentant l'application de l'opérateur *followed by* à chacun des champs d'une variable de type selType.

```
package principal
  uses pint, pbool, preal;
  provides
    node preced(in : selType) returns (out : selType);
    type selType;
body
  type selType = {i : int; b : bool; r : real};
  node preced(in : selType) returns (out : selType);
  var init : selType;
  let
     init = { i = 0, b = true, r = 0.0};
     out = {i = pint::fby1(init.i, in.i),
              b = pbool::fby1(init.b, in.b),
              r = preal::fby1(init.r, in.r)};
  let
end
```

À partir d'un programme LUSTRE V6 contenant plusieurs packages ou modèles, le compilateur produit un programme XEC (pour eXtended-EC) qui ne contient plus qu'un nœud (toutes les appels de nœuds ont été expansés, mais les itérations ont été conservées).

L'analyseur syntaxique et sémantique du code source (réutilisé dans notre travail, voir le chapitre 11) ainsi que le générateur de code en XEC ont été développés à Verimag par M. Bouzouzou. A l'heure actuelle, le générateur de code impératif ainsi que le branchement vers les outils de validation est en cours de développement par Pascal Raymond.

Les itérateurs ont été inclus dans le compilateur LUSTRE-V6. Au niveau syntaxique, on peut itérer aussi bien des noeuds que des opérateurs de base du langage ou d'autres itérations. On peut par exemple écrire T3=map≪map≪+;s≫;t≫(T1,T2) qui calcule la matrice T3 définie par : $\forall i \in [0..t-1], j \in [0..s-1] T3[i,j] = T1[i,j] + T2[i,j]$. Les itérateurs sont conservés dans le format XEC. Le générateur de code prendra en compte les optimisations proposées plus haut.

Chapitre 5

Contrats - Spécification locale de composants

Dans ce chapitre nous présentons en détail le modèle de spécification par contrat (tel qu'introduit au paragraphe 3.6) pour les systèmes réactifs synchrones. Nous commençons pas présenter les contrats dans le cadre des langages orientés objets, fonctionnels ou matériels, avant de présenter les approches adoptées en général pour les systèmes réactifs (paragraphe 5.2). Nous introduisons ensuite les contrats pour les nœuds LUSTRE *pour lesquels nous donnons une définition en terme d'ensembles de traces (paragraphe 5.3). Nous étudions aussi la définition d'un contrat pour le composant résultant de la composition synchrone de composants à contrats, ainsi que la définition d'un contrat d'une itération à partir du contrat du noeud itéré.*
Une version condensée de ce chapitre a été publié dans [MM04a].

5.1 Introduction

Comme nous l'avons vu dans le chapitre 3, il paraît indispensable de pouvoir utiliser plusieurs formalismes pour décrire les comportements réactifs et de leur environnements. Si on veut décrire un environnement physique, on utilisera les automates LUCKY. Si l'on veut décrire un comportement complètement déterministe, on peut le faire en utilisant LUSTRE directement.

Nous voulons étudier maintenant une nouvelle forme de spécification qui nous permettrait de décrire des composants par leurs propriétés et leur exigences, de manière à pouvoir étudier des compositions avant que les composants soient complètement programmés. Il s'agit d'une forme de spécification qui décrit de manière non-déterministe un comportement de composant en séparant :

- d'un coté l'assertion (hypothèse) que fait le composant sur son environnement ;
- de l'autre ce que le composant garantit sur ses sorties lorsqu'il est placé dans un environnement compatible.

Cette séparation assertion/garantie est la base de la notion de *contrat*, d'abord proposée par Meyer [Mey92] pour les langages orientés objet.

En parallèle, on veut pouvoir se servir de ces spécifications non-déterministes pour la validation formelle des composants. Il faut donc que la notion de contrat qu'on introduit soit *fondée sémantiquement*. On pourra ensuite exploiter les spécifications par contrats pour aider à la conception de programmes corrects (voir chapitre 9). Une des applications des contrats en validation est la tech-

nique de vérification compositionnelle par raisonnement *assume/guarantee*.

Le présent chapitre a pour premier but de faire un tour d'horizon de la notion de contrat (section 5.2). Nous présenterons la notion de contrat telle qu'elle est définie dans les langages orientés objets, les langages fonctionnels ou encore dans certains formalismes de description de systèmes matériels. Dans le domaine de systèmes réactifs, l'accent n'est à notre avis pas suffisamment mis sur les moyens d'exprimer une telle forme de spécification dans les langages de programmation, bien qu'elle paraisse devenir incontournable dans le cadre d'une méthodologie de développement par composant.

Enfin, nous présenterons notre proposition de *contrats* pour les composants réactifs. Nous proposons de décrire le comportement des composants à l'aide de couples *Assume/Guarantee* spécifiant les hypothèses sur l'environnement et les garanties du composant vis-à-vis de cet environnement. La sémantique de ces contrats est donné en terme d'ensembles de traces et peut donc être comparée aux autres formes de spécification vues au chapitre 3. Nous définissons un *composant* par :

- une clause *assume* A portant sur les entrées du composant ;
- une clause *guarantee* G portant sur les entrées et les sorties du composant. Le couple (A, G) décrit le *contrat* du composant ;
- un *corps* B qui décrit le comportement déterministe du composant, lorsque celui-ci est entièrement construit.

Remarquons dès maintenant que l'assertion du contrat ne porte pas sur les sorties du composant. Il s'agit d'un choix important sur lequel nous reviendrons plus loin.

5.2 Travaux connexes

La notion de *spécification par contrat* a été introduite dans le cadre de la programmation orientée objet afin d'aider à la production de logiciels plus sûrs. [JTM99] présente 4 grandes catégories de contrats :

- les contrats syntaxiques (principalement représentés par les systèmes de types) ;
- les contrats fonctionnels (ou comportementaux) sur lesquels nous revenons ci-dessous;
- les contrats de synchronisation permettant de spécifier comment des composants peuvent communiquer dans un contexte de communication asynchrone ;
- les contrats de qualité de services.

Dans le cadre des composants réactifs synchrones, nous allons nous intéresser principalement aux *contrats fonctionnels*. Les contrats syntaxiques sont déjà généralement implémentés par les systèmes de typage des langages. Dans un langage synchrone, les contrats relatifs à la synchronisation ne sont en fait pas séparés des contrats fonctionnels. En effet, la synchronisation est une partie de la fonctionnalité. Enfin, nous n'avons pas étudié la notion de qualité de service pour des composants réactifs.

Un contrat fonctionnel est un couple de deux propriétés : l'assertion (appelée *assume* en anglais) et la garantie (*guarantee*). L'idée d'associer à des programmes où à des parties de programmes un couple de conditions *pre* et *post* décrivant l'état du composant avant et après l'exécution du programme a été proposée par Hoare dans [Hoa69]. Le but de la logique de Hoare est principalement la vérification de programmes séquentiels. L'idée d'utiliser un couple de *clauses assume* et *guarantee* pour *décrire* le comportement d'un composant (et non pas seulement pour établir des propriétés qu'il doit vérifier) est, elle, plus récente. La première définition précise des *contrats fonctionnels* (où l'on sépare clairement la spécification d'une fonctionnalité en un couple de clauses *pre* et *post*) a été donnée pour des programmes orientés-objets par Bertrand Meyer dans le cadre du langage Eiffel [Mey92]. Nous revenons sur les contrats des langages orientés objets dans le paragraphe 5.2.1.

Cette notion de contrat a naturellement été étendue à d'autres formes d'expressions. Nous présenterons rapidement dans 5.2.2 les principaux travaux concernant la définition de contrats *assume/guarantee* pour les langages fonctionnels. Dans le cadre des systèmes matériels, les *don't cares séquentiels* sont des relations permettant de décrire les valeurs d'entrées ou de sorties compatibles avec un composant donné. Nous détaillons cette approche en 5.2.3.

Ces différents travaux nous intéressent par le fait qu'ils portent surtout sur l'*expression* de spécifications *haut-niveau* et la séparation entre hypothèses sur l'environnement et garanties sur le comportement du composant. Les approches adoptées pour la description des systèmes réactifs (notamment les *Reactive Modules* et les *Interface Automata* que nous présentons en 5.2.4) n'adoptent pas en général cette séparation : on y décrit simplement des comportements non-déterministes sous la forme de relations entre variables d'entrées et de sorties. Il nous paraît pourtant fondamental du point de vue de la *spécification* des composants d'avoir une séparation claire entre clauses *assume* et *guarantee*.

5.2.1 Contrats pour la spécification par objets

Les langages orientés objet ont été le champ d'application le plus important des contrats. En général, leur définition se fait au travers d'un ensemble d'expressions qu'on associe à une classe (invariant) ou à une méthode (assertion et garantie).

Dans une classe C, une *assertion* est une propriété associée à une méthode M qui porte sur les attributs de la classe C ainsi que sur les paramètres de la méthode, et qui exprime une condition qui doit être satisfaite *au début de l'exécution de chaque appel* à M. De manière symétrique, une *garantie* associée à M porte sur les attributs de C et exprime une condition qui doit être satisfaite *à la fin de l'exécution de chaque appel* à M. On peut aussi définir un *invariant de classe* : on associe à C une propriété qui doit être satisfaite dans tous les état stable de l'objet, c'est-à-dire entre deux appels de méthodes.

EXEMPLE 24 — Considérons une classe Pile décrite en java dont les éléments sont simplement des entiers. Peu importe la représentation choisie, cette pile offre au moins 2 méthodes de manipulation et 1 méthode de consultation. Les manipulations possibles sont l'empilement d'un élément (méthode empiler(int elementAEmpiler)) et le dépilement du sommet de pile (méthode depiler()). La méthode de consultation sommetDePile() permet de récupérer la valeur du sommet de pile. On donne aussi une méthode qui renvoie le nombre d'éléments courant de la pile (nbElements()). On va considérer que le nombre maximum d'éléments de la pile est fixé, et donné par nbMaxElements().

Nous décrivons maintenant les *pre* et *post* conditions pour chacune de ces méthodes. Considérons tout d'abord la méthode sommetDePile. Récupérer le sommet de pile n'a de sens que lorsque la pile est non-vide. On doit donc supposer nbElements()>0 à chaque fois qu'on appelle cette méthode. Lorsqu'on veut dépiler le sommet de pile, on doit d'abord s'assurer que la pile n'est pas vide (nbElements()>0). Symétriquement, lorsqu'on empile un élément, on garantit qu'après l'empilement non seulement la pile n'est pas vide nbElements()>0) mais aussi que le nouveau sommet de pile est bien l'élément qu'on vient d'empiler. Comme le nombre d'éléments dans la pile est majoré, on doit s'assurer avant chaque appel à empiler que le nombre d'éléments déjà dans la pile n'est pas plus grand que le nombre maximum autorisé.

La classe Pile possède un invariant : à chaque appel d'une méthode de la classe (quelle que soit cette méthode), le nombre d'éléments de la pile est toujours positif ou nul. L'ensemble des signatures des méthodes de la classe Pile ainsi que les contrats associés sont donnés à la figure 5.1.

— FIN DE L'EXEMPLE 24

```
class Pile{
/** invariant nbElements>=0 **/

    public int nbElements(){
        . . .
    }
    public int nbMaxElements(){
        . . .
    }

    public void empiler(int elementAEmpiler){
    /** assume nbElements()<nbMaxElements();**/
    /** guarantee nbElements >0
    && sommetDePile()==elementAEmpiler; **/
        . . .
    }

    public void depiler(){
    /** assume nbElements!=0; **/
        . . .
    }

    public int sommetDePile(){
    /** assume nbElements!=0; **/
        . . .
    }
}
```

FIG. 5.1 – La classe Pile avec des contrats spécifiés dans le langage de l'outil *icontract*.

Le langage Eiffel [Mey92] a été le premier langage orienté objet à proposer l'utilisation des contrats. Il propose des éléments syntaxiques pour la définition des contrats. Pour Java, plusieurs extensions ont été proposées parmi lesquelles on peut citer : iContract [Kra98] (qui permet d'exprimer les contrats à l'aide de formules de la logique OCL [OCL97]), jContractor [KHB99] ou encore JML [LBR99]. Pour UML, on peut citer la logique OCL [OCL97] ainsi que la logique BOTL [DKR00] qui est une extension objet de CTL.

Ces langages permettent de décrire l'assertion d'une méthode comme une propriété portant sur l'état de l'objet au moment où l'on appelle la méthode. Jass [BFMW04] est une extension de Java qui permet d'exprimer des propriétés sur des *traces* d'événements et donc sur *l'histoire du comportement* du système. Ces assertions sont décrites à l'aide d'une syntaxe à la CSP [Hoa85] dans laquelle on peut appeler des opérations java. Cette approche semble être celle proposant l'expressivité la plus complète pour des contrats dans le cas de langages objet.

Utilisation des contrats – Les contrats présentent deux avantages majeurs. Tout d'abord, il s'agit d'une forme de spécification haut-niveau qui permet d'adopter une méthode progressive de développement :

- Le programmeur commence par décrire les objets et les interfaces des fonctionnalités associées. Il associe à chaque classe un invariant de classe ;
- Il donne ensuite une spécification générale de chaque fonctionnalité à l'aide d'un contrat ;
- Enfin il implémente chaque fonctionnalité en s'aidant du contrat associé (par raffinement par exemple).

Le second avantage des contrats est l'aide qu'ils apportent dans l'étape de validation des composants et du système entier. L'utilisation principale dans le domaine de la programmation orientée objet est la génération de code défensif à partir des contrats. Cela consiste à transformer ces assertions et garanties en tests (placés au début et à la fin de la méthode considérée) qui lèvent les exceptions appropriées en cas de violation du contrat. L'ajout de code peut diminuer considérablement les performances du code, mais on peut alors en limiter l'utilisation aux phases de développement et de débogage du programme. Lorsqu'une assertion ou une garantie est violée, on cherche à déterminer que est l'objet fautif. Si la précondition d'une méthode est violée, un interpréteur portera la faute sur l'appelant de la méthode ; si la postcondition est violée, l'interpréteur portera la faute sur l'appelé (la classe dans laquelle on trouve la méthode).

Plus récemment, Findler et al. [FF00, FMF01] ou encore iContract [Kra98] ou JMSassert [SYS] ont proposé une extension de la notion de contrat qui prend en compte l'utilisation d'*interfaces* en Java. Dans ce contexte, l'héritage pose certains problèmes quant à la désignation de l'objet responsable de la violation d'une assertion.

Dans le cas des assertions de traces de Jass, chaque assertion décrit un processus CSP. Lors de la simulation du système, on construit l'ensemble T des traces possibles de ce processus CSP. À chaque appel de la méthode concernée les nouvelles valeurs des variables du système sont ajoutées à la trace construite depuis le début de l'exécution. On vérifie au fur et à mesure que cette trace est bien un élément de T.

5.2.2 Contrats pour langages fonctionnels

Dans le cadre des langages fonctionnels, les contrats on été moins étudiés. Dans le cas simple des fonctions du premier ordre, l'utilisation des contrats est similaire au cas des langages objets. Prenons par exemple une fonction f définie par :

$$f\ : int \rightarrow int$$
$$rec\ f = \lambda x. \ldots$$

On peut par exemple associer à f un contrat stipulant comme hypothèse que l'entrée de la fonction est strictement supérieure à 9 et comme garantie que sa sortie est comprise entre 0 et 99. On peut écrire[1]:

$$f\ : int[> 9] \rightarrow int[0..99]$$

Dans un contexte d'évaluation dynamique des contrats, on peut vérifier au moment où f est appelée que son paramètre est bien plus grand que 9. Le compilateur *scheme Bigloo* [Ser04] permet de définir de tels contrats et, à partir de ceux-ci, génère du code défensif à la manière de ce qu'on nous avons décrit pour les langages objets.

Considérons maintenant la fonction g avec le contrat et la définition suivante :

$$g\ : (int[> 9] \rightarrow int[0..99]) \rightarrow int[0..99]$$
$$rec\ g = \lambda\ proc. \ldots$$

Dans ce cas l'évaluation du contrat de g est plus subtile : on n'a pas la valeur de $proc$ au moment où g est appelé et il se peut même qu'on ne connaisse pas cette valeur dans g elle-même mais seulement dans une fonction appelée par g. Dans [FF02], Findler et Felleisen proposent une méthode d'évaluation des contrats de fonctions d'ordre supérieur.

La notion de contrats définie dans [FF02] est très proche de celle que nous utilisons : on associe à chaque fonction un couple de 2 fonctions à résultat booléen : une fonction « *domain* » (correspondant à notre assertion) qui porte sur les entrées de la fonction et une fonction « *range* » (correspondant à notre garantie) qui porte à la fois sur les entrées et les sorties. Mais ces fonctions restent combinatoires : il n'existe pas de notions de traces des valeurs d'entrées/sorties et il n'est donc pas possible de parler de l'histoire du système.

Comme dans le cadre de l'approche orientée objet, les contrats sont utilisés essentiellement pour de la programmation défensive. Ils peuvent être aussi utilisés pour optimiser une fonction (comme dans le cas des systèmes matériels, voir paragraphe suivant 5.2.3). Par contre, aucune utilisation formelle (comme la comparaison d'un contrat avec le code de la fonction) n'est proposée.

5.2.3 Contrats pour les systèmes matériels : les *don't cares*

Dans le domaine de la conception des circuits, le concept de « *don't cares* » est la notion la plus proche des contrats. De manière générale, un « don't care » est un ensemble d'informations décrivant un réseau d'opérateurs qui peut admettre plusieurs implantations différentes. Les don't cares peuvent être décrits par :

- des réseaux d'opérateurs (comme dans [BBS95]) avec une sortie booléenne indiquant à chaque instant si les entrées/sorties courantes satisfont la spécification ;
- des BDDs ;
- de Machines d'États Finies [WB93, WB94]

En général, il existe une séparation claire entre une assertion sur les entrées d'un composant (appelée *input* ou *controllability* don't care) et une garantie portant sur les sorties du composant (appelée *output* ou *observability* don't care).

[1] Cet exemple et le suivant sont directement tirés de [FF02]

a0a1 \ a2a3	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	[0]	1
10	1	1	1	1

(a) En n'utilisant pas de don't cares.

a0a1 \ a2a3	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	[1]	1
10	1	1	1	1

(b) En utilisant un don't care.

FIG. 5.2 – Les tableaux de Karnaugh définissant la variable b.

Ces contrats sont le plus souvent utilisés pour la synthèse [BBS95] ou l'optimisation [BLK92] des circuits. Les utilisations pour la vérification des systèmes (des composants eux-mêmes ainsi que des compositions de composants) sont plus rares. Même si les auteurs de [BBS95] s'intéressent principalement à l'optimisation de circuits à l'aide des don't cares, ils évoquent (sans en donner de solution) l'intérêt des don't cares pour vérifier :

- qu'un composant satisfait bien la spécification décrite par le don't care correspondant ;
- qu'une composition de composants est possible vis-à-vis des contraintes imposées par les don't cares des composants.

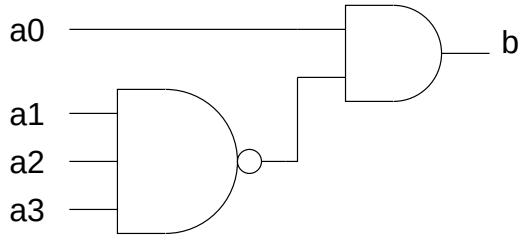
La limitation principale de ce travail est que les auteurs ne s'intéressent qu'à des circuits combinatoires. D'autres travaux s'intéressent à des circuits aussi bien combinatoires que séquentiels (voir [DM93]), mais ne s'attaquent pas à la vérification formelle des implémentations vis-à-vis des spécifications. Une autre limitation importante commune à tous les travaux sur les don't cares réside dans le fait que tous ces travaux considèrent des réseaux booléens (ce qui a une incidence sur les optimisations des composants).

Nous montrons maintenant l'utilisation des don't cares sur un exemple trivial (optimisable « à la main » très facilement).

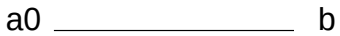
EXEMPLE 25 — On veut décrire un circuit qui prend 4 booléens a0, a1, a2, a3 (représentant un entier A en binaire) en entrée et renvoit 1 booléen b en sortie. b est vrai si A est compris entre 1 et 7. Si A est compris entre 8 et 14, b est faux. Dans tous les autres cas (il y a seulement deux, pour A=0 et A=15) la valeur de b n'est pas spécifiée. On peut admettre qu'on choisit la valeur par défaut de b à 0. Dans ce cas, la valeur de b est donnée par la table de Karnaugh de la figure 5.2(a). On peut en déduire le circuit de la figure 5.3(a) qui calcule exactement la valeur de b telle que spécifiée ci-dessus.

L'utilisation des don't cares permet d'optimiser un tel circuit. Dans le cas où A vaut 0 ou 15, on va simplement indiquer au compilateur que la valeur de b est 'indécise' (c'est une valeur « don't care »). Dans ce cas, il sera capable de déterminer que si on choisit d'assigner la valeur 1 à b lorsque A vaut 15 et 0 lorsque A vaut 0, le circuit se verra simplifié. La table de Karnaugh de la figure 5.2(b) montre en effet que b est vrai lorsque a0 est vrai. Le circuit correspondant, qui implémente bien la spécification donnée au départ est beaucoup plus simple (voir figure 5.3(b)).

— FIN DE L'EXEMPLE 25



(a) En n'utilisant pas de don't cares.



(b) En utilisant un don't care.

FIG. 5.3 – Les circuits calculant la variable b.

5.2.4 Comportements non-déterministes de composants réactifs

Comme nous l'avons souligné plusieurs fois, les approches adoptées pour les langages objets, fonctionnels ou de description de matériel sont limitées en terme d'exploitation de l'information apportée par les contrats. Dans le domaine des systèmes réactifs, la nécessité de décrire des comportements indéterministes apparaît très rapidement, ne serait-ce que pour décrire les environnements. Dans les approche que nous décrivons ci-dessous, l'accent est cependant plus porté sur les utilisations des descriptions non-déterministes que sur les moyens donnés au programmeur pour les exprimer.

Nous présentons maintenant brièvement les *Reactive Modules*, puis les *Interface Automata*, qui sont les plus proches de notre approche en termes de domaines d'applications et de pouvoir d'expression.

Les *Reactive Modules* [AH96] – ont été proposés pour décrire à la fois l'architecture et le comportement de systèmes matériels ou logiciels qui utilisent des communications synchrones ou asynchrones. Ce langage prend en compte 4 caractéristiques qui, d'après les auteurs, sont essentiels à la modélisation de tels systèmes :

- Le langage doit permettre la définition de comportements *indéterministes*, ce qui doit faciliter la description partielle des systèmes ;
- Il doit faciliter la description des interactions entre les composants d'un système en supportant la *concurrence* ;
- Un modèle d'exécution du langage doit être fourni afin de permettre une prototypage et une simulation précoce des systèmes (notamment lorsqu'ils sont encore partiellement définis) ;
- Enfin, ce langage doit avoir une sémantique mathématique clair pour qu'on puisse utiliser des méthodes de validation formelle telles que celles présentées au chapitre 2.

Les *Reactive Modules* permettent de décrire des comportements non-déterministes. En ce sens, ils se rapprochent de l'utilisation que l'on souhaite faire des contrats *assume-guarantee* en LUSTRE. Néanmoins, la notion exacte de contrat n'est pas présente : on ne peut pas séparer explicitement les hypothèses que fait un composant sur son environnement de ce qu'il garantit à cet environnement. Ceci s'explique par le domaine d'utilisation visé par Henzinger et al. pour les *Reactive Modules* qui doivent servir de format intermédiaire pour des langages de plus haut niveau. L'accent est plus porté sur les possibilités de manipulation des composants (tel que le *raisonnement assume-guarantee* que nous décrirons plus loin).

Les *Interface Automata* [dAH01b, dAH01a] – permettent de décrire des comportements de composants par des automates Entrées/Sorties. La manipulation des composants sous la forme assume/guarantee est possible mais seulement après extraction de ces clauses d'après l'automate : la clause *assume* (resp. *guarantee*) peut-être construite par projection de l'automate sur les variables d'entrées (resp. de sorties).

Un composant est représenté par une boîte à l'intérieur de laquelle se trouve l'automate décrivant le comportement du composant. Les canaux d'entrées et de sorties sont représentés par des cercles pleins portés sur le contour de la boîte. Les états de l'automate sont des cercles vides reliés par des flèches symbolisant les transitions.

EXEMPLE 26 — Considérons l'automate de la figure 5.4 tiré de [dAH01a]. Il s'agit d'un automate décrivant le comportement d'un composant qui implémente un service de transmission de messages. Le composant reçoit des messages par le canal msg et les renvoie par le canal $send$. Pour chaque envoi, il reçoit un compte-rendu du récepteur qui est positif (ack) ou négatif ($nack$). Selon le résultat de la transmission, il communique un résultat de transmission (positif avec le signal ok ou négatif avec le signal $fail$) à l'émetteur.

L'état initial 0 est un état dans lequel le composant attend le message à transmettre. Le composant reçoit un message par le canal msg. Il tente ensuite de l'envoyer une première fois (avec la commande $send!$). S'il reçoit une confirmation ($ack?$), il renvoie un signal $ok!$ et se remet en attente dans l'état initial 0. S'il reçoit un signal stipulant que le message n'a pas été reçu ($nack?$), il tente de l'émettre une seconde fois. Si une nouvelle erreur survient (nouvel $nack?$ en bas à droite), le transmetteur émet un message d'erreur ($fail!$).

Cet automate décrit le comportement du composant vu de l'extérieur. Il ne décrit pas comment la communication est réalisée exactement. De plus, on peut retrouver une séparation entre assertion et garantie en masquant les signaux respectivement de sorties (marqués d'un '!') ou d'entrées (marqués d'un '?'). Mais cette information n'est pas clairement disponible lorsqu'on observe l'automate : il faut l'extraire en parcourant le graphe. L'assertion du composant de la figure 5.4 est que les les signaux reçus en entrées par le composant sont, dans l'ordre, un message msg, puis soit un acquittement (ack) soit un non-acquittement ($nack$). Si il reçoit d'abord un $nack$, il attend une fois encore ack ou $nack$. Le comportement garanti est que, après voir envoyé le message (commande $send!$) :

- si ack est reçu, alors il émet ok ;
- si $nack$ est reçu alors il réémet le message, puis si il reçoit ack, il émet ok, ou, si il reçoit $nack$, émet $fail$.

— FIN DE L'EXEMPLE 26

5.3 Contrats pour des composants réactifs

Nous présentons maintenant une notion de contrat assume/guarantee pour les composants réactifs. Un composant est décrit par 3 ensembles de variables (entrées, sorties et locales), et 3 ensembles de traces (décrits sous la forme de step-relations présentées au chapitre 3) : un corps, une clause *assume* et une clause *guarantee*. Après un exemple de motivation, nous décrivons la sémantique des composants à contrat eux-mêmes ainsi que la composition synchrone de ces composants. Nous donnons ensuite la syntaxe des contrats en LUSTRE. Nous décrivons enfin comment construire un composant itératif map, red, fill ou map_red : nous définissons le contrat d'une itération en fonction du contrat du nœud itéré.

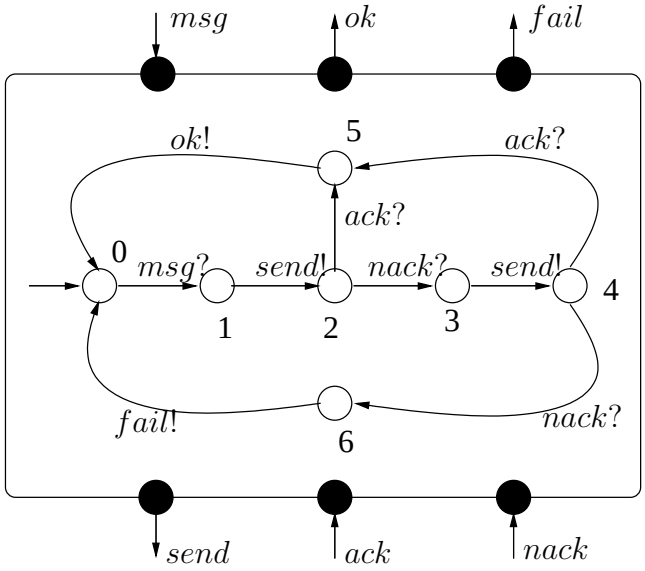


FIG. 5.4 – Un exemple d'*Interface Automaton*.

5.3.1 Motivations et exemple

Comme nous l'avons dit dans l'introduction de ce chapitre, nous voulons définir une notion de contrat pour des composants réactifs sous la forme de couple assertion/garantie.

Les contrats objet tels que nous les avons décrits en 5.2.1 permettent de manipuler une notion de temps particulière bien adaptée à la conception objet. Les pre et post conditions parlent d'une exécution de la méthode à laquelle elles se rattachent. En OCL, il est possible, dans une garantie, de spécifier qu'une variable x est égal à old x + 1 où old signifie « *juste avant cette exécution* ».

Nous avons besoin d'une autre notion de temps. Il peut être utile de parler d'exécutions précédentes de la même méthode. Par exemple, considérons une classe C avec 2 méthodes $m1$ et $m2$, et une spécification exprimant que $m1$ ne doit jamais être appelée avant que $m2$ n'ait été appelée au moins une fois. Une façon d'exprimer ce contrat consiste à attacher un état booléen explicite Σ. Une exécution de $m2$ doit *garantir* que Σ est bien mise à vrai. Une exécution de $m1$ *suppose* que Σ est vrai. En quelque sorte, la mémoire nécessaire à l'expression des clauses *assume* et *guarantee* d'un contrat a été encodée en une variable explicite de la classe.

Dans le cas des composants réactifs, le temps est une notion importante que nous devons pouvoir manipuler facilement dans les contrats. On doit pouvoir parler des exécutions successives d'une partie du code qui constitue *une* réaction du composant à son environnement. Une assertion sur l'environnement d'un composant (comme par exemple « *la température augmente* ») nécessite de pouvoir parler de valeurs successives de la température.

En fait, les contrats que nous allons manipuler vont en quelque sorte être similaires aux *invariants* de classe : ceux-ci décrivent une propriété qui doit être vraie à chaque fois que l'objet concerné est utilisé (c'est-à-dire à chaque instant de l'exécution), comme nos *assume* doivent être vérifiés à chaque fois que le composant est utilisé. La différence principale réside dans le fait que, pour nous, le « à chaque fois » est rythmé par l'horloge du système.

Pour l'instant, nous décrivons les contrats pour LUSTRE, leur syntaxe et leur sémantique et termes d'ensembles de traces. Ces contrats sont assez proches des spécifications assume/commit proposées

par Ketil Stølen [Stø96] ou Manfred Broy [Bro98] pour des réseaux flots de données.

Nous voulons un moyen de description de contrat qui :

- soit expressif : c'est-à-dire ayant toute la puissance d'un langage de trace comme décrit au chapitre 3 ;
- soit syntaxiquement facile à utiliser : on va donc proposer de décrire les contrats dans la même syntaxe que les composants eux-même ;
- sépare clairement l'assertion, la garantie et la description finale (l'implémentation) du composant : le contrat et l'implémentation doivent pouvoir exister séparément.

L'exemple suivant est typique des composants que nous voulons décrire. Nous l'utiliserons dans la suite du chapitre pour illustrer notre propos. Nous commençons par décrire le contrat d'un composant (figure 5.5) qui a une entrée booléenne a et une sortie booléenne b. Ce contrat indique que

« *si a est toujours vraie pendant au moins 2 instant consécutifs* »
alors
« *la sortie b est toujours vraie pendant au moins 3 instants consécutifs*. »

Chacune de ces propriétés peut être décrite à l'aide d'un automate accepteur qui reconnaît les séquences d'entrées (ou de sorties) qui satisfont la propriété. Tout autre style de description pourrait être utilisé ici.

L'assertion du composant que nous souhaitons décrire stipule que lorsque l'entrée a est vraie, elle l'est pendant au moins deux instants consécutifs. Cette propriété est représentée par l'automate de la figure 5.6. Tant que l'entrée a est fausse, l'assertion est vérifiée. Si l'entrée passe à vraie et reste dans cet état pendant au moins deux instants (transitions de $e0$ à $e1$ et de $e1$ à $e2$) l'assertion est toujours vérifiée. Si par contre l'entrée est vraie seulement pendant un instant (transitions de $e0$ à $e1$ puis de $e1$ à $e3$) l'assertion est violée. La garantie du composant (donnée à la figure 5.7) stipule que lorsque la sortie b est vraie, elle l'est pendant au moins trois instants consécutifs.

FIG. 5.5 – Un composant avec une entrée et une sortie booléennes.

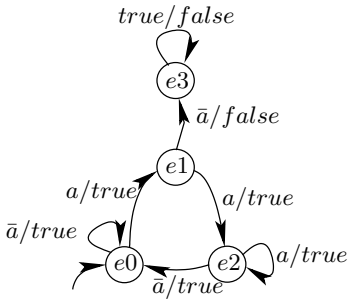


FIG. 5.6 – L'automate pour l'assertion du contrat de l'exemple.

L'écriture de ce contrat ne constitue qu'une première étape dans la description du composant. Il nous faut maintenant décrire son comportement exact. Pour cela, il faut détailler exactement comment b est

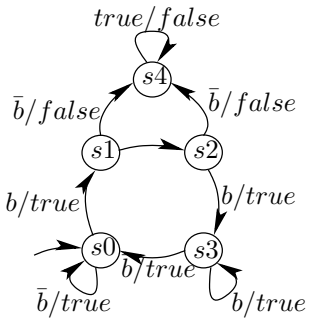


FIG. 5.7 – L'automate pour la garantie du contrat de l'exemple.

calculé en fonction de a. La figure 5.8 donne un exemple de composant implémentant le contrat. Il s'agit d'une bascule mono-stable redéclenchable (en anglais « *Retrigerrable Monostable Flip-Flop* », ou rMFF). Son fonctionnement est expliqué par le chronogramme de la figure 5.9. A chaque fois que son entrée est vraie, le composant rMFF met sa sortie à vrai, pendant un nombre de cycles donné (fixé à 2 dans l'exemple). Si l'entrée est vraie à nouveau pendant que la sortie est vraie, la sortie est *rechargée* et sera vrai pendant le même nombre de cycle à partir de cet instant.

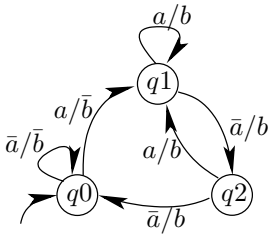


FIG. 5.8 – L'automate d'un composant rMFF réalisant le contrat.

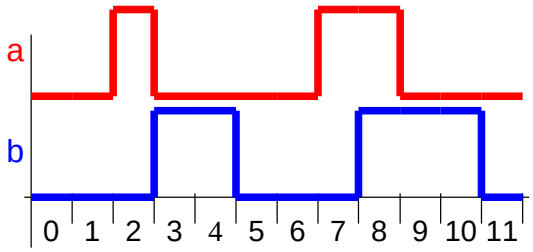


FIG. 5.9 – Chronogramme décrivant le comportement du composant rMFF.

Cet exemple nous permet d'illustrer le fait que nous devons disposer de la même capacité d'expression pour décrire les contrats que pour les composants eux-mêmes : il nous faut notamment pouvoir utiliser des mémoires locales aux assertions (ici, les états des deux automates) qui sont différentes du composant lui-même.

5.3.2 Composants - Sémantique

Définition 21 — Composant *IOLAG*

Un composant IOLAG *est un n-uplet* $(I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$ *où :*
- I *et* O *sont les ensembles d'entrées et de sorties du composant ;*
- L, A *et* G *sont les ensembles de variables respectivement locales au corps, à l'assertion et à la garantie du composant.* I, O, L, A *et* G *sont disjoints deux à deux ;*
- Sb *est une step-relation et* Jb *est une relation combinatoire toutes deux sur* $I \cup O \cup L$, *définissant le comportement du* corps *du composant ;*
- Sa *et* Sg *sont deux step-relations (et* Ja, Jb *sont deux relations combinatoires) respectivement sur* $I \cup A$ *et* $I \cup O \cup G$ *définissant le* contrat *du composant.*

Les relations Jb, Ja et Jg permettent de décrire le comportement du composant à l'instant initial de l'exécution. Les step-relations Sb, Sa et Sg décrivant son comportement par la suite.

On distingue les mémoires du contrat de la mémoire du corps du composant pour deux raisons :
- dans le schéma de développement *progressif* que nous privilégions, le contrat est décrit avant le corps du composant. Lorsqu'on le décrit, on ne sait pas encore de quelle mémoire on aura besoin pour décrire le corps ;
- les clauses A et G n'ont pas besoin de la même mémoire que le corps du composant. Dans l'exemple décrit en 5.3.1, la clause garantie est décrite par un automate dont le nombre d'états est différent de celui du corps du composant. Comme montré dans l'expression de ce composant donnée dans l'exemple ci-après, chaque état de ces automates est transformé en une variable locale de la clause correspondante (variables de G ou de A).

EXEMPLE 27 — Le rMFF de l'exemple du paragraphe 5.3.1 peut être décrit sous la forme d'un composant *IOLAG*. On commence par décrire les ensembles de variables d'entrées, de sorties et les variables locales du corps, de l'assertion et de la garantie.

$I = \{a\}$, $O = \{b\}$;

$L = \{q0, q1, q2\}$, $A = \{e0, e1, e2, e3\}$ et $G = \{s0, s1, s2, s3, s4\}$;

On décrit ensuite le contrat du composant à l'aide de deux step-relations Sa et Sg portant respectivement sur $\{a\} \cup \{e0, e1, e2, e3\}$ et sur $\{a\} \cup \{b\} \cup \{s0, s1, s2, s3, s4\}$. Sa et Sg décrivent à la fois les transitions entre états de l'automate correspondant et aussi le fait qu'on ne doit jamais être dans l'état $e3$ pour l'assertion et l'état $s4$ pour la garantie.

$$
\begin{aligned}
Sa \ : \ & e0' = ((e0 \wedge not\ a) \vee (e2 \wedge not\ a)) \\
& \wedge\ e1' = (e0 \wedge a) \\
& \wedge\ e2' = ((e1 \wedge a) \vee (e2 \wedge a)) \\
& \wedge\ e3' = ((e1 \wedge not\ a) \vee e3) \\
& \wedge\ not\ e3
\end{aligned}
$$

$$
\begin{aligned}
Sg \ : \ & s0' = ((s0 \wedge not\ b) \vee (se3 \wedge not\ b)) \\
& \wedge\ s1' = (se0 \wedge b) \\
& \wedge\ s2' = (s1 \wedge b) \\
& \wedge\ s3' = ((s2 \wedge b) \vee (s3 \wedge b)) \\
& \wedge\ s4' = ((s1 \wedge not\ b) \vee (s2 \wedge not\ b) \vee s4) \\
& \wedge\ not\ s4
\end{aligned}
$$

Deux relations combinatoires Ja et Jg permettent de décrire l'initialisation du contrat :

$Ja : e0$
$Jg : s0$

On décrit le comportement déterministe du composant à l'aide d'une step-relation Sb portant sur $\{a\} \cup \{b\} \cup \{q0, q1, q2\}$:

$$\begin{aligned} Sb \ : q0' &= (q0 \wedge not\ a) \vee (q2 \wedge not\ a) \\ \wedge\ q1' &= (q0 \wedge a) \vee (q1 \wedge a)(q2 \wedge a) \\ \wedge\ q2' &= q1 \wedge not\ a \\ \wedge\ b' &= (q1 \wedge a) \vee (q1 \wedge not\ a) \vee (q2 \wedge a) \vee (q2 \wedge not\ a) \end{aligned}$$

Enfin, on décrit l'initialisation du comportement avec la relation combinatoire Jb :

$Jb : q0$

Les définitions de Sb, Sa et Sg sont un simple encodage de l'automate correspondant. L'initialisation de l'automate est définie dans Jb, Ja et Jg.

— FIN DE L'EXEMPLE 27

Sémantique – Il existe 2 sémantiques possibles d'un composant *IOLAG* C. La première est utile lorsque le corps du composant est connu. On peut alors définir sa sémantique par un ensemble de traces sur $I \cup O \cup L$ décrit par (Sb, Jb). On a alors :

$$\tau_{Composant}(C) = \tau_{StepRel}(Sb, Jb)$$

Dans les phases précoces du processus du développement, il est possible qu'on ne connaisse pas le corps (Sb, Jb) (le développeur peut n'avoir donné que le contrat). On va alors s'intéresser à la sémantique du composant défini par son contrat, c'est-à-dire par les couples (Sa, Ja) et (Sg, Jg). L'ensemble de traces correspondant est alors donné par (voir définition en 3.6) :

$$\tau_{Composant}(C) = \tau_{Contrat}((Sa, Ja), (Sg, Jg)).$$

Notations – Nous définissons maintenant le renommage d'un composant basé sur le renommage des step-relations du paragraphe 3.4.6.3. Nous aurons besoin de cette définition notamment lors de la définition des contrats de composants itératifs.

Définition 22 — Renommage d'un composant

Soit un composant IOLAG $C = (I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$*. Soit un sous-ensemble* V *des variables de* $I \cup O$ *et soit un ensemble* W *de variables tels que* $|W| = |V|$*. On définit le renommage de* C *dans* W *le composant noté* $C_{[V \leftarrow W]}$ *comme le composant* C *où toutes les occurrences de variables de* V *ont été remplacée par des occurrences de variables correspondantes de* W *(aussi bien dans les ensembles de variables* I *et* O *que dans* toutes *les step-relations décrivant le composant). On a :*

$$\begin{aligned} C_{[V \leftarrow W]} \quad = \quad & ((I \setminus V) \cup W, \\ & (O \setminus V) \cup W, \\ & L, A, G, \\ & Sa_{[V \leftarrow W]}, Ja_{[V \leftarrow W]}, \\ & Sg_{[V \leftarrow W]}, Jg_{[V \leftarrow W]}, \\ & Sb_{[V \leftarrow W]}, Jb_{[V \leftarrow W]}) \end{aligned}$$

5.3.3 Pourquoi la clause *assume* ne parle pas des sorties

Comme nous l'avons déjà souligné plus haut, la clause *assume* du contrat d'un composant ne porte pas sur ses sorties. Ce choix se base sur deux remarques essentielles. La première est que l'utilisation des variables de sortie d'un composant dans la clause *assume* de son contrat se fait toujours en parlant du passé de ces variables. En effet, le comportement à un instant donné d'un système ne peut pas se faire (être contraint) sous des hypothèses qui portent sur les valeurs de sorties qu'il calcule dans l'instant courant : ces hypothèses réaliserait une sorte de « bouclage sémantique » (les valeurs des sorties se contraindraient elles-mêmes). Si la clause *assume* devait parler des sorties, elle parlerait donc du *passé strict* de ces sorties. Cela pourrait se faire de deux manière :

- La première consiste à supposer que chaque occurrence d'une variable de sortie dans l'assertion d'un contrat représente sa valeur précédente. Cette solution n'est pas envisageable car elle créé une confusion de sens entre les occurrences des variables de sorties dans le corps du programme et dans l'assertion (une occurrence d'une sortie ne représente pas la valeur de cette sortie au même instant dans les deux contextes) ;
- La seconde solution (plus réaliste) consiste à forcer l'utilisation de pre sur chaque occurrence d'une variable de sortie dans l'assertion. Cette solution paraît plus naturelle, mais entraîne des vérifications supplémentaires.

La deuxième remarque est que *l'utilisation des sorties dans le assume n'est* en fait *pas indispensable* en terme d'expressivité. Si on veut en effet parler d'une sortie du composant dans son assertion, on peut toujours extraire du composant le sous-graphe d'opérateurs calculant cette sortie et l'inclure comme variable locale de l'assertion.

L'utilisation des sorties dans le assume n'étant pas indispensable et compliquant certainement la description ou l'analyse du contrat, nous avons choisi de ne pas la permettre. L'assertion d'un contrat ne porte donc jamais sur les sorties du composant.

5.3.4 Cohérence

Nous donnons maintenant une condition essentielle à la validité d'un contrat. Cette condition porte sur la cohérence des step-relations du contrat vis-à-vis des ensembles de variables.

Définition 23 — Cohérence des step-relations vis-à-vis des ensembles de variables

> *Étant donné un composant $\mathcal{C} = (I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$, les step-relations et relations combinatoires de $\mathcal{C}$ sont dites cohérentes vis-à-vis de ses ensembles de variables si et seulement si la condition suivante est satisfaite :*
>
> $$(Sa, Ja)\lfloor I \rfloor \subseteq (Sg, Jg)\lfloor I \rfloor \tag{5.1}$$

La condition 5.1 signifie que les comportements des variables de I autorisés par l'assertion (c'est-à-dire les traces de $\tau_{StepRel}((Sa, Ja)\lfloor I \rfloor)$ où l'on a caché les variables locales de l'assertion) sont tous des comportements autorisés par la garantie. En d'autres termes, la garantie ne contraint pas les comportements des entrées I plus que l'assertion ne le fait.

La garantie d'un composant est utilisée pour spécifier le comportement des sorties du composant. Pour exprimer ce comportement on autorise l'utilisation des entrées (on ne rencontre pas les mêmes problèmes que lorsqu'on autorise l'utilisation des sorties dans la clause *assume*) dans la clause guarantee. Cette clause guarantee ne doit pas pour autant être utilisée comme une contrainte sur les entrées. Au moins, elle ne doit pas contraindre les entrées plus que la clause *assume*. On exige la cohérence d'un contrat vis-à-vis des variables d'entrées et de sorties du composant comme un *critère de qualité*

du contrat : une garantie contraignant plus les entrées que ne le fait l'assertion correspondante est, en quelque sorte, en contradiction avec son propre rôle.

5.3.5 Composition synchrone de composants

Nous définissons maintenant la *composition parallèle synchrone* de 2 composants. Dans le cas général, les composants que nous venons de définir sont assemblés en réseau flot-de-donnée : les sorties d'un composant peuvent être connectées directement aux entrées d'un autre composant. On ne peut définir de cycle combinatoire (dépendance cyclique instantanée), mais on peut définir des cycles non combinatoires en utilisant un composant particulier (le pre de LUSTRE étendu avec une initialisation) qui mémorise son entrée pendant une étape de l'exécution du composant.

Ce composant pre peut être défini simplement par :

$$(I = \{init, i\}, O = \{o\}, L = \{m\}, A = \emptyset, G = \emptyset,$$
$$Sa : true, Sg : true, Sb : m' = i \wedge o = m, Ja : true, Jg : true, Jb : o = init).$$

où on initialise la mémoire avec la première valeur de l'entrée $init$.

Les composants ainsi connectés effectuent tous une étape de calcul en même temps et peuvent échanger des valeurs de manière instantanée. Dans la suite, on connecte deux composants en identifiant les variables d'entrée/sortie qui portent le même nom dans les deux composants. On utilise les notations sur les step-relations introduites au paragraphe 3.4.6.3, page 59.

Définition 24 — Composition synchrone

Étant donnés 2 composants $\mathcal{C}_i = (I_i, O_i, L_i, A_i, G_i, Sa_i, Ja_i, Sg_i, Jg_i, Sb_i, Jb_i)$ *(avec* $i = 1, 2$*) tels que :*

- *ils ont des ensembles de sorties disjoints (une variable ne peut pas être définie 2 fois) :* $O_1 \cap O_2 = \emptyset$ *;*
- *il n'existe pas de cycle combinatoire, c'est-à-dire que pour chaque variable* $v_1 \in I_1 \cap O_2$ *et chaque variable* $v_2 \in I_2 \cap O_1$*, soit* v_1 *ne dépend pas de* v_2 *(dans* $\mathcal{C}_1$*), soit* v_2 *ne dépend pas de* v_1 *(dans* $\mathcal{C}_2$*) ;*

on peut définir la composition parallèle synchrone *de* $\mathcal{C}_1$ *et* $\mathcal{C}_2$ *— que l'on note* $\mathcal{C}_1 \times \mathcal{C}_2$ *— comme étant le composant* $(I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$ *tel que :*

- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ *;*
- $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$ *;*
- $L = (L_1 \cup L_2) \cup ((I_1 \cup I_2) \cap (O_1 \cup O_2))$ *;*
- $A = A_1 \cup A_2$*;* $G = G_1 \cup G_2$ *;*
- $(Sa, Ja) = (Sa_1, Ja_1)\lfloor I_1 \setminus O_2 \rfloor \cap (Sa_2, Ja_2)\lfloor I_2 \setminus O_1 \rfloor$ *;*
- $(Sg, Jg) = (Sg_1, Ja_1)\lfloor O_1 \setminus I_2 \rfloor \cap (Sg_2, Ja_2)\lfloor O_2 \setminus I_1 \rfloor$ *;*
- $(Sb, Jb) = (Sb_1 \cap Sb_2, Jb_1 \cap Jb_2)$.

Les variables utilisées pour connecter les 2 composants ne sont pas visibles par l'environnement de la composition. Elles ne font pas partie des entrées puisqu'elles sont déjà définies par le comportement d'un des deux composants. Ces variables sont donc masquées dans la définition de l'assertion de la composition ($Sa = Sa_1\lfloor V \setminus O_2 \rfloor \cap Sa_2\lfloor V \setminus O_1$). Elle ne sont pas visibles non plus en tant que sorties de la composition. Cela implique qu'on ne peut plus parler de ces variables dans la garantie de la composition (d'où $Sg = Sg_1\lfloor V \setminus I_2 \rfloor \cap Sg_2\lfloor V \setminus I_1 \rfloor$).

La composition des corps de composants est la composition synchrone standard. L'assertion construite accepte des entrées qui satisfont la conjonction des assertions des deux composants. Sa

garantie exprime que ses sorties doivent satisfaire les garanties des deux composants. Par ailleurs, le masquage des variables utilisées pour connecter les deux composants a un effet sémantique sur la composition : ces variables ne sont plus contraintes dans le nouveau contrat.

Le contrat que nous avons construit pour la composition est *un* contrat possible. Il correspond en quelque sorte au contrat le plus précis qu'on peut construire à partir des deux contrats composés. Le développeur pourra raffiner ce contrat, ou en proposer un autre. On pourra toujours se servir du contrat de la composition comme d'une référence pour valider les contrats proposés ultérieurement.

5.3.6 Exemples

Nous donnons maintenant deux exemples de compositions. Le premier est très simple : il montre la composition de trois instances d'un composant réalisant l'addition de deux entiers. Le second met en oeuvre toutes les formes de branchements que l'on peut rencontrer dans une composition.

```
node assumePlus(acc, elt : int)
returns (assumeOK : bool);
let
   assumeOK = acc>0 and elt>0;
tel

node guaranteePlus(acc, elt, acc' : int)
returns (guaranteeOK : bool);
let
   guaranteeOK = acc'>0;
tel
```

FIG. 5.10 – Un contrat pour le nœud plus.

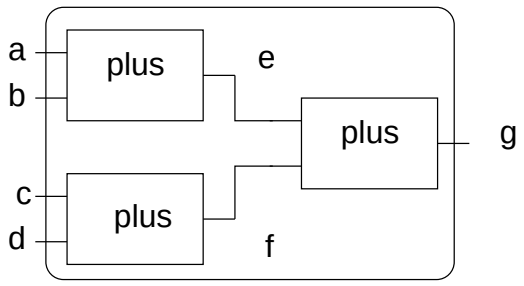


FIG. 5.11 – Composition de 3 instances de plus.

EXEMPLE 28 — On considère le composant plus réalisant la somme de deux entiers. On peut associer un contrat à ce nœud selon lequel il n'accepte que des entrées positives et que sous cette hypothèse la sortie qu'il fournit est forcément positive. Ce contrat est décrit à l'aide des nœuds assumePlus et guaranteePlus de la figure 5.10. On souhaite réaliser un composant qui calcule la somme de 4 entiers. Pour cela, on propose de composer 3 instances du composant plus comme montré à la figure 5.11. Un contrat possible de ce nouveau composant, construit automatiquement par la composition synchrone (voir figure 5.12), stipule que si les 4 entrées a,b,c et d sont positives alors sa sortie g est positive aussi.

— FIN DE L'EXEMPLE 28

EXEMPLE 29 — La fig. 5.13 montre les connexions possibles entre 2 composants $\mathcal{C}_1$ et $\mathcal{C}_2$ définis avec les ensembles d'entrées, de sorties et de variables locales suivants : $I_1 = \{i_1, j_1\}$, $O_1 = \{o_1, j_2\}$, $L_1, A_1, G_1 = \emptyset$ et $I_2 = \{i_2, j_2\}$, $O_2 = \{o_2, j_1\}$, $L_2, A_2, G_2 = \emptyset$. Les contrats sont donnés par les step-relations et relations combinatoires ci-dessous :

```
node assumeCompoPlus(a,b,c,d : int)
returns (assumeOK : bool);
let
   assumeOK = a>0 and b>0 and c>0 and d>0;
tel

node guaranteeCompoPlus(a,b,c,d,g : int)
returns (guaranteeOK : bool);
let
   guaranteeOK = g>0;
tel
```

FIG. 5.12 – Un contrat pour la composition de plus.

$$
\begin{array}{l}
Ja_1(i_1, j_1) \ : true \\
Sa_1(i_1, j_1) \ : j'_1 \geq 0 \vee (j'_1 < 0 \wedge i'_1 < 0); \\
Jg_1(i_1, j_1, o_1, j_2) \ : true \\
Sg_1(i_1, j_1, o_1, j_2) \ : j'_2 > 0 \wedge o'_1 > 0. \\
Ja_2(i_2, j_2) \ : true \\
Sa_2(i_2, j_2) \ : j'_2 > 0 \vee (j'_2 \leq 0 \wedge i'_2 < j'_2); \\
Jg_2(i_2, j_2, o_2, j_1) \ : true \\
Sg_2(i_2, j_2, o_2, j_1) \ : j'_1 > 0 \wedge o'_2 > j'_1.
\end{array}
$$

Le composant $\mathcal{C}_1$ suppose que :

- soit j_1 est positif ou nul ;
- soit j_1 et i_1 sont tous deux strictement négatifs.

Il garantie que j_2 et o_1 sont tous deux positifs. Le composant $\mathcal{C}_2$ suppose que :

- soit j_2 est strictement positif ;
- soit j_1 est négatif ou nul et i_1 est strictement négatif.

Il garantie que j_1 est strictement positif et que o_2 est strictement plus grand que j_1.

On suppose que les corps des composants satisfont le contrat correspondant. Nous souhaitons maintenant construire la composition décrite à la figure 5.13. Grâce à la définition de la composition, on peut donner *un* contrat pour cette composition. On donne ainsi Sa et Sg :

$$
\begin{array}{l}
Sa(i_1, i_2) \ : \exists j_1, j_2.(j_1 > 0 \vee (j_1 < 0 \wedge i'_1 < 0)) \wedge (j'_2 > 0 \vee (j'_2 \leq 0 \wedge i'_2 < j'_2)) \\
Sg(i_1, i_2, o_1, o_2) \ : \exists j_1, j_2.(j'_2 > 0 \wedge o'_1 > 0) \wedge (j'_1 > 0 \wedge o'_2 > j'_1)
\end{array}
$$

Remarquons que :

- ce contrat exprime des contraintes sur les entrées sorties de la composition (i_1, i_2, o_1 et o_2). On a masqué les variables de branchements j_1 et j_2 ;
- les occurrences de j_1 correspondant aux définition de Sa_1 et Sg_1 font toutes références à la valeur passée de cette variable, ce qui est imposé par l'utilisation de l'opérateur pre dans la connexion de $\mathcal{C}_1$ et $\mathcal{C}_2$.

— FIN DE L'EXEMPLE 29

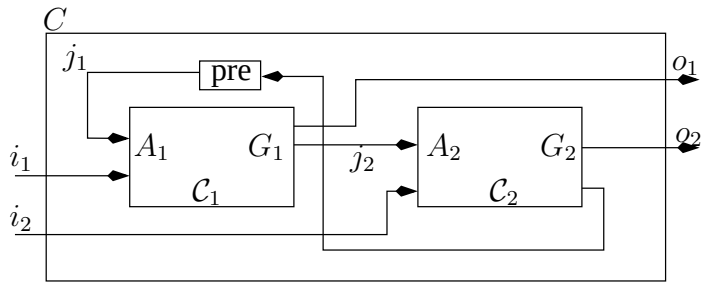


FIG. 5.13 – Un exemple de composition.

5.3.7 Syntaxe des contrats en LUSTRE

D'un point de vue pratique, les contrats pour LUSTRE sont définis comme des couples d'observateurs (A, G). A dénote l'assertion associée au nœud considéré et exprime une propriété de sûreté portant sur les entrées du nœud. G dénote la garantie du nœud et exprime une propriété de sûreté portant à la fois sur les entrées et les sorties du nœud.

Les nœuds A et G possèdent des variables locales (éventuellement indéfinies) et un ensemble d'équations définissant :

- la validité de la propriété de sûreté qu'ils représentent ;
- les valeurs des variables locales définies. La traduction de la forme step-relations vers observateurs est directe comme vu au chapitre 3.

EXEMPLE 30 — Nous présentons à la figure 5.14 la traduction en observateurs LUSTRE de l'assertion et de la garantie du contrat décrit en 5.3.1. Le nœud assumeRMFF code l'automate de l'assertion (guaranteeRMFF celui de la garantie). La traduction des automates en nœuds LUSTRE est simple : on créé une variable locale booléenne pour chaque état. L'expression utilisée pour calculer cette variable est toujours constituée d'une initialisation (vraie seulement pour l'état initial) et d'une disjonction décrivant sous quelles conditions cet état est atteint. Par exemple, dans le nœud *assume* la variable e0 est définie par l'équation :

```
e0 = true -> (pre(e0) and not pre(a)) or (pre(e2) and not pre(a));
```

qui stipule que e0 est l'état initial de l'automate (true -> ...) et qu'il est atteint si, à l'instant précédent :

- soit on était dans l'état e0 et a était faux ;
- soit on était dans l'état e2 et a était faux.

— FIN DE L'EXEMPLE 30

```
node assumeRMFF(a : bool) returns (assumeOK : bool);
var e0,e1,e2,e3 : bool;
let
   e0 = true -> (pre(e0) and not pre(a)) or (pre(e2) and not pre(a));
   e1 = false -> pre(e0) and pre(a);
   e2 = false -> (pre(e1) and pre(a)) or (pre(e2) and pre(a));
   e3 = false -> (pre(e1) and not pre(a)) or e3;
   assumeOK = not e3;
tel


node guaranteeRMFF(a,b : bool) returns (guaranteeOK : bool);
var s0,s1,s2,s3,s4 : bool;
let
   s0 = true -> (pre(s0) and not pre(b)) or (pre(s3) and not pre(b));
   s1 = false -> pre(s0) and pre(b);
   s2 = false -> pre(s1) and pre(b);
   s3 = false -> (pre(s2) and pre(b)) or (pre(s3) and pre(b));
   s4 = false -> (pre(s1) and not pre(b)) or (pre(s2) and not pre(b))
                 or pre(s4);
   guaranteeOK = not s4;
tel
```

FIG. 5.14 – Les nœuds LUSTRE pour l'assertion et la garantie du contrat de l'exemple rMFF.

Nous avons défini ce contrat à l'aide de 2 automates pour faciliter la compréhension du comportement de ce composant, mais on peut imaginer utiliser des compteurs pour implémenter chacune des propriétés. Pour décrire un contrat, on dispose de toute la puissance d'expression du langage LUSTRE.

5.3.8 Composition de nœuds à contrats en LUSTRE

Nous montrons à présent comment réaliser la composition synchrone (présentée au paragraphe 5.3.5) sur des nœuds LUSTRE. Bien sûr, on sait composer deux nœuds LUSTRE par simple identification des variables de branchements. Ce qui nous intéresse plus ici est la génération du contrat pour la composition de deux nœuds à contrats.

La composition décrite au paragraphe 5.3.5 fait apparaître dans le contrat de la composition de deux composants des variables quantifiées existentiellement. Dans la composition de nœuds LUSTRE, nous devrons générer des variables locales indéfinies. Les utilisations possibles de ces nœuds sont donc particulières : on ne peut pas les compiler directement, ni les fournir en entrées des outils de vérification. Nous proposerons plus loin des pistes pour les prendre en compte dans l'interprétation et la validation des composants.

Nous montrons pour l'instant le contrat obtenu en LUSTRE pour l'exemple de la figure 5.13. D'un point de vue pratique, les contrats étant décrits à l'aide de nœuds, on utilise ces derniers directement dans la définition des contrats de la composition.

EXEMPLE 31 — Nous considérons deux composants décrits par leur interface LUSTRE. C1 est défini par :

node C1(i1,j1 :int) **returns** (o1,j2 : int);

et C2 par :

node C2(i2,j2 :int) **returns** (o2,j1 : int);

Nous donnons tout d'abord à la figure 5.15 les nœuds définissant les contrats des composants $\mathcal{C}_1$ (assumeC1 et guaranteeC1) et $\mathcal{C}_2$ (assumeC2 et guaranteeC2). La composition que nous définissons a pour interface :

node compo(i1,i2) **returns** (o1,o2);

Le contrat correspondant (pour lequel nous avions donné les step-relations au paragraphe 5.13) est défini par les nœuds assumeCompo et guaranteeCompo de la figure 5.16. On notera l'utilisation de variables locales indéfinies j1 et j2 correspondant aux quantifications existentielles sur j_1 et j_2 de la figure 5.13.

— FIN DE L'EXEMPLE 31

```
node assumeC1(i1,j1 : int) returns (assumeOK : bool);
let
   assumeOK = true -> j1 >= 0 or (j1<0 and i1<0);
tel

node guarateeC1(i1,j1,o1,j2 : int) returns (guaranteeOK : bool);
let
   guaranteeOK = true -> j2>0 and o1>0;
tel

node assumeC2(i2,j2 : int) returns (assumeOK : bool);
let
   assumeOK = true -> j2>0 or (j2 <= 0 and i2<j2);
tel

node guaranteeC2(i2,j2,o2,j1 : int) returns (guaranteeOK : bool);
let
   guaranteeOK = true -> j1>0 and o2>j1;
tel
```

FIG. 5.15 – Les nœuds LUSTRE décrivant les contrats de C1 et C2.

5.3.9 Composants itératifs

Nous avons défini au chapitre 4 les itérateurs de tableaux pour LUSTRE. Nous donnons maintenant une définition des itérateurs de tableaux basée sur la notion de composant que nous avons défini plus haut. Nous définissons le contrat d'une itération à partir du contrat du nœud itéré. Grâce à cette définition, on peut considérer les itérations comme des composants à contrats à part entière et, notamment, leur appliquer les manipulations présentées au chapitre 9.

Le contrat d'une itération est ici construit itérativement à partir du contrat du nœud itéré.

```
node assumeCompo(i1,i2 : int) returns (assumeOK : bool);
var j1,j2 : int;
let
   assumeOK = true -> assumeC1(i1,pre(j1)) and assumeC1(i2,j2);
tel

node guaranteeCompo(i1,i2,o1,o2) returns (guaranteeOK : bool);
var j1,j2 : int;
let
   guaranteeOK = true -> guaranteeC1(i1,pre(j1),o1,j2)
                              and guaranteeC2(i2,j2,o2,j1);
tel
```

FIG. 5.16 – Les nœuds LUSTRE décrivant un contrat de compo.

EXEMPLE 32 — [Construction du contrat d'une itération simple] Soit le nœud plus défini au paragraphe 4.3.2.1. Au paragraphe 5.3.5, on a associé un contrat à ce nœud, stipulant que si les entrées sont toutes deux positives, alors la sortie du nœud est positive aussi.

Considérons maintenant une première itération du nœud plus avec un map. Cette nouvelle opération map≪plus;taille≫ prend en entrées deux tableaux d'entiers et rend un tableau d'entiers, tous de taille taille. À partir du contrat du de plus on peut construire un contrat pour l'itération map≪plus;taille≫. Son assertion stipule que tous les éléments des tableaux d'entrées sont positifs. Sa garantie stipule que les éléments du tableaux de sortie sont tous positifs également. Les deux nœuds spécifiant ce contrat, assumeMapPlus et guaranteeMapPlus sont donnés à la figure 5.17. L'opérateur forall permet de spécifier que tous les éléments satisfont la même propriété. Il sera présenté plus loin, au paragraphe 8.2.1. Le nœud unPositif permet de spécifier qu'une valeur entière est positive. Il est défini au paragraphe 8.2.2.1.

On souhaite maintenant construire l'itération du nœud plus à l'aide d'un red. Le composant ainsi construit (red≪plus;taille≫), initialisé avec la valeur 0 permet de calculer la somme des éléments d'un tableau de taille taille (il s'agit de l'itération déjà présentée au paragraphe 4.3.2.1). L'assertion de ce nouveau composant exprimera la propriété « les éléments du tableau d'entrée sont tous positifs *et* la constante 0 est positive. » Sa garantie exprime que « Le résultat de la réduction est positif ». Ce contrat est donné à la figure 5.18.

— FIN DE L'EXEMPLE 32

Pour chaque itération, on peut construire automatiquement un contrat à partir du contrat du nœud itéré. Nous donnons dans les paragraphes suivants des règles de construction pour chacun des itérateurs map, red, fill et map_red. A chaque fois on considère le nœud itéré spécifié comme un composant défini au paragraphe 5.3.2. Le composant correspondant à l'itération est construit récursivement à partir du composant itéré :

- Une itération de taille 1 est simplement une instance du nœud itéré ;
- Une itération de taille n est construite en composant une instance du nœud itéré avec l'itération de taille n-1.

Comme dans le cas de la composition, le contrat construit pour une itération est *un* contrat possible, le contrat le plus précis qu'on peut construire à partir du contrat du noeud itéré.

```
node assumeMapPlus(Tab1, Tab2 : intˆtaille)
returns (assumeOK : bool);
let
   assumeOK = forall≪assumePlus;taille≫(Tab1,Tab2);
tel

node guaranteeRedPlus(Tab1, Tab2, TabOut : intˆ)
returns (guaranteeOK : bool);
let
   guaranteeOK = forall≪unPositif;taille≫(TabOut);
tel
```

FIG. 5.17 – Un contrat pour l'itération map≪plus;taille≫.

```
node assumeRedPlus(init : int; Tab2 : intˆtaille) returns (assumeOK : bool);
let
   assumeOK = init>0 and forall≪unPositif;taille≫(Tab2);
tel

node guaranteeRedPlus(init : int; Tab : intˆtaille; res : int)
returns (guaranteeOK : bool);
let
   guaranteeOK = res>0;
tel
```

FIG. 5.18 – Un contrat pour l'itération red≪plus;taille≫.

5.3.9.1 red

Soit un composant $C = (I, O, L, A, G, Ja, Sa, Jb, Sb, Jg, Sg)$ tel que $I = \{a_{in}\} \cup e$ et que $O = (a_{out})$ (où a_{in} et a_{out} représentent les accumulateurs d'entrée et de sortie et e les entrées correspondant aux éléments de tableaux) et soit un entier s (la taille de la réduction). On définit le composant $Red(C, s)$ avec :

- pour entrée l'ensemble $I_R = \{init\} \cup \{It_0, \ldots, It_{s-1}\}$ où $init$ correspond à la variable d'initialisation et It aux entrées tableaux ;
- pour sortie l'ensemble $O_R = \{out\}$ où out est l'accumulateur de sortie.

$Red(C, s)$ est défini inductivement par composition synchrone du composant C :

- $Red(C, 1) = C_{[a_{in} \leftarrow init, e \leftarrow It_0, a_{out} \leftarrow out]}$;
- $Red(C, s+1) = Red(C, s)_{[out \leftarrow acc]} \times C_{[a_{in} \leftarrow acc, e \leftarrow It_s, a_{out} \leftarrow out]}$.

La construction du composant red est illustrée à la figure 5.19.

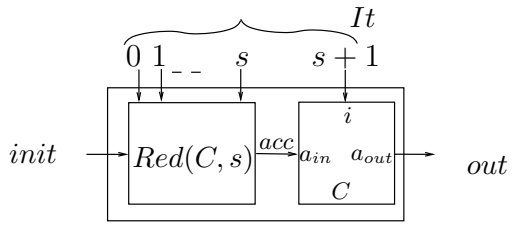


FIG. 5.19 – Définition récursive du composant red.

5.3.9.2 map

Soit un composant $C = (I, O, L, A, G, Ja, Sa, Jb, Sb, Jg, Sg)$ tel que $I = e$ et $O = f$ (où e (resp. f) représente les entrées (resp. sorties) correspondant aux éléments de tableaux). Soit un entier s. On définit le composant $Map(C, s)$ avec :

- pour entrée l'ensemble $I_M = \{It_0, \ldots, It_{s-1}\}$, où It représente les entrées tableaux du map ;
- pour sortie l'ensemble $O_M = \{Ot_0, \ldots, Ot_{s-1}\}$, où Ot représente les sorties tableaux du map.

$Map(C, s)$ est défini inductivement à partir du composant C :

- $Map(C, 1) = C_{[e \leftarrow It_0, f \leftarrow Ot_0]}$;
- $Map(C, s+1) = Map(C, s) \times C_{[e \leftarrow It_s, f \leftarrow Ot_s]}$

5.3.9.3 fill

Soit un composant $C = (I, O, L, A, G, Ja, Sa, Jb, Sb, Jg, Sg)$ tel que $I = \{a_{in}\}$ et $O = \{a_{out}\} \cup f$ (où e (resp. f) représente les entrées (resp. sorties) correspondant aux éléments de tableaux et a_{out} correspond à l'accumulateur de sortie. Soit un entier s. On définit le composant $Fill(C, s)$ avec :

- pour entrée l'ensemble $I_M = \{init\}$;
- pour sortie l'ensemble $O_M = \{out\} \cup \{Ot_0, \ldots, Ot_{s-1}\}$.

$Fill(C, s)$ est définit récursivement à partir de C :

- $Fill(C, 1) = C_{[a_{in} \leftarrow init, a_{out} \leftarrow out, f \leftarrow Ot_0]}$;
- $Fill(C, s+1) = Fill(C, s)_{[out \leftarrow acc]} \times C_{[a_{in} \leftarrow acc, a_{out} \leftarrow out, f \leftarrow Ot_s]}$.

5.3.9.4 map_red

Soit un composant $C = (I, O, L, A, G, Ja, Sa, Jb, Sb, Jg, Sg)$ tel que $I = \{a_{in}\} \cup e$ et $O = \{a_{out}\} \cup f$ (où $\{a_{in}\}$ (resp. $\{a_{out}\}$) représente l'entrée (resp. la sortie) accumulateur et e (resp. f) représente les entrées (resp. sorties) correspondant aux éléments de tableaux). Soit un entier s. On définit le composant $Map\_Red(C, s)$ avec :

- pour entrée l'ensemble $I_M R = \{init\} \cup \{It_0, \ldots, It_{s-1}\}$;
- pour sortie l'ensemble $O_M R = \{out\} \cup \{Ot_0, \ldots, Ot_{s-1}\}$;

et est définit inductivement par :

- $Map\_Red(C, 1) = C_{[a_{in} \leftarrow init, e \leftarrow It_0, a_{out} \leftarrow out, f \leftarrow Ot_0]}$;
- $Map\_Red(C, s+1) = Map\_Red(C, s)_{[out \leftarrow acc]} \times C_{[a_{in} \leftarrow acc, e \leftarrow It_s, a_{out} \leftarrow out, f \leftarrow Ot_s]}$.

5.4 Conclusions

Nous avons présenté une notion de contrat assume-guarantee pour des composants réactifs synchrones. Nous montrerons au chapitre 9 les utilisations qui peuvent en être faites dans un contexte de validation formelle.

Cette notion de contrat, répandue depuis de nombreuses années dans le domaine du génie logiciel orienté objet, est assez récente dans le cadre des systèmes réactifs. Elle semble prometteuse, notamment au regard des études de cas que nous présentons dans les chapitres 6 et 7.

Le projet *Alidecs* (*Langages et Atelier Intégré pour le Développement de Composants Embarqués Sûrs*) démarre en septembre 2004 pour une durée de 2 ans, dans le cadre de l'*ACI*[2] *sécurité*). Plusieurs laboratoires (équipe Sémantique, Preuve et Implémentation du *LIP-6*, équipe *Synchrone* de Verimag, projets INRIA *Pop-Art* et *Mimosa*, équipe *CMOS* du LaMI) tentent de mettre en commun leurs efforts afin d'« étudier les systèmes embarqués critiques de grande taille, pour lesquels la réutilisation devient un problème crucial ». Le projet privilégie une approche *langage* des aspects suivants :

- support au cycle de vie, depuis les phases initiales de spécification jusqu'au code embarqué efficace ;
- mécanismes de composition modulaires utilisables aussi bien dans les programmes, dans leurs spécifications, dans la description des environnements ;
- validation précoce des assemblages de composants ;
- validation précoce du comportement dynamique des systèmes par simulation des programmes et des spécifications, tout au long des phases de développement ; la possibilité d'exécuter des programmes incomplets est un des objectifs prioritaires[3].

Les contrats tels que nous les introduisons dans cette thèse devraient aider à répondre à plusieurs de ces critères.

Les contrats seront aussi étudiés dans le cadre du projet européen *Assert* (*(Automated proof based System and Software Engineering for Real-Time*[4]) qui démarre lui aussi en septembre 2004 pour une durée de 3 ans. Ce projet regroupe plusieurs industriels de l'avionique et du transport (l'Agence Spatiale Européenne, Alcatel Space, EADS, Dassault, Esterel Technologies, Prover,...) ainsi que plusieurs laboratoires universitaires européens (ETH Zurich, INRIA, CNRS-LAAS, Verimag,...), et a comme objectif global de mutualiser les efforts des participants en terme de spécification correcte des

[2] action conjointe du CNRS, de l'INRIA et du Ministère de la Recherche.

[3] Cette description est directement inspirée de la page de présentation du projet /http://www-verimag.imag.fr/SYNCHRONE/alidecs/index.php.

[4] http://www.assert-online.net/

systèmes embarqués. Il vise à développer en milieu industriel l'utilisation de méthodes de développement (spécification et validation) éprouvées en milieu académique, notamment au sein d'un processus de développement logiciel commun standard. En ce qui concerne les contrats, l'accent devrait être avant tout mis sur l'utilisation des contrats dans le processus de *vérification* ainsi que sur des aspects d'*exécution précoce*, que nous évoquons dans les perspectives de ce document.

Partie III

Exemples

Chapitre 6

Gyroscope : structure redondante tolérante aux pannes

Dans ce chapitre, nous nous intéressons à une première étude de cas qui concerne la spécification d'un système tolérant aux pannes de traitement de données gyroscopiques à bord d'un avion. Ce programme reçoit des informations de la part de quatre gyroscopes (désignant chacun la position de l'avion) et synthétise une information globale qui sera utilisée par le système de commande de vol. Nous montrons comment les itérations et les contrats peuvent être utilisés pour décrire ce système. Nous insisterons aussi sur l'importance d'une démarche progressive dans la conception d'un système de moyenne ou grande taille.

Après une rapide introduction, nous présentons informellement le fonctionnement du système étudié au paragraphe 6.2. Nous appliquons ensuite une méthodologie de spécification top-down *des différentes parties de l'application. Nous présentons tout d'abord leur interface au paragraphe 6.3. Nous spécifions ensuite les contrats des composants principaux (section 6.4) avant d'en donner une implémentation à la section 6.5. Enfin, nous présentons au paragraphe 6.6 une propriété globale du système que nous étudierons plus spécialement au chapitre 10.*

La version présentée ici est une version finale qui a été obtenue après évolution de plusieurs versions. A chaque version nous avons pris en compte une nouvelle caractéristique du système. Cette démarche, sur laquelle nous reviendrons au chapitre 10, montre l'utilité des contrats à la fois pour la spécification et pour la validation.

6.1 Introduction

L'application qui nous intéresse est un système tolérant aux pannes de traitement de données gyroscopiques utilisé dans un avion. Le but est de montrer l'utilité

- des itérations pour exprimer la structure régulière du programme ;
- des contrats pour la description des composants constituant le système tôt dans le cycle de développement.

6.2 Description informelle du gyroscope

On considère un système informatique embarqué dans un avion qui a pour tâche de traiter les données gyroscopiques de l'appareil. Ce système reçoit des informations de 4 dispositifs physiques identiques (des gyroscopes) chacun fournissant des informations de variations de positions de l'appareil selon trois axes (voir figure 6.1) :

- *Roll*, correspondant au *roulis* (mouvement d'oscillation autour de l'axe du fuselage, c'est-à-dire de la longueur de l'appareil) ;
- *Yaw*, correspondant au *lacet* (mouvement d'oscillation autour d'un axe vertical passant par le centre de gravité de l'appareil) ;
- et *Pitch*, correspondant au *tangage* (mouvement d'oscillation autour de l'axe des ailes, c'est-à-dire de la largeur de l'appareil).

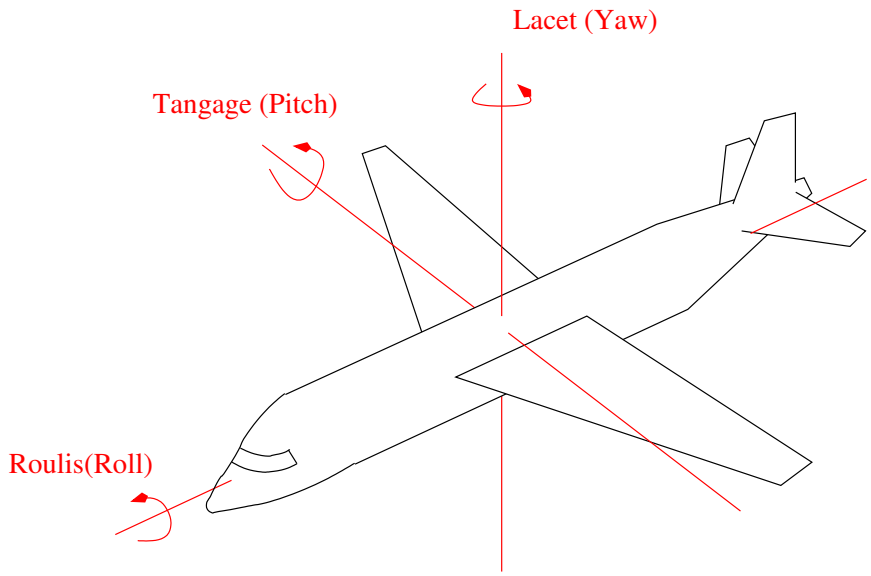


FIG. 6.1 – Un avion et ses 3 axes de mouvement *Roll*, *Pitch* et *Yaw*.

À partir de ces informations dupliquées, le système va calculer une valeur pour chaque axe. Cette valeur pourra être considérée par le reste du système comme la valeur valide renvoyée par les gyroscopes pour chacun des 3 axes (voir figure 6.2).

6.2.1 Structure

Pour chaque axe, un gyroscope fournit 2 valeurs décrivant exactement le même phénomène (comme nous le verrons plus tard, la présence de *2* valeurs est utile pour détecter d'éventuels erreurs de connexion entre les gyroscopes et le système informatique). Le programme FTSGyroscope [1] que nous devons décrire (voir figure 6.3) reçoit donc 3 groupes (un pour chaque axe) de 4 couples de valeurs (l'entrée Axes). À partir de ces informations, il doit calculer 3 valeurs : une pour chaque axe (la sortie Secure_Values).

Chacun de ces trois groupes d'information est traité par une instance du composant EvaluateAxis (présenté à la figure 6.4). Celui-ci reçoit donc en entrée 4 couples de valeurs (appelés channels)

[1] FTS signifie *Fault Tolerant System*, en français *Système Tolérant aux Pannes*

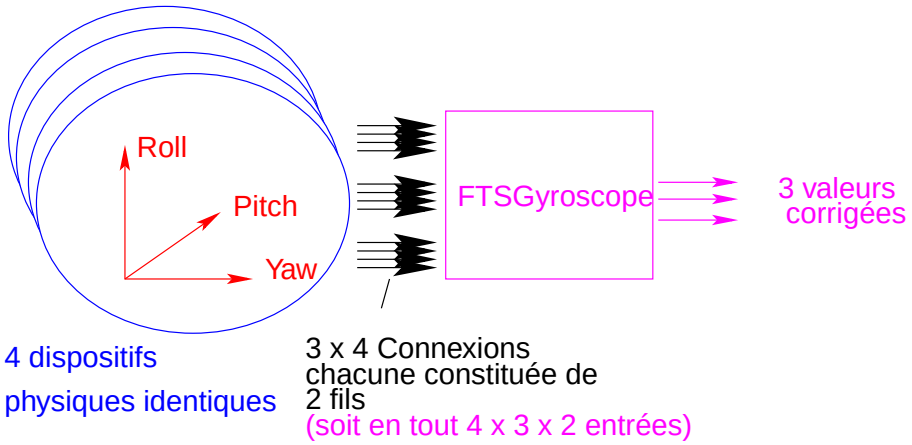


FIG. 6.2 – FTSGyroscope dans son environnement.

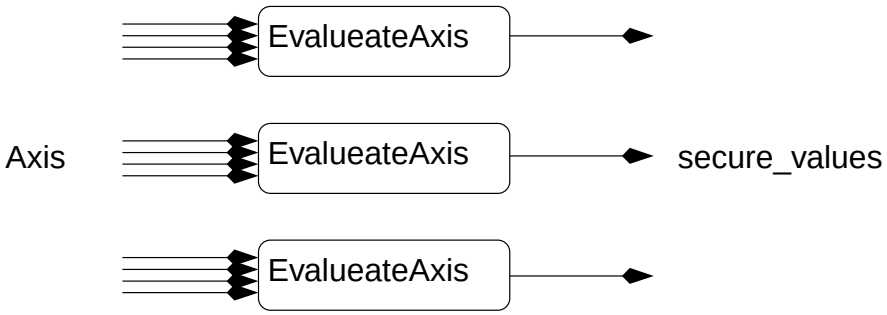


FIG. 6.3 – La structure générale de FTSGyroscope.

et doit fournir en échange la nouvelle valeur pour l'axe correspondant (AxisValue). Notons ici que l'itération du noeud Channel reçoit en paramètres accumulateur les valeurs calculées dans l'instant *précédent* par toutes les instances de Channel. Cette valeur, notée pre(resChannels) est propagée à toutes les instances de Channel qui ne la modifient pas. Sur le dessin, ce rebouclage a été symbolisé par des points sur les fils resChannels concernés.

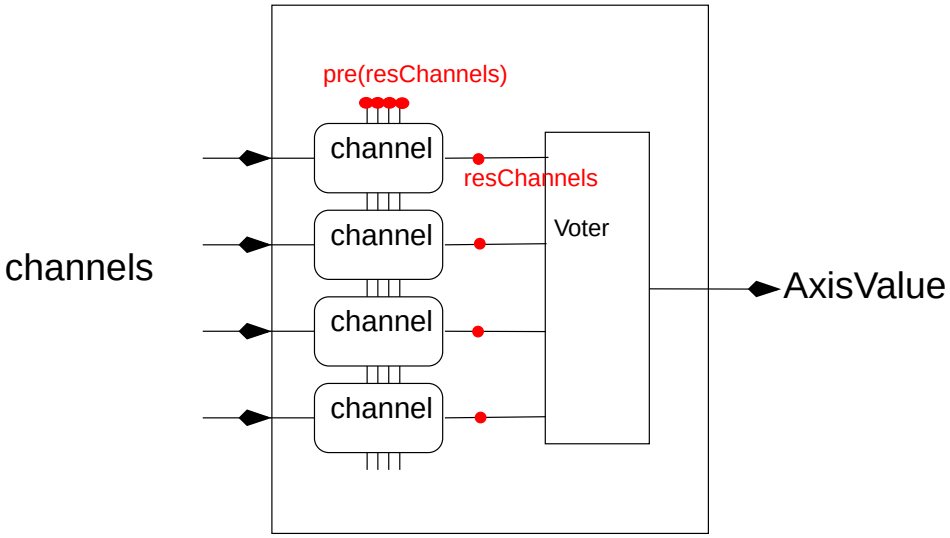


FIG. 6.4 – Un Axe.

Chacun des 4 couples reçus par EvaluateAxis est traité de la même manière avec le nœud Channel (figure 6.5). Ce nœud reçoit un couple inChannel et calcule 2 informations (la sortie outChannel) : une valeur correspondant à une correction de ses deux entrées et un booléen indiquant la validité de la valeur qu'il calcule (indiquant donc si le reste du système peut se servir de cette valeur ou pas). Chaque instance de Channel reçoit de plus les valeurs calculées par les quatre canaux de l'axe à l'instant précédent (notée previousOutChannel) afin de déterminer des erreurs « *croisées* » (voir en 6.2.2). Le composant Channel utilise deux composants (Maintain et CrossFailDetect) pour détecter

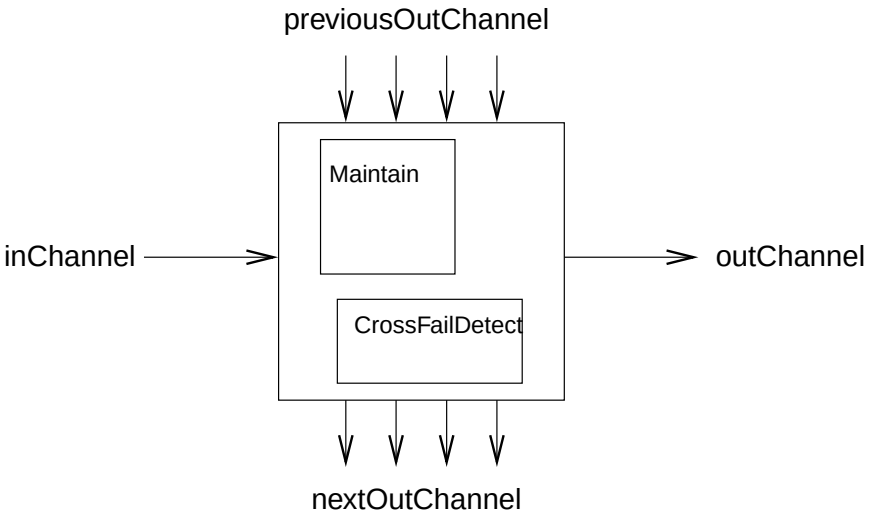


FIG. 6.5 – Un Channel.

d'éventuelles pannes. Nous les présenterons en 6.2.2.

Les valeurs calculées par les 4 channels d'un axe sont transmises à un *voteur* (appelé simplement Voter) qui, selon les pannes détectées calcule la variation d'angle de la position de l'avion fournie par les gyroscopes pour l'axe concerné. Dans la version originale du programme, le mode de calcul de la valeur finale dépendait du nombre de pannes détectées (moyenne olympique si aucune panne, médiane si seulement un channel non-valide, etc.). Pour simplifier la présentation ici, nous avons décidé de calculer le vote avec une simple moyenne des valeurs valides.

Ce système a été une première fois décrit en LUSTRE [HMM+03], sans utiliser ni les contrats ni les tableaux. En ce qui concerne les tableaux, le lecteur peut déjà se convaincre de la régularité intrinsèque du système à deux niveaux :

- au niveau des axes ;
- au niveau des channels ;

puisque dans les deux cas les calculs appliqués sont identiques pour chaque axes ou channel. La description que nous donnons plus loin est basée sur l'utilisation des contrats et des tableaux.

6.2.2 Les fautes

Le système que nous étudions ici est *tolérant aux fautes* (ou *pannes*), c'est-à-dire qu'il est conçu pour pouvoir réagir à certaines évolutions *incorrectes* de son environnement. Par exemple, le système doit pouvoir fonctionner normalement si un des 4 gyroscopes tombe en panne.

Il existe deux catégories de fautes :

- Les fautes des *capteurs* liées au dysfonctionnement des gyroscopes eux-mêmes ;
- Les fautes de *liaison*qui sont dues à des connexions défectueuses entre le système physique (les gyroscopes) et l'ordinateur implémentant le système de traitement.

Le nœud Maintain dont nous avons parlé plus haut permet de détecter les *fautes de liaison* : il compare les 2 valeurs reçues par le Channel et émet une erreur si ces 2 valeurs diffèrent trop l'une de l'autre *pendant trop longtemps*. Il existe donc une sorte de *délai de propagation* des erreurs de capteurs. Si une erreur ne dure pas plus longtemps qu'une certaine borne fixée en constante dans le programme, alors on considère que l'erreur n'est pas significative.

Les pannes de capteurs sont détectées en comparant les valeurs venant des différents channels associés à chaque axe. Le nœud CrossFailDetect détecte ce genre d'erreurs. Dans la spécification originale, il existe aussi une notion de délai de propagation des erreurs de liaisons.

Ces deux fautes sont émises par chaque channel à l'attention du reste du système pour signaler qu'il ne faut pas prendre en compte la valeur qu'il renvoie.

6.3 Interface des composants

Nous commençons par présenter les différents types de données manipulées par le programme. Ensuite, nous proposons de suivre un principe de développement *top-down* en décrivant dans l'ordre :

- les interfaces des composants constituant le système (FTSGyroscope, OneAxis, Channel et Voter) ;
- les contrats associés ;
- une implémentation possible des composants.

La structure régulière mise en évidence plus haut est codée par des programmes avec itérations. Des itérations sont également utilisées dans les contrats pour exprimer des propriétés sur les paramètres tableaux des composants.

6.3.1 Types des données

Les données fournies par les gyroscopes sont représentées par une variable de type Faulty_Array : un tableau à 2 dimensions contenant les informations des **3** axes et, pour chaque axe, de **4** channels :

```
type Faulty_Array = Faulty_ChannelT^4^3;
```

Le type Faulty_ChannelT est utilisé pour les entrées du composant Channel et représente un couple de valeurs reçues d'*un* gyroscope pour *un* channel :

```
type Faulty_ChannelT={valuea : int,valueb : int};
```

Comme nous l'avons vu plus haut, chaque channel rend un couple constitué d'un booléen dénotant une potentielle erreur local_failure et de la valeur calculée pour le channel local_value :

```
type Valid_ChannelT = {local_failure : bool,local_val : int};
```

6.3.2 Le système entier

Le composant principal FTSGyroscope reçoit en entrée un tableau de 3 axes (une variable de type Faulty_Array). En sortie, il fournit 3 réels secure_values représentant les valeurs des 3 axes :

```
node FTSGyroscope(Axis : Faulty_Array) returns (secure_values : int^3);
```

Notons que :

- axis[0] contient les informations venant des gyroscopes pour l'axe *roll* (*roulis*), secure_values[0] contient la valeur calculée par le système pour l'axe *roll* ;
- axis[1] contient les informations venant des gyroscopes pour l'axe *pitch* (*tangage*), secure_values[1] contient la valeur calculée par le système pour l'axe *pitch* ;
- axis[2] contient les informations venant des gyroscopes pour l'axe *yaw* (*lacet*), secure_values[2] contient la valeur calculée par le système pour l'axe *yaw*.

6.3.3 Un axe

Chaque axe est traité par une instance du nœud EvaluateAxis qui a le profil suivant :

```
node EvaluateAxis(channels : Faulty_ChannelT^4; delta : int) returns (AxisValue : int);
```

Il reçoit en entrée un tableau channels de 4 Faulty_Channels ainsi qu'une valeur delta (différente pour chaque axe) et rend en sortie la nouvelle valeur pour l'axe AxisValue. L'entrée delta sera utilisée par les Channels pour comparer les deux valeurs reçues et détecter une éventuelle erreur de liaison.

6.3.4 Un channel

Un Channel reçoit 4 entrées. La première est un tableau contenant les valeurs calculées pour tous les channels, *à l'instant précédent*. Ces valeurs permettent au channel de détecter d'éventuelles erreurs croisées (à l'aide du nœud CrossFailDetect).

La seconde entrée nbInChannel indique le numéro du channel qu'on est en train de traiter et est nécessaire lorsqu'on détermine les erreurs croisées, afin de ne pas considérer la valeur calculée par l'instance courante du nœud Channel. L'entrée inChannel correspondant à un élément du tableau

channels reçu directement depuis les entrées du nœud EvaluateAxis. Enfin, l'entrée delta représente la différence maximum autorisée entre les deux valeurs que reçoit le channel et lui permet de signaler une faute de liaison. Cette valeur est fixée pour chaque axe par une constante (DELTA_ROW, DELTA_PITCH et DELTA_YAW).

Le nœud Channel fournit 2 sorties : la première (nextOutChannel) est une copie à l'identique du tableau previousOutChannel. La seconde est un couple de type Valid_ChannelT qui contient les informations calculées par le channel qui seront ensuite utilisées par le voteur. Cette variable outChannel est aussi *rebouclée* et utilisée avec un temps de retard par les différentes instances du nœud Channel par l'intermédiaire de l'entrée previousOutChannel.

```
node Channel(previousOutChannel : Valid_ChannelT^4;
             nbInChannel : int;
             inChannel : Faulty_ChannelT;
             delta : Faulty_ChannelT;)
returns (nextOutChannel : Valid_ChannelT^4;
         outChannel : Valid_ChannelT);
```

6.3.5 Le voteur

Le voteur reçoit les couples de type Valid_ChannelT calculées par les 4 occurrences de Channel et retourne une seule valeur vote.

```
node Voter(channels : Valid_ChannelT^4) returns (vote : int);
```

6.4 Spécification par contrats

L'étape suivante dans la description du système consiste à décrire les contrats des composants qu'on utilise. Comme nous l'avons proposé au chapitre 5.3.7, nous allons associer à chaque nœud du programme un couple d'observateurs : l'un représentant l'assertion du contrat, l'autre représentant la garantie. Les propriétés que nous allons encoder dans ces contrats sont toutes tirées de la spécification informelle dont nous disposions au début de l'étude de cette application.

6.4.1 Extension des interfaces

Les propriétés que nous allons donner pour décrire les comportements des composants (leur contrats) nécessitent l'introduction de nouveaux paramètres. Ces paramètres sont rajoutés à l'interface du nœud lui-même, mais ne sont pas utilisés dans son corps. Il ne sont utilisés que dans les observateurs pour l'assertion ou la garantie.

La propriété typique d'un système de calcul de moyenne de plusieurs informations physiques est de la forme suivante : si les valeurs en entrées ne sont pas trop fausses (c'est-à-dire pas trop loin de la vrai valeur physique), alors la valeur en sortie ne l'est pas non plus. Pour exprimer cela, la spécification du système nécessite, au niveau global, des constantes qui expriment des valeurs idéales pour chacun des 3 axes. On veut notamment pouvoir exprimer la propriété selon laquelle le système ne produit pas des valeurs complètement incohérentes par rapport à ces valeurs idéales. Cette notion d'« idéal » est donnée aux travers de valeurs constantes fixées pour toute l'exécution. *Ces valeurs ne doivent pas être*

utilisées dans le calcul des sorties. Elles servent seulement à s'assurer que les valeurs calculées par notre système sont cohérentes.

Nous représentons la valeur idéale d'un axe par un nouveau paramètre du nœud EvaluateAxis qui s'appelle sobrement ideal. On représente la distance minimale entre la valeur calculée et la valeur idéale par la variable delta_to_ideal. La figure 6.6 illustre l'extension de l'interface du nœud. Rappelons qu'il ne sera pas possible d'utiliser ces variables dans le corps du composant, mais seulement dans son contrat. Au niveau le plus haut, ces valeurs sont déterminées par des constantes : IDEAL_GOD et DELTA_TO_IDEAL_GOD, IDEAL_PITCH et DELTA_TO_IDEAL_PITCH, IDEAL_YAW et DELTA_TO_IDEAL_YAW.

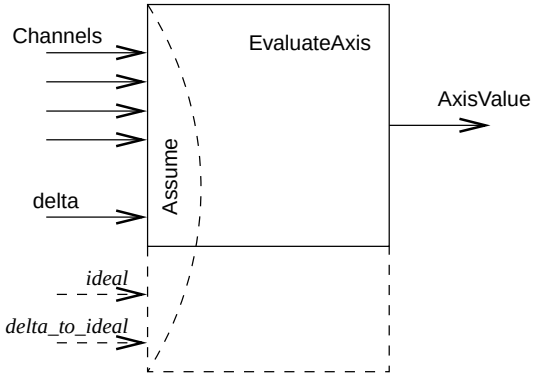


FIG. 6.6 – Ajout de paramètres pour l'expression des contrats.

6.4.2 Un axe

À partir des informations fournies par 4 gyroscopes pour un axe, EvaluateAxis calcule effectivement la valeur unique correspondant à cet axe.

Assertion – EvaluateAxis suppose qu'au plus 3 des couples de valeurs qu'il reçoit sont trop loin de la valeur idéale pour l'axe correspondant.

On considère qu'un couple est trop loin de la valeur idéale lorsqu'*au moins une des deux valeurs est trop loin de la valeur idéale*. On va commencer par décrire à l'aide d'une itération le comptage des channels non valide. Cette itération utilise le nœud NbTooFar représenté à la figure 6.7.

Ce nœud prend en entrée le nombre de channels qui ont déjà été évalués dans les étages précédents de l'itération comme étant « *trop éloignés* ». Il reçoit aussi un channel, ainsi que les constantes ideal et delta_to_ideal. On l'utilise ensuite dans l'itération qui calcule le nombre nbTooFar de valeurs trop éloignées parmi *tous* les channels associés à un axe. L'assertion est valide tant que ce nombre est inférieur à 3 :

```
node assumeEvaluateAxis(channels : Faulty_ChannelIT^4;
                        delta : int;
                        ideal : int;
                        delta_to_ideal : int)
returns (assumeOK : bool);
var nbTooFar : int;
let
   nbTooFar = red<<NbTooFar;4>>(0,channels,
                                ideal^4,delta_to_ideal^4);
   assumeOK = nbTooFar <= 3;
tel
```

Garantie – Lorsque placé dans un environnement satisfaisant l'assertion que nous venons de décrire, EvaluateAxis garantit que la valeur qu'il calcule n'est pas trop éloignée de la valeur ideal correspondante. Cette propriété est simplement décrite par le nœud suivant :

```
node guaranteeEvaluateAxis(channels : Faulty_ChannelIT^4;
                           delta : int;
                           ideal : int;
                           delta_to_ideal : int;
                           AxisValue : int)
returns (guaranteeOK : bool);
let
   guaranteeOK = abs((AxisValue - ideal)) < delta_to_ideal;
tel
```

6.4.3 Un channel

Pour chaque axe, il y a 4 instances du nœud Channel. L'assertion du nœud Channel porte sur les valeurs calculées par les 3 autres instances à l'instant précédent. Sa garantie exprime une relation entre la valeur que le nœud calcule et les valeurs qu'il reçoit des gyroscopes.

Assertion – Le nœud Channel suppose que ses 3 collègues ne sont pas tous tombés en panne. On utilise pour cela le nœud NbOtherFailures qui calcule le nombre de pannes parmi les channels voisins. Par ailleurs, il suppose que les valeurs calculées par ses collègues qui ne sont pas tombés en panne sont toutes proches de la valeur idéale pour l'axe traité. Pour cela, il fait appel à un nœud NotFarFromIdeal. Nous ne détaillerons pas plus les nœuds NbOtherFailures et NotFarFromIdeal. Le nœud assumeChannel est donné ci-dessous :

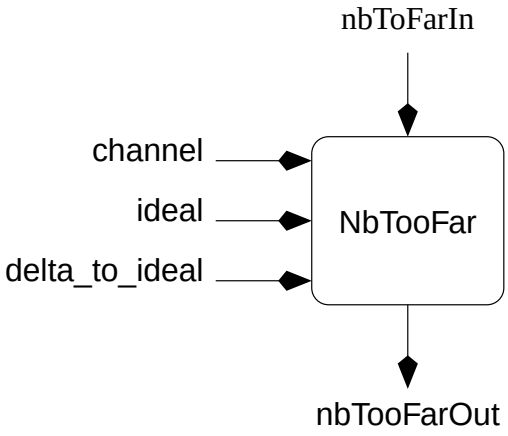


FIG. 6.7 – Le nœud NbTooFar.

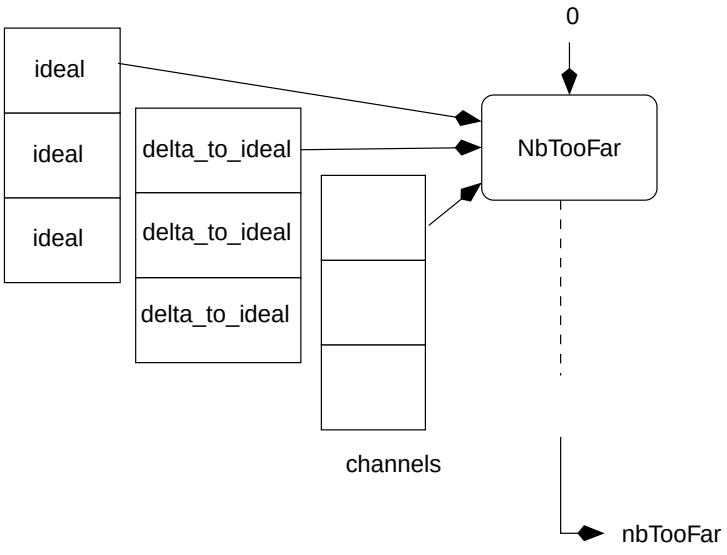


FIG. 6.8 – L'itération utilisée dans l'assertion du composant EvaluateAxis.

```
node assumeChannel(previousOutChannel : Valid_ChannelT^4;
                   nbInChannel : int;
                   inChannel : Faulty_ChannelT;
                   delta : int;
                   ideal : int;
                   delta_to_ideal : int)
returns (assumeOK : bool);
var unfailedChannels_notFarFromIdeal : bool;
let
   assumeOK = NbOtherFailures(previousOutChannel,
                              nbInChannel)<3
                              and unfailedChannels_notFarFromIdeal;
   unfailedChannels_notFarFromIdeal = NotFarFromIdeal(previousOutChannel,
                                                      nbInChannel,
                                                      ideal,
                                                      delta_to_ideal);
tel
```

Garantie – Lorsque placé dans un environnement satisfaisant l'assertion que nous venons de décrire, le nœud Channel garantit au choix une des deux propriétés suivantes :

- Il émet une erreur locale. C'est le cas si les valeurs qu'il reçoit en entrée sont trop différentes l'une de l'autre pendant trop longtemps ;
- Il n'émet pas d'erreur et la valeur qu'il calcule (outChannel.local_value) n'est pas trop éloignée des valeurs qu'il a reçues en entrée (inChannel.valuea et inChannel.valueb).

Le nœud Maintain évoqué plus haut sert à réaliser la détection des erreurs locales. Comme nous l'avons vu plus haut, ce composant permet de ne propager une erreur que si celle-ci perdure un temps jugé significatif.

En attendant que l'erreur soit confirmée, il n'est néanmoins pas question que le Channel utilise les valeurs erronées qu'il reçoit pour calculer sa valeur de sortie, même si l'erreur n'est pas propagée. Si une erreur *instantanée* est détectée, et jusqu'à ce que cette valeur soit considérée comme importante au travers du nœud Maintain, on va forcer le channel à utiliser les dernières valeurs *valides* qu'il a reçues en entrée.

Il faut prendre cette notion de propagation temporelle de l'erreur détectée en compte dans le contrat du Channel. En effet, celui-ci ne garantit pas que la valeur qu'il rend en sortie n'est pas trop loin des valeurs qu'il vient de recevoir en entrée, mais qu'elle n'est pas trop loin des dernières valeurs valides qu'il a reçues en entrée. Au final, Channel garantit que :

- Soit il émet une erreur locale (c'est-à-dire que les valeurs qu'il a reçues en entrée ont été trop éloignée pendant trop longtemps). Cette propriété s'écrit simplement outChannel.local_failure ;
- Soit la valeur qu'il émet en sortie n'est pas trop éloignée des dernières valeurs valides reçues en entrée. Ces valeurs sont maintenues à jour dans la variable lastValidChannel grâce au nœud lastValid. Dans le cas où aucune erreur instantanée n'est détectée, lastValidChannel vaut la valeur courante de inChannel. Au final, on doit comparer à chaque instant la valeur courante de sortie avec les valeurs mémorisées dans lastValidChannel. On a donc :

```
abs((lastValidChannel.valuea - outChannel.local_value)) <= delta
abs((lastValidChannel.valueb - outChannel.local_value)) <= delta
```

Au démarrage du système, la mise en route de la détection des erreurs prend du temps : le nœud Maintain ne signalera pas d'erreurs avant le TIME-ième instant. En attendant, on décide de reporter les erreurs instantanées. Dans le contrat du channel, on distingue donc deux comportements : le

premier, mis en place tant que le nœud Maintain n'a pas atteint sa stabilité de fonctionnement (c'est-à-dire durant les TIME premiers instants), consiste à reporter les erreurs instantanées. On implémente cela avec la variable initProp. Ensuite, on va reporter les erreurs 'temporelles' avec la variable normalProp. La partition temporelle de la garantie finale est réalisée à l'aide du nœud _during_then_ qui renvoie la valeur de son premier paramètre pendant les TIME premiers instants et de son troisième paramètre ensuite. L'observateur guaranteeChannel exprime cette propriété :

```
node guaranteeChannel(previousOutChannel : Valid_ChannelT^4;
                      nbInChannel : int; inChannel : Faulty_ChannelT;
                      delta : int;ideal : int;delta_to_ideal : int;
                      nextOutChannel : Valid_ChannelT^4;
                      outChannel : Valid_ChannelT)
returns (guaranteeOK : bool);
var instantFailure : bool;
    lastValidChannel : Faulty_ChannelT;
    initProp, normalProp : bool;
let
  normalProp = outChannel.local_failure
               or ((abs((lastValidChannel.valuea - outChannel.local_value)) <= delta)
                   and (abs((lastValidChannel.valueb - outChannel.local_value)) <= delta));
  initProp = outChannel.local_failure
             or ((abs((inChannel.valuea - outChannel.local_value)) <= delta)
             and (abs((inChannel.valueb - outChannel.local_value)) <= delta));
  lastValidChannel = lastValid(inChannel,delta);
  guaranteeOK = _during_then_(initProp, TIME, normalProp);
tel
```

6.4.4 Le voteur

Assertion – L'assertion du voteur est décomposée en 2 propriétés :

- Elle suppose tout d'abord qu'il y a au plus 3 channels qui se déclarent en erreur (propriété décrite à l'aide du nœud totalNbFailures qui compte le nombre de channels se déclarant en erreur) ;
- Ensuite, parmi les channels qui n'émettent pas d'erreur, la valeur qu'ils fournissent n'est pas trop éloignée de la valeur ideal (propriété représentée par la variable locale notFailedYieldsValueNearIdeal).

```
node assumeVoter(channels : Valid_ChannelT^4;
                 ideal : int;
                 delta_to_ideal : int)
returns (assumeOK : bool);
var notFailedYieldsValueNearIdeal : bool;
let
  assumeOK = totalNbFailures(channels)<=3
             and notFailedYieldsValueNearIdeal;
  notFailedYieldsValueNearIdeal =
          checkValidChannels(channels, ideal, delta_to_ideal);
tel
```

Garantie – Sous cette hypothèse, le voteur garantit que la valeur qu'il calcule n'est pas trop éloignée de la valeur idéale. Cette propriété est donnée dans le nœud guaranteeVoter suivant :

```
node guaranteeVoter(channels : Valid_ChannelT^4;
                    ideal : int;
                    delta_to_ideal : int;
                    vote : int)
returns (guaranteeOK : bool);
let
   guaranteeOK = abs(vote-ideal)<delta_to_ideal;
tel
```

6.5 Implémentation

6.5.1 Le système entier

Le système gyroscopique est décrit à l'aide du nœud FTSGyroscope ci-dessous.

```
node FTSGyroscope(axis : Faulty_Array) returns (secure_values : int^3)
let
   secure_values = map<<EvaluateAxis;3>>(axis,
           [DELTA_ROLL, DELTA_PITCH, DELTA_YAW],
           [IDEAL_ROLL,IDEAL_PITCH,IDEAL_YAW]
           [DELTA_TO_IDEAL_ROLL,DELTA_TO_IDEAL_PITCH,DELTA_TO_IDEAL_YAW]);
tel
```

La sortie secure_values est simplement calculée en « *mappant* » le composant EvaluateAxis sur le tableau axis.

6.5.2 Un axe

L'implémentation du composant EvaluateAxis nécessite tout d'abord la variable locale resChannels qui est le tableau calculé par les Channels, ensuite fourni au Voter.

Comme nous l'avons expliqué plus haut, resChannels est communiqué (avec un instant de retard) aux nœuds Channel afin de permettre la détection d'erreur de capteurs. L'accumulateur de l'itération (initialisé avec la variable mapredInit) permet de propager cette valeur à toutes les instances du nœud Channel. Cette valeur n'est pas disponible à l'initialisation du système (puisque l'itération de Channel n'a pas encore produit de valeur pour resChannels). Ainsi, à l'initialisation, mapredInit vaut une valeur arbitraire :

```
mapredInit = {false, 0.0}^4,
```

Aucune faute n'a été détectée et on donne une valeur quelconque pour chaque channel. La variable dumbChannel n'est utile que syntaxiquement pour décrire l'itération map_red : elle n'est pas réutilisée.

Pour calculer resChannels, on utilise donc une itération map_red du composant Channel. Le paramètre [0,1,2,3] permet d'identifier à chaque étage le channel que l'on est en train de considérer. On a notamment besoin de cette information pour détecter les erreurs croisées.

```
node EvaluateAxis(channels : Faulty_ChannelT^4;delta : int;
                  ideal : int; delta_to_ideal : int)
returns (AxisValue : int)
var resChannels : Valid_ChannelT^4;
    dumbChannel, initChannels : Valid_ChannelT^4;
    mapredInit : Valid_ChannelT^4;
let
  initChannels = {local_failure = false, local_value = 0.0}^4;
  dumbChannel,resChannels = map_red<<Channel;4>>(mapredInit,[0,1,2,3],
                                                 channels,delta^4,
                                                 ideal^4, delta_to_ideal^4);
  AxisValue = Voter(resChannels, ideal^4, delta_to_ideal^4);
  mapredInit = (initChannels -> pre(resChannels));
tel
```

6.5.3 Un channel

L'implémentation du composant Channel se découpe principalement en 2 parties. D'un coté on détecte d'éventuelles erreurs de liaison (voir 6.2.2) à l'aide du nœud Maintain. De l'autre, on va détecter d'éventuelles erreurs de capteurs à l'aide du nœud CrossFailDetect.

Comme nous l'avons dit lors de la présentation du contrat de Channel, le comportement n'est pas exactement identique selon que le système est dans sa phase d'initialisation (les TIME) premiers instants ou dans la phase de fonctionnement normal. La valeur renvoyée par le channel en dépend aussi. On utilise donc deux définitions différentes. La première, représentée par la variable localChannel_duringInit donne la valeur du channel pendant la phase d'initialisation. On détecte alors les erreurs de capteurs de manière instantanée. On a donc :

```
localChannel_duringINIT ={local_failure = instantFailure;
                          local_value = if instantFailure
                                        then 0
                                        else ((inChannel.valuea + inChannel.valueb) div 2)};
```

La seconde définition du channel, représentée par localChannel_afterInit donne sa valeur dans le reste de l'exécution. Là, on ne propage les erreurs de capteurs que si elle durent assez longtemps (utilisation du nœud Maintain) :

```
localChannel_afterINIT = {local_failure = maintain,
                          local_value = if instantFailure
                                        then (lastValidChannel.valuea
                                              + lastValidChannel.valueb) div 2
                                        else ((inChannel.valuea + inChannel.valueb) div 2)};
```

Le nœud Maintain vérifie simplement que la valeur booléenne qu'il reçoit en second paramètre est vraie pendant au moins TIME instants. On vérifie ainsi que inChannel.valuea et inChannel.valueb ne sont pas trop éloignées l'une de l'autre pendant plus de TIME instants avec l'expression :

```
instantFailure = abs((inChannel.valuea - inChannel.valueb))>delta
maintain = Maintain(TIME, instantFailure);
```

On utilise une variable locale supplémentaire, appelée localChannel qui permet de rassembler les deux définitions que nous venons de donner à l'aide du nœud _during_then qui permet de définir la valeur d'une variable sur deux périodes de temps distinctes : une allant de l'initialisation du système à la borne TIME fournie en paramètre, et l'autre valant pour le reste de l'exécution. localChannel est définie par :

```
localChannel = _during_then_(localChannel_duringINIT, TIME, localChannel_afterINIT);
```

Le channel fourni en sortie est défini comme suit : Le premier champ (outChannel.local_value) est calculé comme une disjonction entre la présence d'une erreur locale (représentée par localChannel.local_failure) et une erreur de liaison dont l'existence est déterminée par le composant CrossFailDetect. On a simplement :

```
local_failure = (localChannel.local_failure
                 or CrossFailDetect(nbInChannel,
                                    localChannel,
                                    previousOutChannel))
```

Le second champ outChannel.local_value contient la valeur calculée pour localChannel.local_value. Le nœud Channel complet est donné à la figure 6.9.

6.5.4 Le voteur

Le voteur calcule simplement une moyenne des valeurs fournies pour les channels valides. On commence par *masquer* les valeurs non valides : si channels[i].local_failure vaut faux, mask[i] vaut 0.0, sinon, il vaut channels[i].local_value. Il ne nous reste plus qu'à calculer la somme des éléments de mask avec l'équation :

```
globalSum = red≪sum;4≫(0.0,mask)
```

et de diviser cette valeur par le nombre de channels qui sont valides, donné par l'équation :

```
nbValid = countValidChannels(channels)
```

Le nœud Voter est donné ci-dessous :

```
node Voter(channels : Valid_ChannelT^4;
           ideal : int;
           delta_to_ideal : int)
returns (vote : int);
var globalSum : int;
    nbValid : int;
    mask : int^4;
let
  nbValid = countValidChannels(channels);
  globalSum = red≪sum;4≫(0.0,mask);
  vote = (globalSum / nbValid);
  mask = map≪masking;4≫(channels);
tel
```

```
node Channel(previousOutChannel : Valid_ChannelT^4;
             nbInChannel : int;
             inChannel : Faulty_ChannelT;
             delta : int;
             ideal : int;
             delta_to_ideal : int)
returns (nextOutChannel : Valid_ChannelT^4; outChannel : Valid_ChannelT)
var localChannel : Valid_ChannelT;
    maintain : bool;
    instantFailure : bool;
    lastValidChannel : Faulty_ChannelT;
    localChannel_afterINIT, localChannel_duringINIT : Valid_ChannelT;
let
  instantFailure = abs(inChannel.valuea - inChannel.valueb)>delta;
  maintain = Maintain(TIME, instantFailure);
  lastValidChannel = lastValid(inChannel, delta);
  localChannel_afterINIT ={local_failure = maintain,
                           local_value = if instantFailure
                                         then (lastValidChannel.valuea
                                               + lastValidChannel.valueb) div 2
                                         else ((inChannel.valuea
                                                + inChannel.valueb) div 2)};
  localChannel_duringINIT ={local_failure = instantFailure;
                            local_value = if instantFailure
                                          then 0
                                          else ((inChannel.valuea
                                                 + inChannel.valueb) div 2)};
  localChannel = _during_then_(localChannel_duringINIT, TIME, localChannel_afterINIT);
  outChannel = {local_failure = localChannel.local_failure
                                or CrossFailDetect(nbInChannel,
                                                   localChannel, previousOutChannel),
                local_value = localChannel.local_value};
  nextOutChannel = previousOutChannel;
tel
```

FIG. 6.9 – Implémentation du nœud Channel.

6.6 Propriété globale du système

Au niveau global, on doit vérifier que le programme satisfait la propriété suivante : *la valeur calculée par le système pour chaque axe ne doit pas être trop loin de la valeur idéale correspondante.* Pour exprimer cette propriété, on utilise un nœud ValueIsSecure défini comme suit :

```
node ValueIsSecure(accu_in : bool;
                   secure_value : int;
                   ideal, delta_to_ideal : int)
returns (is_valid : bool);
let
   is_valid = ((abs((secure_value - ideal)) < delta_to_ideal) and accu_in);
tel
```

On exprime la propriété désirée avec une itération de ce nœud sur le tableau secure_values et les constantes correspondant aux deltas et valeurs idéales associées aux axes :

```
node propOnGyroscope(axis : Faulty_Array) returns (isValid : bool);
var secure_values : intˆ3;
let
   secure_values = FTSGyroscope(axis);
   isValid = red≪ValueIsSecure;3≫(true,
             secure_values,
             [IDEAL_ROLL,IDEAL_PITCH,IDEAL_YAW]);
             [DELTA_TO_IDEAL_ROLL,DELTA_TO_IDEAL_PITCH,DELTA_TO_IDEAL_YAW],
tel
```

Notons que cette propriété n'est pas vérifiable sur ce système, notamment car elle suppose des contraintes sur les entrées que nous n'avons pas exprimées. Nous y reviendrons plus loin, mais il nous faut insister maintenant sur le fait qu'on ne cherche pas à valider des propriétés globales sur de tels systèmes. La partie IV a pour but de proposer des manipulations de base pour *aider* à la compréhension et à la validation d'un système comme celui que nous venons d'étudier.

6.7 Commentaires

Les itérations s'appliquent très bien aux systèmes tolérants aux pannes comme le système gyroscopique. En effet, ces systèmes sont caractérisés par la duplication des informations qu'ils reçoivent en entrée. On duplique les entrées pour s'assurer de pallier les éventuelles erreurs de connexions qui seraient indécelables si on n'avait qu'une version des entrées. Le calcul pour chaque duplication des entrées est en général identique, donc tout à fait exprimable avec un map.

La complexité des composants qui constituent l'application a permis de mettre en évidence l'utilité des contrats lors de la phase de spécification. Les versions des différents composants que nous avons présentées ici sont les versions les plus évoluées que nous avons étudié. Mais leur spécification (notamment celle du Channel) a été progressive : nous avons commencé par développer une version simple sans prise en compte de l'aspect *temporel* de la propagation des erreurs, ni des dépendances croisées entre les différentes instances de channel. Nous avons ensuite étendu cette version de base, en modifiant d'abord le contrat, puis l'implémentation du nœud.

Les contrats facilitent ce travail progressif de spécification. Ils permettent d'expliciter des propriétés sur les composants qui nous aident à raffiner leur comportement. On constate notamment que dans le cas du nœud Channel, on a décrit une partie du comportement à l'aide d'une variable locale lastValidChannel qui était déjà utilisée dans le contrat. Dès la spécification du contrat, on a pu identifier que le composant aurait besoin de cette mémoire pour calculer effectivement ses sorties. L'effort de description du composant a été ainsi décomposé et on a pu *réutiliser* des informations données dans le contrat pour écrire l'implémentation.

Les contrats que nous avons proposés pour les différents composants du système sont relativement complexes : cet exemple met en évidence la nécessité d'un langage expressif pour la description des langages. Il faut en effet un langage au moins aussi expressif que celui utilisé pour décrire les *corps* des composants. Dans notre cas, les contrats sont décrits en LUSTRE sans aucune contrainte.

Enfin, nous avons vu que pour exprimer certaines propriétés, nous avons dû ajouter des paramètres aux composants (les variables ideal ou delta_to_ideal). Cette idée est très importante, car elle nous permet, sans nouvelle construction au niveau langage, de coder des propriétés quantifiées universellement sur des paramètres exterieurs au programme concerné. Par exemple, on a pu écrire une propriété de la forme : $\forall ideal.\forall delta\_to\_ideal.$ si $|x - ideal| < delta\_to\_ideal$ alors $|y - ideal| < delta\_to\_ideal$ où x et y étaient des variables du programme. Cette forme de propriété est triviale à écrire, puisqu'elle découle de l'interprétation que font les outils de preuves des entrées d'un noeud.

Chapitre 7

ELMU : structure régulière et bibliothèque de composants

Ce chapitre présente une seconde étude de cas, tirée elle aussi du domaine de l'avionique. Il s'agit d'un programme de répartition de charges électriques dans un avion : la charge électrique est fournie par quatre générateurs et la répartition de la consommation électrique pour l'habitacle est assurée par un programme informatique. Nous montrons comment les itérations permettent de décrire la structure régulière de ce système et comment les contrats sont utilisables pour décrire les composants de base.
Après avoir présenté l'intérêt de la spécification par contrat pour cette application, nous suivons un cheminement de spécification top-down *comme pour l'application gyroscopique. Nous commencerons par une présentation informelle de l'application au paragraphe 7.3, avant d'en présenter la structure au paragraphe 7.4. Nous donnerons l'implémentation des principaux composants au paragraphe 7.5 et présenterons une propriété globale du système en 7.6. Nous reviendrons sur cette propriété globale ainsi que sur des propriétés portant sur les composants les plus importants de l'application au chapitre 10.*

7.1 Introduction

L'étude de cas que nous présentons concerne le développement d'un logiciel de gestion de la répartition des charges électriques au sein d'un avion (appelé *ELMU* pour *Electric Load Managment Unit*. Elle nous a été proposée par la société Airbus et elle a fait l'objet d'une étroite collaboration entre l'équipe responsable du projet à Airbus et l'équipe Synchrone de Verimag.

La version de l'application que nous présentons ici résulte d'une restructuration totale de l'application pour en faire apparaître les régularités intrinsèques et y appliquer un mode de programmation à base d'itérations. De part sa régularité, cette application a consisté une des grandes motivations pour l'introduction des itérateurs dans le langage LUSTRE.

7.2 Spécifications par contrats sur composants de base

Contrairement au gyroscope, nous n'avions pas de spécification suffisamment précise du ELMU pour construire des contrats intéressants des composants principaux qui constituent l'application. Par

contre, l'application utilise intensivement un grand nombre de petits composants. La *réutilisabilité* d'un composant est alors un argument fortement en faveur de l'utilisation des contrats.

Nous avons décrit au chapitre 5 un composant nommé rMFF (la « *bascule mono-stable redéclenchable* »). Ce composant est en fait beaucoup utilisé dans le ELMU. Une spécification informelle en est donnée dans laquelle on identifie clairement une condition sur l'entrée E et une conséquence sur la sortie S.

Une bibliothèque de composants est donnée avec l'application. Ceux-ci peuvent être :

- combinatoires : détermination du maximum de 3 (ou de 7) valeurs entières, calcul de la moyenne de 2 entiers ;
- ou temporels : bascule mono-stable non redéclenchable, confirmation d'un signal booléen (dans le cas où il est vrai plus d'un certain nombres de cycles).

Pour chacun de ces composants de base, on donne aussi, dans la documentation, une « *Recommandation d'utilisation* » qui est exprimée en français. Ces recommandations sont des contraintes que doivent satisfaire les entrées qu'on fournit au composant si l'on veut que le composant fonctionne « correctement ». Elles correspondent tout à fait à ce qu'on peut exprimer avec les assertions de nos contrats.

7.3 Présentation de l'application

L'application qui nous intéresse gère la répartition des charges électriques dans un avion. L'électricité est produite par des *générateurs* et est consommée par des *charges* (qui peuvent regrouper plusieurs appareils électriques tels que des groupes de lumières, un ensemble de ventilation, etc.).

Le rôle du système qu'on étudie est de décider de l'affectation des charges aux générateurs : à tout moment, on veut définir quel générateur doit produire l'électricité consommée par une charge donnée. Cette activité nécessite des opérations de délestage ou de lestage des générateurs suivant les besoins en électricité des charges, d'éventuelles surcharges, voire une panne d'un ou de plusieurs générateurs.

Typiquement, si un générateur tombe en panne, les charges associées qui sont jugées comme *critiques* ne doivent pas manquer d'électricité : il faut décider quel générateur va remplacer le générateur défectueux et re-répartir les charges sur les générateurs. Même sans panne, on doit veiller à la bonne répartition des charges sur les générateurs afin d'éviter toute surcharge.

Entre les générateurs et les charges se trouvent des *barres* de connexion qui permettent d'organiser les connexions (voir la figure 7.1). Une charge est toujours associée à une même barre, mais un générateur peut être associé à différentes barres au court du temps (potentiellement plusieurs en même temps). Dans la suite de la présentation, nous nous abstrayons de la notion de barre et nous intéressons uniquement à la correspondance entre générateurs et charges.

Généricité – Il est intéressant de mentionner que le nombre de charges et de barres n'est pas fixé de manière définitive. Il se peut très bien qu'en développant le logiciel on s'aperçoive qu'il est nécessaire d'avoir un générateur de plus (ou qu'on se rende compte de la nécessité de certaines charges supplémentaires).

Cette généricité ainsi que la symétrie du traitement des charges et des générateurs sont tout à fait en faveur de l'utilisation des itérateurs. Les informations sur les charges et les générateurs étant représentées par des tableaux, il suffit d'augmenter la taille de ces tableaux pour prendre en compte des nouveaux éléments. Nous reviendrons au chapitre 10 sur les avantages des itérations pour la validation de cette application.

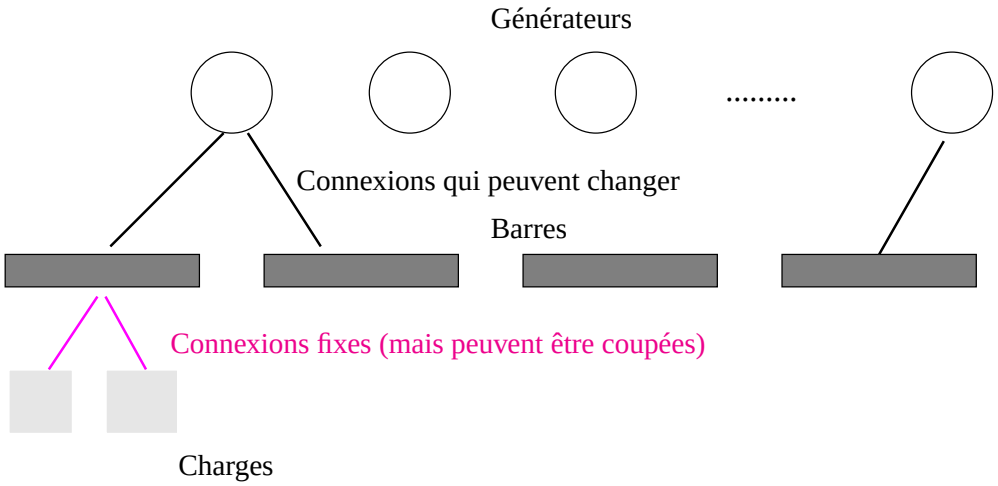


FIG. 7.1 – Connexions entre générateurs, barres et charges.

7.4 Structure

La structure que nous présentons maintenant a été proposée par Pascal Raymond à Verimag, à partir des spécifications fournies par la société Airbus. Nous avons adapté une partie de cette structure en ajoutant des informations utiles pour l'expression de contrats simples sur certains des composants.

7.4.1 Structure générale

Les figures 7.2 et 7.3 montrent respectivement le traitement des charges et le traitement des générateurs. Le système reçoit une entrée de son environnement nommée EntreeGlob de type T_EntreeGlob. Cette variable regroupe 6 types d'informations :

- Un tableau de correspondances entre charges et générateurs, noté EntreeGlob.chg2gen. Le i-ème élément de ce tableau donne le numéro du générateur sur lequel la charge i est connectée ;
- Un tableau EntreeGlob.mesure_chgs contenant les mesures de chaque charge (représentant la puissance électrique consommée par une charge) ;
- Un tableau EntreeGlob.priorite_chgs contenant les niveaux de priorité de chaque charge (permettant de savoir si une charge est cruciale pour le fonctionnement du système et ne doit pas être débranchée) ;
- Un tableau EntreeGlob.mesure_gens contenant les mesures de chaque générateur (représentant la puissance électrique produite par un générateur) ;
- Un tableau de booléens EntreeGlob.surcharge_gen indiquant quels générateurs sont en surcharge.

Le système reçoit aussi en entrée des informations sur les éventuelles pannes des générateurs.

7.4.2 Traitement des charges

En sortie de ce traitement (effectué par le nœud traite_charge itéré sur le tableau de charges EntreesGlob.mesure_chgs) on sait pour chaque charge si elle est susceptible d'être délestée. L'itération map≪traite_charge≫ reçoit des entrées que l'on peut découper en deux :

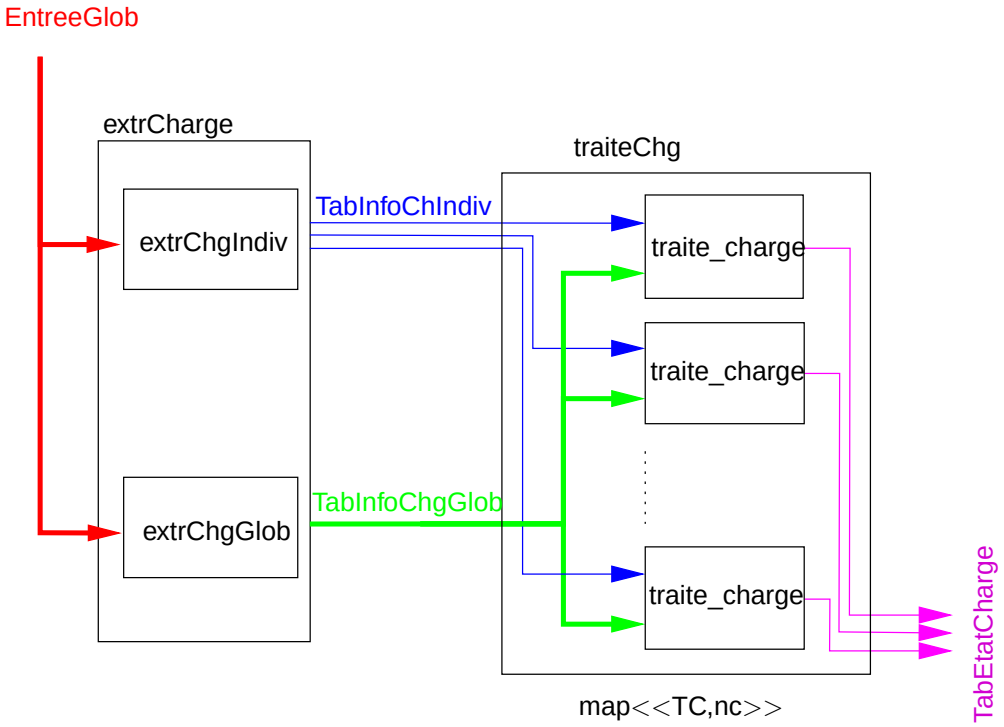


FIG. 7.2 – ELMU première partie : traitement de l'information liée aux charges.

- d'un côté une partie *information individuelle* qui concerne l'état des charges. Pour représenter cette information, on utilise le type T_InfoChgIndiv qui donne pour chaque charge sa valeur nominale ainsi que sa priorité ;
- de l'autre une partie *information globale* concernant l'association charges/générateurs (on utilise le type T_InfoChgGlob pour représenter cette information) ;

La priorité d'une charge peut prendre une valeur parmi les 3 suivantes :

- forte (PRIO_CHG_FORTE), signifiant que la charge ne doit jamais être débranchée ;
- moyenne (PRIO_CHG_MOY), signifiant que la charge peut être débranchée, si c'est vraiment nécessaire ;
- faible (PRIO_CHG_FAIBLE), signifiant que la charge peut être débranchée à tout moment.

Le résultat des traitements de charge est une information de type T_EtatCharge qui contient les informations nécessaires pour chaque charge (une valeur parmi EC_ON, EC_OFF, EC_LESTAGE et EC_DELESTAGE). Dans la version simplifiée que nous présentons on considérera qu'une charge peut être :

- branchée (EC_ON) ;
- débranchée (EC_OFF) ;
- en lestage (EC_LESTAGE), ce qui correspond à une montée en puissance progressive de la charge ;
- en délestage (EC_DELESTAGE).

7.4.3 Traitement des générateurs

Comme pour le traitement des charges, on va découper les informations en deux parties :

- d'un coté une partie individuelle représentée à l'aide du type T_InfoGenIndiv ;

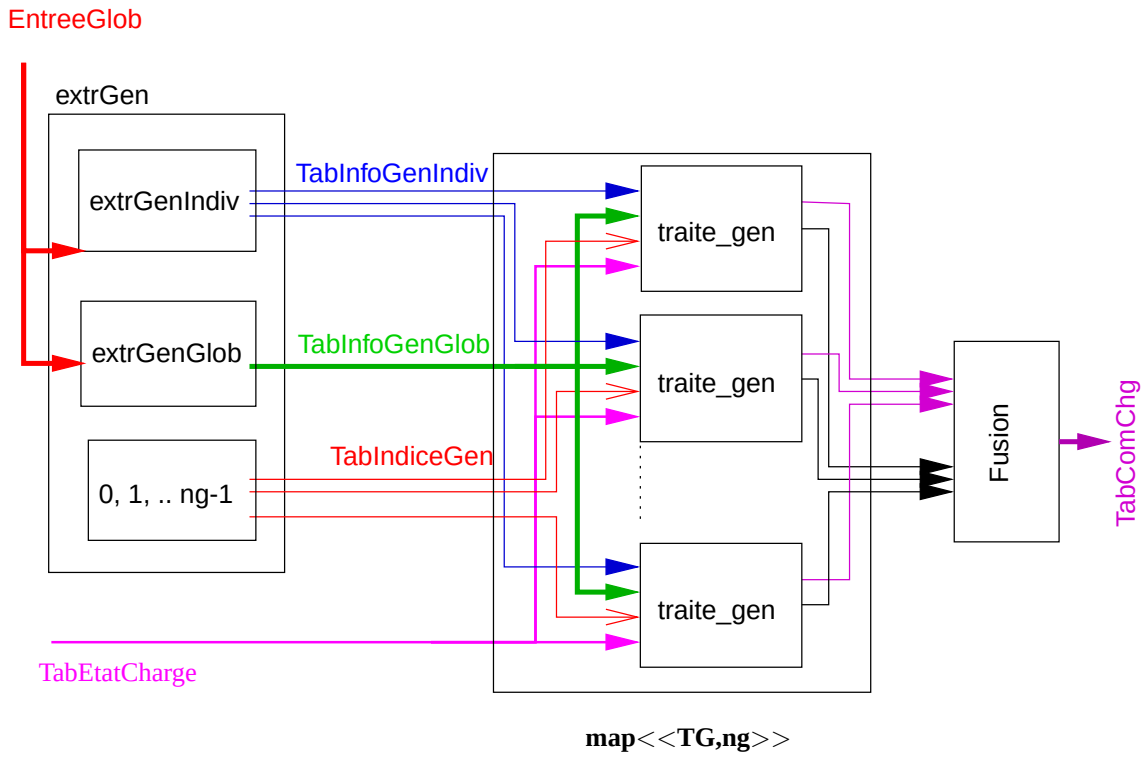


FIG. 7.3 – ELMU première partie : traitement de l'information liée aux générateurs.

- de l'autre une partie globale représentée à l'aide du type T_InfoGenGlob.

En plus, on fournit au traitement des générateurs la sortie produite par le traitement des charges (variable de type T_EtatChargeˆNB_CHARGES où NB_CHARGES est le nombre de charges). Le traitement de ces informations se fait à l'aide de l'itération map≪traite_gen≫.

En sortie, le nœud traiteGen fournit deux tableaux. AllTabComChg associe à chaque générateur une information sur chaque charge qui peut être, par exemple, de savoir si cette charge est éteinte ou allumée (ou encore dans certains modes non précisés). AllTabComVal associe à chaque générateur un tableau de NB_CHARGES booléens spécifiant (par exemple) si *ce* générateur doit continuer à produire de l'électricité pour telle ou telle charge.

7.4.4 Nœud de fusion

Le dernier composant Fusion sert uniquement à restructurer les données calculées par le traitement des générateurs en un seul et même tableau.

7.5 Codage des différents composants

7.5.1 Constantes et types

On commence par définir les constantes et les types nécessaires à la description du programme. On définit le nombre de générateurs et le nombre de charges du système. Cette information pourra facilement évoluer au cours du développement du système : seules ces quelques lignes seront à modifier !

```
const NB_CHARGES = 20;
const NB_GENE = 4;
```

On a aussi besoin de valeurs constantes pour le type T_EtatCharge (on pourrait utiliser un type énuméré) :

```
const EC_ON = 0;
const EC_OFF = 1;
const EC_LESTAGE = 2;
const EC_DELESTAGE = 3;
```

Globalement, une charge est déclarée EC_ON par le traitement des charges lorsqu'elle ne doit *surtout* pas être délestée ou éteinte. EC_OFF signifie qu'on peut sans problème éteindre la charge, EC_LESTAGE qu'on doit la laisser en lestage et EC_DELESTAGE qu'elle peut être délestée. On déclare aussi le type T_ComChg :

```
const COM_OFF = 0;
const COM_ON = 1;
```

C'est par cette information qu'on détermine si un générateur doit alimenter une charge (on lui fournit alors la valeur COM_ON pour cette charge), ou qu'il doit cesser de l'alimenter (valeur COM_OFF).

Dans l'application réelle les types sont potentiellement complexes. Nous avons grandement simplifié ces types (principalement remplacés par des valeurs entières) afin de ne nous intéresser qu'à la structure sous forme d'itération du programme. Par contre nous avons par endroit utilisé des structures qui paraissent inutiles (structures à un seul champ) mais qui ont l'avantage de pouvoir être facilement étendues sans changer la forme du programme.

On utilise ainsi des tableaux d'entiers pour les valeurs des charges et des générateurs ainsi que des tableaux de booléens principalement utile pour tester des égalités de tableaux d'entiers :

```
type INTNB_CHARGES = intˆNB_CHARGES;
type BOOLNB_CHARGES = boolˆNB_CHARGES;
type INTNB_GENE = intˆNB_GENE;
type BOOLNB_GENE = boolˆNB_GENE;
```

On donne ensuite (figure 7.4) les déclarations des types mentionnés plus haut (pour les entrées/sorties globales, les groupes de valeurs extraites pour les générateurs ou pour les charges...).

7.5.2 Nœud principal

Le nœud principal (donné à la figure 7.5) réalise, dans l'ordre :

- L'extraction des informations relatives aux charges et des générateurs à partir des entrées globales ;
- Le traitement des charges ;
- Le traitement des générateurs ;
- La fusion des informations venant du traitement des générateurs.

Il accepte comme entrée la variable EntreeGlob et renvoie la sortie TabComChg, tableau de valeurs de type T_ComChg.

```
type T_EntreeGlob = {chg2gen  : INTNB_CHARGES,
                     mesure_chgs  : INTNB_CHARGES,
                     mesure_gens  : INTNB_GENE};
                     priorite_chgs : T_niveauPrioriteNBC;
                     surcharge_gen : BOOLNB_GENE;}
type T_InfoChgIndiv = {mesure_chg : int,
                       priorite_chg : T_niveauPriorite};
type T_InfoChgGlob = {chg2gen : INTNB_CHARGES};
type T_EtatCharge = int;
type T_InfoGenIndiv = {mesure_gen : int,
                       surcharge_gen : bool}
type T_InfoGenGlob = {elt_bidon : int,
                      chg2gen : INTNB_CHARGES};
type T_ComChg = int;
type T_ComChgNB_CHARGES =T_ComChgˆNB_CHARGES;
```

FIG. 7.4 – Types utilisés dans le ELMU.

```
node normal(EntreeGlob : T_EntreeGlob)
returns (TabComChg : T_ComChgˆNB_CHARGES);
var
    TabInfoChgIndiv : T_InfoChgIndivˆNB_CHARGES;
    TabInfoChgGlob  : T_InfoChgGlobˆNB_CHARGES;
    TabEtatCharge  : T_EtatChargeˆNB_CHARGES;
    TabInfoGenIndiv : T_InfoGenIndivˆNB_GENE;
    TabInfoGenGlob  : T_InfoGenGlobˆNB_GENE;
    TabIndiceGen  : INTNB_GENE;
    AllTabComChg  : T_ComChgNB_CHARGESˆNB_GENE;
    AllTabComVal  : BOOLNB_CHARGESˆNB_GENE;
let
   TabInfoChgIndiv, TabInfoChgGlob = extrCharge(EntreeGlob);
   TabEtatCharge = traiteChg(TabInfoChgIndiv, TabInfoChgGlob);
   TabInfoGenIndiv, TabInfoGenGlob, TabIndiceGen = extrGen(EntreeGlob);
   AllTabComChg, AllTabComVal = traiteGen(TabIndiceGen,TabInfoGenIndiv,
                                          TabInfoGenGlob,TabEtatCharge);
   TabComChg = fusion_com(AllTabComChg, AllTabComVal);
tel
```

FIG. 7.5 – Nœud principal de l'application du ELMU.

7.5.3 Extraction des informations relatives aux charges

Le nœud extrCharge permet d'extraire et de structurer les données qui concernent uniquement le traitement des charges. Comme nous l'avons indiqué plus haut, ces informations sont divisées en deux groupes :

- d'un côté les informations individuelles des charges dont l'extraction est réalisée par le nœud extract_tab_info_chg_indiv ;
- de l'autre les informations globales extraites par le nœud extract_info_chg_glob.

L'implémentation de extrCharge contient donc simplement un appel à chacun de ces deux nœuds (voir figure 7.6)

7.5.4 Extraction des informations relatives aux générateurs

Comme pour les informations relatives aux charges, on utilise deux nœuds extract_tab_info_gen_indiv et extract_info_gen_glob pour extraire les informations individuelles et globales des générateurs. Le nœud extrGen, donné à la figure 7.6, calcule aussi un tableau d'entiers TabIndiceGen (que nous avons mentionné plus haut) qui contient les numéros des générateurs.

```
node extrCharge(EntreeGlob : T_EntreeGlob)
returns (TabInfoChgIndiv : T_InfoChgIndivˆNB_CHARGES;
         TabInfoChgGlob  : T_InfoChgGlobˆNB_CHARGES);
let
   TabInfoChgIndiv = extract_tab_info_chg_indiv(EntreeGlob);
   TabInfoChgGlob = extract_info_chg_glob(EntreeGlob)ˆNB_CHARGES;
tel
                    ---------------------------------------

node extrGen(EntreeGlob : T_EntreeGlob)
returns (TabInfoGenIndiv : T_InfoGenIndivˆNB_GENE;
         TabInfoGenGlob  : T_InfoGenGlobˆNB_GENE;
         TabIndiceGen  : INTNB_GENE;);
let
   TabInfoGenIndiv = extract_tab_info_gen_indiv(EntreeGlob);
   TabInfoGenGlob = extract_info_gen_glob(EntreeGlob)ˆNB_GENE;
   TabIndiceGen = fill≪incr_acc; NB_GENE≫(0);
tel
```

FIG. 7.6 – Les nœuds extrCharge et extrGen.

7.5.5 Traitement des charges

Le traitement des charges se fait à l'aide du nœud traiteCharge qui itère le nœud traite_charge mentionné plus haut. Comme nous l'avons dit plus haut, nous ne connaissons pas exactement le fonctionnement du nœud traite_charge. On peut néanmoins imaginer la version suivante, implémentée à la figure 7.7.

Si la charge est de priorité maximale PRIO_CHG_FORTE alors elle doit rester allumée en tout temps. Si la charge est d'une priorité moyenne PRIO_CHG_MOY, elle doit rester en lestage pendant un nombre de cycles MAX_ONTIME_FOR_MOY_PRIO. Si elle est restée en lestage au delà de

cette limite, elle peut être délestée. Enfin, si la charge est de priorité la plus faible, elle doit rester en lestage pendant au moins MAX_ONTIME_FOR_FAIBLE_PRIO cycles. Sinon, on peut directement l'éteindre.

Notons que la valeur de EtatCharge ne représente pas directement une *commande* : si on émet EC_ON (ou EC_DELESTAGE), la charge *doit* rester allumée (ou *doit* être délestée), mais si on émet EC_LESTAGE (ou EC_OFF), la charge *peut* être lestée (ou éteinte).

```
node traiteChg(TabInfoChgIndiv : T_InfoChgIndivˆNB_CHARGES;
               TabInfoChgGlob : T_InfoChgGlobˆNB_CHARGES)
returns (TabEtatCharge : T_EtatChargeˆNB_CHARGES);
let
   TabEtatCharge = map≪traite_charge; NB_CHARGES≫(
                        TabInfoChgIndiv,TabInfoChgGlob);
tel
-------------------------------------------------------

node traite_charge(InfoChgIndiv : T_InfoChgIndiv;
                   InfoChgGlob  : T_InfoChgGlob)
returns (EtatCharge : T_EtatCharge)
var countLestTime : int;
let
   countLestTime = 0 -> if (pre EtatCharge=EC_LESTAGE or pre EtatCharge=EC_ON)
                        then pre(countLestTime + 1)
                        else 0;
   EtatCharge = EC_OFF ->
                if(InfoChgIndiv.priorite_chg = PRIO_CHG_FORTE)
                then EC_ON
                else if InfoChgIndiv.priorite_chg = PRIO_CHG_MOY
                     then if countLestTime > MAX_ONTIME_FOR_MOY_PRIO
                          then EC_DELESTAGE
                          else EC_LESTAGE
                     else if InfoChgIndiv.priorite_chg = PRIO_CHG_FAIBLE
                          then if countLestTime > MAX_ONTIME_FOR_FAIBLE_PRIO
                               then EC_OFF
                               else EC_LESTAGE
                          else pre(EtatCharge);
tel
```

FIG. 7.7 – Traitement d'une charge.

7.5.6 Traitement des générateurs

Le traitement des générateurs se fait à l'aide du nœud traiteGen qui itère le nœud traite_gen mentionné plus haut. Ces nœuds sont décrits à la figure 7.8. Le tableau TabComChg contient, pour chaque charge, la valeur COM_ON si elle doit être maintenue allumée et COM_OFF si elle doit être éteinte. Pour décider de cette valeur, le nœud iterTraite_gen regarde si le générateur qui fournit de l'énergie à la charge est en surcharge. Si il est en surcharge et que la charge peut être délestée ou éteinte, il ordonne de l'éteindre (message COM_OFF). Si elle ne doit pas être éteinte (condition elt_tabEtatCharge = EC_ON or elt_tabEtatCharge = EC_LESTAGE à vrai), on commande de

la laisser allumée (message COM_ON). Le tableau TabComVal est défini comme suit : son i-ème élément est vrai si et seulement si le générateur d'indice indice_gen fournit de l'énergie à la charge i. Il ne nous intéressera pas particulièrement.

```
node traiteGen(TabIndiceGen  : INTNB_GENE;
               TabInfoGenIndiv : T_InfoGenIndivˆNB_GENE;
               TabInfoGenGlob  : T_InfoGenGlobˆNB_GENE;
               TabEtatCharge : T_EtatChargeˆNB_CHARGES)
returns (AllTabComChg :T_ComChgNB_CHARGESˆNB_GENE;
         AllTabComVal  : BOOLNB_CHARGESˆNB_GENE);
let
   AllTabComChg, AllTabComVal = map≪traite_gen; NB_GENE≫ (
                                    TabIndiceGen,TabInfoGenIndiv,
                                    TabInfoGenGlob,TabEtatChargeˆNB_GENE);
tel
                    ------------------------------------------

node traite_gen(indice_gen : int;
                InfoGenIndiv  : T_InfoGenIndiv;
                InfoGenGlob : T_InfoGenGlob;
                TabEtatCharge : T_EtatChargeNB_Charges)
returns (TabComChg  : T_ComChgˆNB_Charges;
         TabComVal : BOOLNB_Charges);
var TabIndiceGen : INTNBC; res_fill : int; acc_out : Taccumulateur_traite_gen;
let
   acc_out, TabComChg = map_red≪iterTraite_gen;NB_CHARGES≫(
                        {ind_gen = indice_gen;
                         estEnSurcharge = InfoGenIndiv.surcharge_gen},
                        InfoGenGlob.chg2gen,TabEtatCharge);
   TabComVal = map≪egal_indice;NB_CHARGES≫(TabIndiceGen, InfoGenGlob.chg2gen);
   res_fill, TabIndiceGen = fill≪copie;NB_CHARGES≫(indice_gen);
tel
                    ------------------------------------------

node iterTraite_gen(acc_in : Taccumulateur_traite_gen;
                    elt_infoGenGlob : int;
                    elt_tabEtatCharge : T_EtatCharge)
returns (acc_out : Taccumulateur_traite_gen;
         elt_tabComChg : T_ComChg);
let
   acc_out = acc_in;
   elt_tabComChg = COM_ON -> if((elt_infoGenGlob = acc_in.ind_gen)
                                and acc_in.estEnSurcharge)
                           then if(elt_tabEtatCharge = EC_ON)
                                   or (elt_tabEtatCharge = EC_LESTAGE)
                                then COM_ON
                                else COM_OFF
                           else pre(elt_tabComChg);
tel
```

FIG. 7.8 – Traitement des générateurs.

7.6 Propriété globale du système

Une propriété globale du ELMU est que si un générateur est en surcharge (information fournie par le tableau EntreeGlob.surcharge_gen) à un instant t, alors il ne doit plus être à un instant t+δ_t, où δ_t est une constante donnée dans la spécification.

Le fait qu'un générateur soit en surcharge est une information fournie par l'environnement (c'est une mesure *physique*) . Dans le système, on l'interprète de la manière suivante : un générateur est en surcharge si et seulement si la somme cumulée des charges qui lui sont associées (information calculée sur les entrées du ELMU) est plus grande qu'un certain seuil de charge maximal que le générateur peut supporter. Ce seuil est lui aussi fixé par une constante (plus exactement par un tableau de constantes donnant les seuil de surcharge pour tous les générateurs). La fonctionnalité principale du système est de réagir à l'information qu'il reçoit de l'environnement en activant les commandes correspondantes (voir figure 7.9).

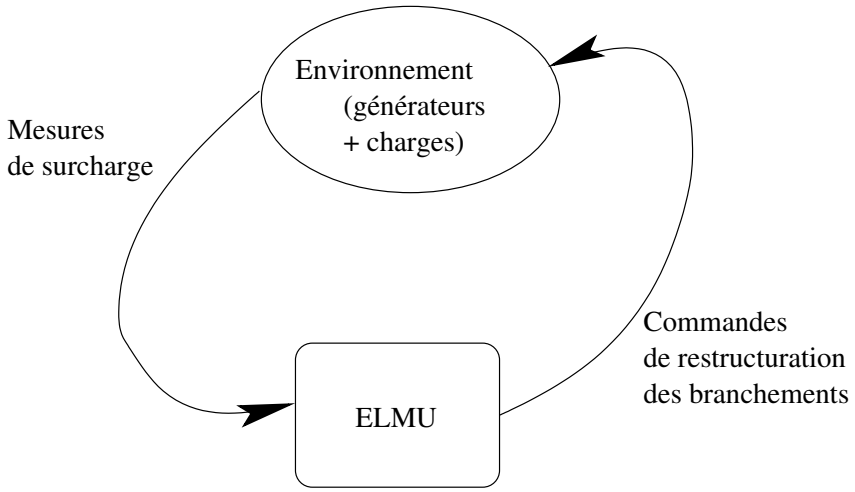


FIG. 7.9 – Interaction du ELMU avec son environnement.

Le système ELMU réalise, entre autre, la satisfaction de cette propriété : dès qu'un générateur se déclare en surcharge, le ELMU va tenter de rééquilibrer la répartition des charges selon un principe proche de celui exposé aux paragraphes 7.5.5 et 7.5.6. Ce rééquilibrage ne se réalise pas forcément instantanément, puisqu'il n'est pas toujours possible de débrancher immédiatement une charge, il se peut qu'elle doive rester allumée plusieurs cycles. Par ailleurs, le rééquilibrage ne se fait pas que par débranchement de certaines charges. Il est aussi possible d'associer des charges qui ne doivent pas être débranchées à un autre générateur. Cet aspect n'a pas été pris en compte dans l'implémentation que nous avons donnée plus haut.

7.7 Commentaires

Nous avons donné ci-dessus une version *adaptée* de la fonction de calcul des délestages et de réengagements des charges. Le détail de fonctionnement de ce calcul (basé sur un algorithme type « sac à dos ») n'a jamais été disponible. Nous n'avions qu'une idée très vague de la façon dont les réarrangements des charges sur les générateurs sont faites. Les choix que nous avons fait pour les traitements

des charges et des générateurs peuvent paraître, pour cette raison, simpliste. Néanmoins, cette étude de cas nous a principalement intéressé pour deux aspects :

- La régularité qu'on a pu identifier dans les traitements des charges ou des générateurs (exprimables à l'aide de maps) ;
- L'utilisation d'une bibliothèque de composants pour lesquels, le plus souvent, un contrat est donné de manière informelle.

En ce qui concerne les itérations, l'application a pu être entièrement réécrite en les utilisant. Le gain en terme de taille de la spécification a été très important. Le code C généré à partir des planches SCADE a lui aussi été réduit de manière importante. A titre de comparaison, le code C obtenu à partir de la spécification *sans* itération ne pouvait même pas être compilé par le compilateur GCC car il était *trop gros* (de l'ordre de 300 000 lignes de code). Le code C obtenu pour la version avec itérateurs est beaucoup plus petit : il ne dépasse pas le millier de lignes de code.

Le gain en terme d'effort de spécification est aussi important : la société Airbus a par exemple souhaité faire évoluer son application pour qu'elle prenne en compte 5 générateurs, au lieu de 4 comme c'était spécifié dans l'application originale. L'application proposée est maintenant entièrement *générique* sur le nombre de générateurs et la modification est triviale à effectuer : il suffit de changer la valeur d'une constante. Si l'on ne se sert pas d'itérations, il faut recopier une partie du code (déjà dupliquée 4 fois) une cinquième fois pour gérer le fonctionnement du cinquième générateur.

Cette application met aussi en évidence l'utilité des contrats pour la spécification de composants de base utilisés intensivement. On rejoint ici l'idée de bibliothèques de composants évoquée au paragraphe 5.4 qui sont spécifiés par leur contrat ce qui aide leur réutilisation.

Partie IV

Exploitation

Introduction

L'idée principale du travail présenté dans cette thèse est celle de la nécessité d'une méthodologie de développement correct des systèmes réactifs. Dans la partie II, nous avons fait des propositions d'extension du langage LUSTRE pour répondre à des besoins de *description* des systèmes. Ces propositions sont :

- d'un côté des itérateurs pour la description des systèmes réguliers ;
- d'un autre des contrats pour la spécification locale des composants.

Les exemples de la partie III illustrent l'utilité de ces extensions. Nous avons notamment insisté sur l'intérêt de la généricité impliquée par les itérations, et sur la progression dans la démarche de spécification permise par les contrats.

Les contrats et les itérations ont été proposés pour répondre à des besoins rencontrés dans ces deux études de cas. Celles-ci n'ont pas été proposées à *posteriori* comme des faire-valoirs de nos propositions mais sont à prendre comme les réelles motivations de ce travail.

Nous avons aussi remarqué que ces extensions (aussi bien les itérations que les contrats) augmentent l'effort de spécification : elles demandent une rigueur de développement importante (rigueur qui doit être vue comme un avantage dans toute méthode de développement), notamment pour la spécification des contrats qui imposent de décrire d'une manière relativement précise les interactions des composants avec leurs environnements. L'utilisation de ces extensions peut paraître lourde voire superflue à certains utilisateurs. Pour les convaincre, il faut encore montrer quel gain on tire d'un tel effort.

Dans la présente partie, nous étudions les possibilités d'exploitation de ces différents aspects langage dans un but de validation des applications. Nous souhaitons montrer que cet effort de spécification impliqué n'est pas vain, mais permet la mise en place de diverses manipulations aidant à la validation.

Les chapitres 8 et 9 proposent un catalogue de manipulations élémentaires :

- sur les itérations (découpage de propriétés portant sur des tableaux pour les ramener à des propriétés plus petites portant sur des éléments de tableaux) ;
- sur les contrats (mise en parallèle du contrat d'un composant avec son implémentation, vérification de la validité de la composition de deux composants, etc.) ;
- mais aussi mixtes (portant à la fois sur les itérations et les contrats).

Nous verrons que ces manipulations sont éventuellement automatisables par traduction en objectifs pour les outils de preuve habituels. On peut ainsi mettre à profit l'ensemble d'outils et de méthodes de validation pour valider des parties de l'application concernée.

Mais ces manipulations n'ont pas nécessairement un but de preuve globale d'un système. Telles que, elles ne permettent pas de s'assurer de la validité d'une application mais représentent des outils de base de manipulation des programmes pour *aider* à leur validation.

Autour de ces manipulations de base, nous proposons une *ébauche* de méthodologie de développement rigoureux. Cette méthodologie n'est pas automatique ni complète : elle vise uniquement à mettre en place une rigueur de développement et de validation qui doit amener à une plus grande assurance

dans la validité d'une application.

Dans le chapitre 10, nous revenons sur les exemples des chapitres 6 et 7, afin de montrer comment nos manipulations permettent d'*aider* à mieux comprendre et à valider ces applications. Nous souhaitons insister sur le fait que l'effort de spécification mis en évidence aussi bien dans le système gyroscopique que dans le ELMU est très utile puisqu'il offre des possibilités de validation supplémentaires. Une version préliminaire de cette mise en application (notamment concernant le système gyroscopique) a été décrite dans [MM04b].

Le chapitre 11 décrit un prototype que nous avons développé durant cette thèse pour valider et illustrer les différentes manipulations des chapitres 8 et 9. Cet outil n'est pas un outil de preuve automatique, mais une interface permettant d'appliquer les manipulations que nous proposons. À partir d'un programme donné, l'outil génère simplement des objectifs de preuve qui peuvent ensuite être fournis à différents outils de vérification.

Chapitre 8

Manipulation des itérations

Nous avons présenté au chapitre 4 les itérateurs de tableau et nous avons insisté sur le fait que les tableaux permettaient de structurer *de manière régulière les spécifications. Nous montrons maintenant comment il est possible de prendre en compte cette forme de programmation dans la phase de validation des systèmes. Nous donnons dans un premier temps (section 8.2) des exemples de propriétés que les itérateurs permettent de décrire. Le paragraphe 8.3 synthétise les travaux connexes à notre approche. À la section 8.4, nous présentons une première méthode qui a consisté à étendre la sémantique de* LUSTRE *en PVS pour utiliser ce dernier pour la preuve des propriétés itératives. Enfin, la section 8.5 décrit un schéma de preuve pour les programmes avec itérateurs.*

8.1 Introduction

Au chapitre 4 nous avons introduit des opérateurs qui permettent de décrire des systèmes réguliers. Nous avons déjà vu les avantages de ces structures de langage à différents niveaux, d'un point de vue spécification bien sûr, mais aussi d'un point de vue compilation et efficacité du code produit par le compilateur.

Les domaines d'application que nous visons nécessitent, nous l'avons déjà dit, de pouvoir valider formellement les programmes. Les itérateurs n'augmentent pas le pouvoir d'expression du langage : il est possible de traduire les programmes itératifs en du code LUSTRE sans tableau et on peut alors utiliser les outils de validation standard de LUSTRE pour prouver des propriétés.

Cette approche n'est pas très satisfaisante : les programmes à traiter sont alors très gros et on rencontre un problème d'explosion combinatoire pour tous les outils disponibles. De plus, l'expansion des tableaux pour la validation interdit toute prise en compte de la structuration des programmes induite par les itérateurs, alors qu'elle pourrait être très importante dans la validation des propriétés. On aimerait des méthodologies de validation qui prennent cette structuration des programmes en compte.

Le but de ce chapitre est de proposer une méthodologie qui permette de prendre en compte la structure itérative d'un programme LUSTRE dans la phase de validation. Dans un premier temps (section 8.2), nous donnons des exemples de propriétés qui nous intéressent et nous étudions comment ces propriétés peuvent être exprimées en LUSTRE. Ensuite, nous présentons à la section 8.3 des travaux traitant de la preuve de programmes réguliers. Ces travaux se partagent en deux grandes catégories : la première concerne la preuve des réseaux symétriques et la seconde la validation des algorithmes itératifs généraux.

Dans la section 8.4, nous présentons une première approche, étendant le travail de [DC00] aux programmes LUSTRE avec itérateurs. Nous concluons que la difficulté d'écriture et de manipulations des programmes dans le langage de spécification fourni par PVS rend cette approche difficile à poursuivre (nous rejoignons en cela les conclusions de [DC00]).

Enfin, nous proposons au paragraphe 8.5 un mécanisme d'extraction d'objectifs de preuve : à partir d'une propriété à prouver sur un programme manipulant des tableaux, nous générons un ensemble de propriétés (plus petites) à prouver sur un programme manipulant des éléments de tableaux. Les objectifs de preuve ainsi construits sont suffisants, mais pas forcément nécessaires pour la preuve de la propriété initiale.

8.2 Propriétés sur tableaux

Comme nous l'avons vu plus haut, une propriété en LUSTRE est généralement exprimée à l'aide d'un *observateur*, c'est-à-dire un nœud possédant pour entrées les entrées/sorties du programme considéré et une seule sortie booléenne représentant la valeur de vérité de la propriété.

Une propriété peut porter sur des variables de type tableau en utilisant toute la puissance du langage.

En général, on va s'intéresser à des propriétés exprimées à l'aide de réductions à sortie booléenne ou des propriétés portant sur des résultats d'itérations (de type quelconque).

Dans ce paragraphe, nous allons nous intéresser à l'expression de telles propriétés. Dans un premier temps, nous étudierons les propriétés qui portent sur les variables tableaux d'un programme. Ces propriétés seront en générale exprimées à l'aide de réduction à résultat booléen. Bien sûr, en même temps qu'une propriété itérative, on pourra donner une hypothèse itérative. Nous introduirons aussi un nouvel itérateur, appelé « *forall* » qui nous permettra d'exprimer des propriétés symétriques.

Ensuite, nous décrirons des propriétés, par exemples temporelles, exprimées à l'aide d'itérateurs, mais ne portant pas nécessairement sur des tableaux du programme considéré.

8.2.1 Propriétés symétriques : l'opérateur forall

Il est parfois utile d'exprimer qu'une propriété s'applique à tous les éléments d'un tableau. Dans ce cas, elle est dite *symétrique*. On trouvera notamment un exemple de propriété symétrique dans l'application de traitement de données gyroscopiques décrite au chapitre 6.

Par exemple, on peut vouloir écrire que tous les éléments du tableau Out (dont il était question plus haut) croissent dans le temps. On peut décrire une telle propriété à l'aide d'une simple réduction : à chaque étage de l'itération, l'accumulateur booléen est vrai si tous les éléments jusqu'à l'étage courant croissent. Pour cela, on doit décrire un nœud itérable permettant de tester si un élément du tableau satisfait la propriété et qui propage la valeur de vérité de la propriété le long de l'itération :

```
node iteratedProp(acc_in : bool; elt : int) returns (acc_out : bool);
let
   acc_out = acc_in and (true -> elt > pre(elt));
tel
```

La propriété s'exprime alors avec l'itération suivante :

```
node Obs(In, Out : int^10) returns (ok : bool);
let
   ok = red<<iteratedProp;10>>(true,Out);
tel
```

Le problème de cette forme est qu'il n'est pas du tout évident de reconnaître la symétrie de la propriété. On pourrait se contenter de dire que seuls les nœuds à accumulateur booléen et tels que l'accumulateur de sortie soit calculé avec une équation de la forme :

acc_out = acc_in **and** ...

définissent une symétrie. Mais cette approche présente deux inconvénients :

- Premièrement, elle contraint le programmeur à décrire ses propriétés symétriques avec une forme trop commune (rien n'indique que la conjonction du nœud itérée représente une propriété symétrique) et surtout *enfouie* : la symétrie n'est pas apparente au niveau de la réduction, mais uniquement *dans* le nœud itéré ;
- Deuxièmement, l'identification de la symétrie dans ce cas demande une analyse fine du nœud itéré.

Il nous paraît plus intéressant de proposer un opérateur particulier pour exprimer ces propriétés symétriques. L'expression de telles propriétés sera ainsi rendue explicite et leur manipulation plus facile.

Nous introduisons donc l'opérateur forall défini comme suit. L'opération forall consiste à étudier la validité d'une propriété scalaire $\mathsf{P} = \lambda t.b$ sur tous les éléments d'un ou plusieurs tableaux (T, dans notre exemple). Le résultat est vrai si et seulement si tous les éléments de T satisfont la propriété P. La syntaxe abstraite de l'opérateur forall est la suivante :

$$ok = forall(g, T).$$

La sémantique de cette équation est décrite par l'équation suivante :

$$ok = \bigwedge_{i=0}^{i=size-1} g(T[i]).$$

Les syntaxe et sémantique LUSTRE sont données ci-dessous. Si P est un observateur LUSTRE de profil :

$$\tau_1 \times \tau_2 \times \ldots \times \tau_l \rightarrow bool,$$

et si n est une *constante statique*, alors forall<<P;n>> est considéré comme un nœud de profil :

$$\tau_1\hat{}\mathrm{n} \times \tau_2\hat{}\mathrm{n} \times \ldots \times \tau_l\hat{}\mathrm{n} \rightarrow bool.$$

Toute propriété exprimée à l'aide d'un forall peut se réécrire en une propriété utilisant un red.

EXEMPLE 33 — On peut réécrire l'exemple que nous avons donné plus haut avec l'opérateur forall. Il faut tout d'abord changer le nœud itéré : il ne nécessite plus d'accumulateur d'entrée. On a :

```
node iteratedProp2(elt : int) returns (acc_out : bool);
let
   acc_out = true -> elt > pre(elt);
tel
```

La propriété s'exprime alors avec l'observateur suivant :

```
node Obs(In, Out : intˆ10) returns (ok : bool);
let
   ok = forall≪iteratedProp2;10≫(Out);
tel
```

— FIN DE L'EXEMPLE 33

8.2.2 Propriétés combinatoires sur tableaux

Nous présentons maintenant deux exemples de propriétés portant sur des tableaux. Ces propriétés sont purement combinatoires, nous montrerons au paragraphe 8.2.3 des propriétés temporelles.

8.2.2.1 Moyenne incrémentale

Le programme suivant est un observateur purement combinatoire exprimant une propriété classique sur un tableau d'entiers. On veut exprimer le fait que si tous les éléments d'un tableau sont positifs, alors la moyenne de ces éléments est positive aussi.

Il nous faut exprimer tout d'abord que tous les éléments d'un tableau sont positifs. Pour cela, on écrit le nœud tousPositifs :

```
node tousPositifs(tableau : intˆ10) returns (ok : bool);
let
   ok = forall≪unPositif;10≫(tableau);
tel
```

où le nœud $unPositif$ décrit le fait qu'un entier est positif :

```
node unPositif(element : int) returns (ok : bool);
let
   ok = element>0;
tel
```

En utilisant le nœud Moyenne, donné à la figure 8.1, on décrit la propriété voulue avec l'observateur suivant :

```
node Prop(tableau : intˆ10) returns (ok : bool);
var moy : int;
let
   moy = Moyenne(tableau);
   ok = tousPositifs(tableau) => (moy>0);
tel
```

```
node Moyenne(tableau : int^10) returns (moyenne : int);
var sum : int;
let
   sum = red<<+;10>>(0,tableau);
   moyenne = sum div 10;
tel
```

FIG. 8.1 – Nœud calculant la moyenne d'un tableau d'entiers.

8.2.2.2 *« au plus un vrai »*

On considère un tableau tab de booléens d'une taille quelconque (on a choisi 10 pour l'exemple). On veut s'assurer que, parmi les éléments du tableau, au plus 1 seul est vrai à un instant donné. On peut décrire ce problème avec un compteur, mais nous choisissons de donner une version entièrement booléenne. Pour cela, nous utilisons le type structuré atMost_struct suivant :

```
type atMost_struct = {hasBeen0 : bool,
                      hasBeen1 : bool,
                      hasBeen2 : bool,
                      atMost1 : bool};
```

Nous allons utiliser une variable de type atMost_struct comme accumulateur d'une itération sur tab. En sortie de chaque étage de l'itération :

- hasBeen0 est vrai si et seulement si on a vu zéro élément du tableau à vrai ;
- hasBeen1 est vrai si et seulement si on a vu un élément du tableau à vrai (soit on en avait déjà vu un avant soit on n'en avait vu aucun et l'élément courant est vrai) ;
- hasBeen2 est vrai si et seulement si on a vu deux éléments du tableau à vrai (soit on en avait déjà vu au moins deux à l'étape précédente de l'itération, soit on en avait vu 1 et l'élément courant est vrai) ;
- atMost1 passe à faux si et seulement si on a vu *au moins* deux éléments du tableau à vrai (ce champ vaut exactement not(hasBeen2).

Ce calcul est effectué par le nœud atMostOne_iter :

```
node atMostOne_iter(accu_in : atMost_struct;
                    elt_in : bool)
returns (accu_out : atMost_struct);
let
   accu_out = {hasBeen0 = accu_in.hasBeen0,
               hasBeen1 = accu_in.hasBeen1 or
                          (accu_in.hasBeen0 and elt_in),
               hasBeen2 = accu_in.hasBeen2 or
                          (accu_in.hasBeen1 and elt_in),
               atMost1 = not(accu_in.hasBeen2 or
                          (accu_in.hasBeen1 and elt_in))};
tel
```

Le nœud atMostOne permet de vérifier qu'un tableau de booléen satisfait la propriété désirée. il itère le nœud atMostOne_iter en initialisant l'accumulateur avec hasBeen0 = true, hasBeen1 =

false, hasBeen2 = false et atMost1 = true puisqu'aucun élément n'a encore été détecté à vrai. La propriété est vrai si le champ acc_out de la sortie de l'itération est vrai.

```
node atMostOne(tab : bool^10)
returns (ok : bool);
var accu_out : atMost_struct;
let
   accu_out = red<<atMostOne_iter;10>>(
                        {hasBeen0 = true, hasBeen1 = false,
                         hasBeen2 = false, atMost1 = true},
                        tab);
   ok = accu_out.atMost1;
tel
```

8.2.3 Expression de propriétés temporelles avec les tableaux

Les deux propriétés que nous avons décrit dans le paragraphe précédent portaient sur des tableaux utilisés dans le programme. Les programmes qui suivent ne manipulent pas de tableaux. Mais les propriétés que nous allons exprimer nécessitent des itérations pour exprimer des propriétés portant sur le passé des variables du programme : on va vouloir par exemple exprimer le fait qu'une variable satisfait une propriété pendant plusieurs instants consécutifs. Une itération peut être utilisée pour exprimer cette propriété en mémorisant les valeurs successives de certaines variables.

Ces propriétés seront traitées de la même façon que celles présentées dans le paragraphe précédent.

8.2.3.1 Mémoire à n instants en arrière

La propriété suivante utilise la mémoire à n instants décrite au paragraphe 4.3.5.2. Ce programme peut être utilisé pour exprimer toute sorte de propriété portant sur une mémoire bornée de l'état d'un système (périodicité, croissance d'un flot entier, etc...). On donne ici un observateur pour la propriété « un signal booléen est vrai pendant 10 instants consécutifs ».

```
node AllTrue(acc_in, elt : bool) returns (acc_out : bool);
let
   acc_out = acc_in and elt;
tel

node vraiNCoupsPasses(x : bool) returns (ok : bool);
var Tab : bool^10;
let
   Tab = memoireNInstants(x);
   ok = red<<AllTrue;10>>(true, Tab);
tel
```

Prouver que ok est vrai revient à prouver que x est toujours vrai. Nous verrons dans l'exemple 8.2.4.1 l'utilité de cette propriété.

8.2.4 Expression de contrats à l'aide d'itérations

Ces propriétés itératives peuvent maintenant servir dans la description des contrats de composants. L'exemple du paragraphe 8.2.4.1 décrit un contrat d'un noeud qui ne manipule pas de tableau. Comme la propriété du paragraphe précédent, ce contrat nécessite l'utilisation d'itérations. Le composant du paragraphe 8.2.4.2 possède quant à lui des paramètres tableaux.

8.2.4.1 Un composant à contrat itératif dans le ELMU

Une variante du composant rMFF présenté au paragraphe 5.3.1 est utilisée dans le ELMU. Ce composant, appelé MTRIG possède une entrée E et une sortie S booléennes. La spécification qui en est donnée indique que si E est vrai à un instant donné, alors S est vrai pendant les cycle instants suivants. À la différence du rMFF, MTRIG est un mono-stable *non-rechargeable* : si E passe de faux à vrai alors que S est vrai, il est ignoré.

On donne tout d'abord la taille du cycle du MTRIG :

```
const cycle = 10;
```

On décrit ensuite le contrat de ce composant. L'assertion associée est vide : l'entrée du composant peut prendre n'importe quelle valeur au cours du temps :

```
node assumeMTRIG(E : bool) returns (okAssume : bool);
let
   okAssume = true;
tel
```

La garantie du MTRIG stipule qu'à tout instant, si E a été vrai cycle instants auparavant et que S n'était pas vrai à cet instant, alors S a été vrai pendant les cycles instants qui ont suivi. La propriété « *Être vrai* cycles *instants auparavant* » est représentée à l'aide du nœud vraiNCoupsAvant.

```
node vraiNCoupsAvant(V : bool) returns (ok : bool);
let
   ok = red≪vraiAvant;cycle≫(V, trueˆcycle);
tel

node vraiAvant(acc_in1, elt_bidon1 : bool) returns (acc_out1 : bool);
let
   acc_out1 = false -> pre(acc_in1);
tel
```

La propriété « *Être vrai pendant les* cycles *instants qui viennent de s'écouler* » est représentée à l'aide du nœud vraiNCoupsPasses donné au paragraphe 8.2.3.1.

La propriété désirée est encodée à l'aide du nœud guaranteeMTRIG ci-dessous.

```
node guaranteeMTRIG(E, S : bool) returns (okGuarantee : bool);
var EvraiAvant : bool; SpasVraiAvant : bool;
    SVraiNcoupsPasses : bool;
let
   EvraiAvant = vraiNCoupsAvant(E);
   SpasVraiAvant = not(vraiNCoupsAvant(S));
   SVraiNCoupsPasses = vraiNCoupsPasses(S);
   okGuarantee = (EvraiAvant and SpasVraiAvant) => SVraiNCoupsPasses;
tel
```

Remarque 9 *On notera, dans les propriétés que nous décrivons pour exprimer cette garantie, l'utilisation qui est faite des itérations alors que les données manipulées par le programme* MTRIG *ne sont pas des tableaux. Les propriétés qu'on exprime cependant sur ce programme dépendent d'un nombre d'instants fini et connu statiquement. Ce nombre est un paramètre générique de la spécification du composant : on peut très bien en changer la valeur au cours du processus de développement. Les itérations permettent de simplifier l'écriture de tels programmes. Imaginons qu'on ait écrit une version de* vraiNCoupsPasses *sans itérations. Si l'on veut changer le nombre d'instants qui s'écoulent entre l'émission du dernier signal d'entrée et celle du dernier signal de sortie, il faudrait alors rajouter* à la main *une instance de l'opérateur* pre. *Avec les itérations, on a juste à changer la valeur de la taille de l'itération.*

Finalement, voici le composant MTRIG qui définit de façon déterministe la valeur de la sortie S en fonction de l'entrée E. On utilise un compteur qui est initialisé à la valeur cycle à chaque fois qu'il reçoit une entrée (et qu'il n'est pas *déjà* en train de compter). La sortie S est vraie :

- si on est en train de compter (c'est-à-dire qu'on a déjà vu E à vrai il y a moins de cycle instants) ;
- ou si E est vrai. Cette condition est nécessaire car le comptage commence avec un instant de décalage par rapport à l'occurrence d'un signal vrai sur E.

On a :

```
node MTRIG(E : bool) returns (S : bool)
%ASSUME:assumeMTRIG%
%GUARANTEE:guaranteeMTRIG%;
variscounting : bool;
    counter : int;
let
   S = E or iscounting;
   iscounting = E -> pre(counter > 1);
   counter = (if E then cycle else 0)
             -> (if (E and not(iscounting)) then cycle else pre(counter)-1);
tel
```

Remarquons simplement la différence principale avec le composant rMFF : ici on n'attend pas un instant pour émettre S sur un front motant de E, alors qu'il y avait un temps de décalage dans le rMMF.

8.2.4.2 Contrat de noeud à paramètres tableaux

On donne maintenant un contrat d'un noeud (voir figure 8.2) qui a pour entrée un tableau I et pour sortie un tableau T. Contrairement à l'exemple précédent, les tableaux font partie du programme, il ne sont pas rajoutés pour exprimer une propriété.

```
node N(I : intˆsize) returns (T : intˆsize);
%ASSUME:assumeN%
%GUARANTEE:guaranteeN%
let
   ...
tel

node assumeN(I : intˆsize) returns (assumeOK : bool);
let
   assumeOK = red≪unPositif;size≫(true;I);
tel

node guaranteeN(I,T : intˆsize) returns (guaranteeOK : bool);
var sum : int;
     tousPositif : bool;
let
   sum = red≪+;size≫(0;T);
   tousPositif = red≪unPositif;size≫(true;T);
   guaranteeOK = unPositif(sum) or tousPositif;
tel
```

FIG. 8.2 – Un contrat itératif pour un noeud N.

L'assertion assumeN stipule que tous les éléments du tableau I sont positifs. La garantie guaranteeN exprime la propriété suivante :

« Les éléments du tableau T sont tous positifs *ou* leur somme est positive. »

8.3 Travaux connexes

Nous avons étudié deux catégories de travaux :

- La prise en compte de la symétrie dans la validation par modèle qui nous intéresse par le fait que ce sont des techniques entièrement automatisables ;
- La preuve d'algorithmes itératifs, qui s'intéresse à la preuve de propriété sur des programmes impératifs avec boucle de type while. Ces travaux nous intéressent notamment pour l'étude des preuves d'invariants de boucles.

8.3.1 Prise en compte de la symétrie dans la vérification par modèle

La prise en compte de la symétrie a été proposée dans plusieurs travaux afin de réduire l'explosion d'état dans le processus de vérification par modèle (model-checking). Rappelons (voir section 2.3.2, page 33) que le model-checking de propriétés de sûreté consiste à parcourir tous les états de l'automate représentant le comportement d'un programme afin de déterminer l'atteignabilité de certains états d'erreur.

Les symétries structurelles induisent une relation d'équivalence entre états de l'automate. Il est alors suffisant d'explorer un seul état par classe d'équivalence lors du parcours de l'automate. Par exemple, considérons un algorithme d'exclusion mutuelle pour deux processus A et B. L'état où A est en

section critique et B attend est équivalent à l'état où B est en section critique et A attend. Il n'est donc pas nécessaire d'explorer les 2 états pour prouver l'exclusion mutuelle.

La première utilisation de la notion de symétrie a été proposée pour des réseaux de Petri [HJJJ85]. Les travaux les plus significatifs dans ce domaine ont été ceux de Emerson et al. [ES93, CEJS98, CJEF96] et de Dill et Ip autour du système Murφ [ID93a, ID93b, ID96].

8.3.1.1 Les symétries dans le model-checking de formules CTL

Dans [ES93], Emerson et Sistla montrent comment exploiter la symétrie dans le model-checking de formules temporelles (décrite en CTL*) sur des Systèmes de Transitions Finis (STF) (des *structures de Kripke*). Nous tâchons ci-dessous de présenter rapidement le principe de leur technique.

La notion de symétrie utilise celle de groupe (ensemble munie d'une relation binaire associative possédant un élément neutre et un élément inverse). Étant donné un STF $M = (S, R, L)$ où S est l'ensemble d'états, R est la relation de transition et L est une fonction d'étiquetage des états (par des ensembles de propositions atomiques), un *groupe de symétrie* noté G est un groupe sur S qui préserve la relation R et partitionne S en classes d'équivalence nommées *orbites*. À partir de M, on peut construire un modèle quotient M_G qui contient au moins un représentant de chaque classe d'équivalence. L'espace d'état S_G étant en général plus petit que S, cette méthode permet la vérification de systèmes plus larges.

Les auteurs montrent comment, dans certains cas, la symétrie d'un modèle peut être dérivée de la topologie du graphe associé et exploitent ces notions pour la vérification de systèmes concurrents à variables partagées.

Dans [MHB98], Brayton *et al* étendent ces travaux et montrent comment des symétries dans la formule temporelle (et pas seulement dans le graphe d'état du système) à prouver peuvent être utilisées pour accélérer la vérification.

8.3.1.2 Les symétries dans Murφ

L'approche proposée autour du langage Murφ se base sur cette même notion de symétrie sur l'automate représentant le comportement d'un système, mais en ajoutant un aspect langage avec lequel il est plus facile pour l'utilisateur d'expliciter les symétries éventuelles.

Murφ est un langage (accompagné d'un compilateur et d'un model-checker) dédié à la description de systèmes asynchrones concurrents. Un système est décrit par :

- un ensemble de déclarations de constantes, de types et de variables globales ;
- une collection de *règles de transition* décrivant le comportement du système décrit ;
- une description des états initiaux du comportement ;
- un ensemble d'invariants.

Une règle de transition est un *commande gardée* constituée *d'une condition booléenne portant sur les variables globales ; et d'une action modifiant les valeurs des variables globales*.

Dans [ID93a], Dill et Ip s'intéressent à deux problèmes :

- La détection de symétries structurelles du système à valider ;
- La détection à la volée d'états symétriquement équivalents pendant la vérification, afin de réduire l'ensemble d'états à construire.

Pour répondre au premier point, les auteurs proposent l'extension du langage de spécification en introduisant un nouveau type de données appelé *scalarset*. Un type scalarset représente un ensemble fini de valeurs qui peuvent être échangées dans un état global du comportement sans que la suite de

celui-ci soit modifiée. Un scalarset définit une forme de relation d'équivalence entre toutes les valeurs possibles du type.

Remarque 10 *Les réductions de symétries basées sur l'utilisation de* scalarset *ont aussi été proposées pour l'outil de vérification* UPPAAL *(voir [HBL+04]).*

Les variables déclarées de type « *scalarset* » ne peuvent être manipulées qu'à travers un nombre restreint d'opérateurs qui préservent tous la symétrie :

- L'opérateur logique standard $\exists$:
  Exists ID : scalarsetType **Do** ExpressionBooléenne **EndExists** ;
- L'opérateur $\forall$:
  Forall ID : scalarsetType **Do** ExpressionBooléenne **EndForall** ;
- Un opérateur Ruleset qui permet d'exprimer la répétition de règles de transitions pour la description du système :
  Ruleset ID : scalarsetType **Do** rulesequence **EndRuleset** ;
- Un opérateur itératif de type for :
  For id : scalarsetType **Do** statements **EndFor**
  qui permet d'exprimer, *dans* la définition d'une règle de transitions, des itérations ressemblant à celles que nous définissons en LUSTRE.

Les opérateurs Forall et Exists sont utilisés dans la partie *invariance* du système. For et Ruleset sont utilisés dans la partie *comportement* de la description. Nous donnons maintenant un exemple de programme Murφ décrivant un protocole de cohérence de cache tiré de [ID93a].

EXEMPLE 34 — La figure 8.3 donne le programme Murφ décrivant un protocole de cohérence de cache. Le but de ce programme est d'assurer la cohérence d'une mémoire M à laquelle plusieurs processeur accedent (représentés par un tableau P). On doit s'assurer qu'à tout instant, au plus *un* processeur accède en écriture à un emplacement de mémoire donné.

On commence par définir les variables du système :

- P est un tableau de couples <State,Value> indexé sur pid (les numéros des processeurs) et indiquant pour chaque processeur l'état de ce processeur ;
- M est un tableau de « *pages mémoires* » indexé sur mid (les numéros de cases mémoires). Chaque élément de M est un tableau de cases mémoires indexé sur address. Pour chaque emplacement mémoire on a le triplet d'informations <State,Dir,Mem> représentant l'état de la mémoire à cet emplacement (State), une liste des processeurs utilisant en ce moment cet emplacement (Dir) et la valeur de la mémoire à cet emplacement (Mem).

Remarquons que pid, mid, address et value sont des types « *Scalarset* » : le système décrit (et donc l'automate associé) est symétrique notamment vis-à-vis des processeurs et des espaces mémoires.

On définit ensuite une partie du comportement du protocole à l'aide d'un ensemble Ruleset qui stipule que *étant donné* un processeur n, une adresse mémoire (représentée par h et a) et une valeur à écrire v, si le processeur n est autorisé à écrire en mémoire à l'adresse h,a de la mémoire (la partie utilisée par le processeur est associée à celui-ci par l'information « Cache »), c'est-à-dire si l'état du cache associé est Locally_Exmod alors il y écrit la valeur v.

Enfin, on exprime à l'aide d'un ensemble de Forall une propriété qui doit être vérifiée par le système à tout instant de son exécution. On doit s'assurer qu'étant donnés deux processeurs n1 et n2 différents, l'état d'une case mémoire associée à chacun des deux processeurs (présente dans leur « Cache ») ne peut être Locally_Exmod, c'est-à-dire que les deux processeurs ne peuvent pas être autorisés à écrire dans cette case mémoire en même temps.

— FIN DE L'EXEMPLE 34

L'utilisation dans la syntaxe des programmes de cette notion de scalarset fait apparaître au niveau modèle des *automorphismes de graphes* qui permettent de regrouper les états du système par classes d'équivalence. Sur des exemples de protocoles de cohérence de cache, l'application de cette méthode a permis de réduire de près de 90 % la taille de l'espace d'états (voir les résultats présentés dans [ID96]) mais aussi le temps de vérification (diminution jusqu'à 25%). Une extension de Murφ présentée dans [ID96] propose l'introduction d'un type RepetitiveID qui permet d'expliciter la réplication des composants (forme de map).

Liens avec nos travaux – Cette approche est intéressante puisque d'une part elle prend en compte la symétrie du système à vérifier, mais en plus elle permet au développeur du système de spécifier dans une certaine mesure cette symétrie en offrant des constructions langage spécifiques.

Les formes For, Forall peuvent être comparées à nos itérations qui permettent d'exprimer aussi bien la régularité du système que celle de la propriété à vérifier (voir section 8.2, sur l'expression de propriétés à l'aide d'itérations).

Dans les travaux que nous citons, la prise en compte effective de la symétrie se fait directement sur le modèle de graphe du système, et s'intègre directement à l'outil de vérification. L'approche que nous proposons dans cette thèse est résolument orientée *langage* : nous préférons mettre en place les manipulations liées à la symétrie d'un système directement au niveau langage. Cela présente deux principaux avantages :

- Le premier est un aspect *rendu d'information à l'utilisateur* : il est plus facile pour l'utilisateur de comprendre des manipulations faites par un outil de validation si celles-ci sont effectuées dans le langage que lui-même manipule ;
- Le second concerne la généralité de l'approche : en effectuant les manipulations dans le langage source lui-même on peut plus facilement continuer à utiliser les outils standards de validation (model-checker, mais aussi test, débogage,etc.).

8.3.2 Preuve de programmes itératifs

Dans [Hoa69], Hoare a proposé une approche déductive à la preuve de programme impératifs basée sur ce qui est depuis connue comme la *logique de Hoare*. Étant donnée une propriété R et un programme Q, on s'intéresse à la correction de R par rapport à Q. La correction *partielle* de R par rapport au programme Q signifie que, si celui-ci s'arrête alors R est satisfaite à la fin de l'exécution de Q. La correction *totale* (qui est plus intéressante en général) ajoute une notion de preuve d'arrêt à la correction partielle : elle signifie que l'exécution de Q s'arrête bien et qu'ensuite R est vérifiée.

A chaque programme (ou groupe d'instructions) Q, on associe un couple (P,R) de propriétés. P est une précondition du programme (c'est-à-dire une propriété qui doit être vérifiée avant que Q ne soit exécuté) et R en est la postcondition (évaluée après que le programme a été exécuté). On note en général :

$$P\{Q\}R$$

qui se lit : « si la propriété P est vraie avant l'exécution du programme Q alors Q termine et R est vrai lorsque Q termine son exécution.

Nous allons principalement nous intéresser à l'application de cette approche aux boucles. Cette règle, proposée dans [Hoa69] et largement étudiée dans [BM75], concerne la preuve d'itérations de type « *tant que* » de la forme :

```
Type
      pid : Scalarset(numProcessor);
      mid : Scalarset(numMemory);
      address : Scalarset(numAddress);
      value : Scalarset(valueCount);
Var
    P : Array [pid] of
          Record
             State : enum {Invalid, Shared, Master};
             Value : value;
          End
    M : Array [mid] of
           Array [address] of
             Record
                State : enum {Uncached, Shared_Remote, Dirty_Remote};
                Dir : Array [1..dirsize] of pid;
                Mem : value;
    ...

Ruleset v : value Do
Ruleset h : mid Do
Ruleset n : pid Do
Ruleset a : address Do
   Rule"Modifying value at cache"
        P[n].Cache[h][a].State = Locally_Exmod
   ⇒
   Begin
        P[n].Cache[h][a].Value = v;
   End
Endrulset
Endrulset
Endrulset
Endrulset

Invariant "Ony a single master copy exists"
  Forall n1 : pid Do
  Forall n2 : pid Do
  Forall h : mid Do
  Forall a : address Do
     !(n1¬ =n2
       & P[n1].Cache[h][a].State = Locally_Exmod
       & P[n2].Cache[h][a].State = Locally_Exmod)
Endforall
Endforall
Endforall
Endforall
```

FIG. 8.3 – Un protocole de cohérence de cache en Murφ.

while B **do** S

qui décrit qu'une portion de programme S est répétée jusqu'à ce qu'une condition B soit satisfaite.

Pour prouver :

P{**while** B **do** S}R

il faut :

– trouver un invariant Inv, c'est-à-dire une propriété Inv telle que Inv and B{ S}Inv ;
– prouver que P $\Rightarrow$ Inv et que Inv and not B $\Rightarrow$ R.

La difficulté principale de cette approche est de trouver l'invariant Inv. Dans notre cas, on va essayer de prouver que la propriété R est un invariant de la boucle, c'est-à-dire que R and B{ S} R. On essaiera donc de prouver que :

- R est vraie avant l'itération ;
- et que R est vrai après un passage dans le corps de la boucle (dans un cas où la condition B est vraie à l'entrée du corps de boucle).

Ces deux conditions ne sont pas nécessaires. Par conséquent, si l'on n'arrive pas à les prouver, on ne peut pas en conclure que R est fausse.

Cette preuve de boucle while représente une correction seulement partielle, au sens de ce que nous avons présenté plus haut. Si l'on souhaite une correction totale, on doit encore prouver l'arrêt de la boucle. Dans notre cas, l'arrêt est assuré par la construction même des itérations puisque ce sont des boucles de type for sur des tableaux de taille statiquement connue.

8.3.2.1 Vérification d'architectures régulières en ALPHA

Dans un domaine plus proche du notre [DQ94] et plus récemment [MA04] proposent une méthode de vérification pour des architectures régulières décrite en Alpha. Une partie de cette méthode consiste à simplifier l'implémentation du système à vérifier en appliquant une règle d'induction sur des structures à une dimension. Cette règle d'induction s'applique aux variables flot et permet aux auteurs de prouver l'invariance d'une propriété dans le temps et est similaire à celle proposée dans [DC00]. Cette règle, qui ressemble à celle que nous proposons au paragraphe 8.5, n'est pas utilisée pour traiter les autres dimensions (spatiales) du programme.

8.4 Extension de la sémantique LUSTRE en PVS avec les opérateurs tableaux

Nous avons présenté au chapitre 2 (page 35) un encodage de la sémantique de LUSTRE en PVS. Nous proposons maintenant une extension de cette sémantique incluant les itérateurs de tableaux. Nous montrons aussi la preuve d'un exemple simple à l'aide de PVS. Cette approche s'est avérée insatisfaisante pour des raisons que nous évoquerons plus loin.

8.4.1 Tableaux et itérateurs

8.4.1.1 Types tableaux

Les objets que nous manipulons sont des flots de valeurs codés en PVS par des sequences (voir paragraphe 2.3.3). Nous utiliserons par la suite la notation suivante :

```
LInt : TYPE = sequence[int]
LBool : TYPE = sequence[bool]
```

LInt et LBool représentent donc respectivement les types *flot d'entiers* et *flot de booléens*.

Les tableaux sont représentés à l'aide du type PVS ARRAY. On va tout d'abord définir les indices des tableaux. Pour cela, on utilise :

```
Taille : posnat
```

pour décrire la taille des tableaux (Taille est un entier strictement positif, ce qui représente le fait qu'on ne manipule pas de tableaux vides). On définit un type IDX qui sera utilisé pour les indices des tableaux. Les éléments de IDX sont donc des entiers positifs ou nuls et inférieurs strictement à Taille (comme en LUSTRE les tableaux sont indexés entre 0 et Taille-1). On a :

```
IDX : TYPE={k:nonneg_int|k<Taille}
```

Les tableaux d'entiers sont donc des fonctions de IDX vers LInt. Chaque élément d'un tableau d'entiers est donc un flot d'entiers. On définit pour cela le type LTabInt :

```
LTabInt : TYPE = ARRAY[IDX -> LInt]
```

De la même façon, on définit le type des tableaux d'éléments booléens :

```
LTabBool : TYPE = ARRAY[IDX -> LBool]
```

8.4.1.2 Définitions des itérateurs dans la sémantique PVS de LUSTRE

Nous présentons maintenant un encodage possible des itérateurs LUSTRE en PVS. On étudie dans la suite seulement les itérations red et map, l'encodage du fill et du map_red sont similaires. L'encodage que nous avons choisi n'est pas totalement générique : on va devoir décrire une forme d'itération pour chaque signature de nœud itéré. En effet, tous les opérateurs sont paramétrés par les tableaux paramètres de l'itération et la fonction itérée elle-même. Cet inconvénient disparaît lorsqu'on considère qu'on pourrait à terme compiler un programme LUSTRE automatiquement en PVS suivant le format présenté ici.

Il est aussi important de remarquer que l'encodage choisi n'est pas le seul possible. Il n'est pas forcément le plus élégant, et le lecteur pourra remettre en cause certains points. Mais la simplicité de la syntaxe utilisée présente au moins l'avantage de permettre des preuves assez faciles.

red – Globalement, on représente les itérations avec des fonctions récursives paramétrées par les accumulateurs d'entrées, les tableaux d'entrées, le nœud itéré, représenté par une fonction sur les flots.

On peut par exemple décrire une réduction d'un nœud N_reduced qui prend 2 paramètres (un accumulateur booléen et un élément de tableau entier) et rend un résultat (un accumulateur booléen). Pour cela, on va écrire une fonction récursive Lred qui en fait peut être utilisée pour itérer tout nœud ayant cette signature.

```
LRed(Init : LBool, Index : IDX , T : LTabInt,
       N_reduced : FUNCTION[LBool,LInt -> LBool]) : RECURSIVE LBool =
IF(Index=0)
THEN N_reduced(Init,T(0))
ELSE N_reduced(LRed(Init,Index-1,T,N_reduced), T(Index))
ENDIF
MEASURE Index
```

Cette réduction prend les paramètres suivants :

- Init est l'expression d'initialisation de l'itération ;
- T est le tableau d'entrée de la réduction ;
- N_reduced est le nœud itéré ;
- I est l'indice de l'étage courant de l'itération ;

Le résultat de la réduction dépend de l'indice couramment inspecté dans l'itération : si on en est à l'étage initial (premier élément des tableaux, représenté par l'indice 0, on applique simplement le nœud itéré à la valeur initiale de l'accumulateur et aux premiers éléments des tableaux. Sinon, on applique le nœud itéré à l'élément courant des tableaux T(Index) et à la valeur rendue par l'itération jusqu'à présent. Cette valeur est calculée en appliquant la fonction LRed aux paramètres suivants :

- Init n'est pas modifié ;
- I a été décrémenté de 1 (on doit calculer l'étage précédent dans l'itération) ;
- le tableau d'entrée T et le nœud itéré _reduced ne sont pas modifiés.

Le lecteur aura noté la présence d'une ligne particulière :

```
MEASURE Index
```

En PVS, pour chaque fonction récursive f définie, on donne au système une *mesure* entière (fonction portant sur les paramètres de la fonction récursive) qui doit décroître à chaque appel de f. Cette fonction permet au système de s'assurer que la récurrence s'arrête toujours. Ici, on garantit qu'à chaque appel la valeur du paramètres Index diminue. C'est bien le cas puisqu'à chaque appel la nouvelle valeur de Index est décrémentée et que la récursion s'arrête lorsque Index=0.

EXEMPLE 35 — Considérons une variante du nœud unPositif défini au paragraphe 8.2.2.1 (on s'est contenté de traduire le nœud, initialement utilisé avec un forall pour l'utiliser avec un red) :

```
node unPositif(acc : bool; elt : int) returns (acc_out : bool);
let
   acc_out = acc and elt > 0;
tel
```

Comme nous l'avons vu au chapitre 1, ce programme peut être décrit en PVS, par :

```
unPositif_type(acc : LBool, elt : LInt) : TYPE = { out : LBool |
          out = (acc AND Ltrue   (pre(elt)>=0))}

unPositif_axiom : AXIOM FORALL (acc : LBool, elt : LInt) :
          EXISTS (out : LBool) : TRUE

unPositif_node(acc : LBool, elt : LInt) : allPositive_type(acc,elt)
```

On souhaite maintenant décrire en PVS la réduction suivante :

$$\text{red}\ll\text{unPositif;Taille}>>(\text{true,T1})$$

On écrit pour cela l'expression suivante :

$$\forall \text{Index, T1. red(LTrue,Index,T1,unPositif\_node)}$$

où LTrue représente un flot booléen constant true de LUSTRE.

— FIN DE L'EXEMPLE 35

map – La définition de red que nous venons de donner est assez proche de la définition en LUSTRE. Celle du map est par contre un peu différente. Afin de faciliter les preuves, nous avons trouvé plus pratique de définir le map comme une réduction particulière. Considérons par exemple que l'on veuille itérer un nœud N qui prend un entier en entrée et rend aussi un entier en sortie. En LUSTRE, N a pour signature :

node N(elt1 : int) **returns** (elt2 : int);

À partir de cette définition, on va construire un nœud N_mapped qui a la forme suivante :

```
node N_mapped(acc_in : bool; elt1, elt2 : int) returns (acc_out : bool);
let
   acc_out = acc_in and elt2 = N(elt1);
tel
```

En fait, nous venons simplement de transformer N en un observateur N_mapped qui accepte tous les couples de valeurs (a,b) tels que b=N(a). En accord avec ce qui a été montré au chapitre 3, ces deux versions décrivent les même comportements en terme d'ensembles de traces. L'expression map≪Taille;N≫(T1) se transforme alors en map≪Taille;N≫(true,T1,T2).

En PVS, N est logiquement de type :

N : FUNCTION[LInt -> LInt]

Le nœud N_mapped est donc de la forme :

N_mapped : FUNCTION[LBool, LInt, LInt -> LBool]

et est tel que :

N_mapped(acc,a,b) = acc and (b=N(a))

Cette notation peut paraître étrange, mais elle permet d'appliquer le même schéma de preuve par induction pour la réduction qui représente la propriété à prouver et pour un éventuel map qui représente un calcul sur les paramètres de la propriété à prouver (type cascade d'itérations). En effet, une fois qu'on a définit N_mapped, la définition de map≪N;taille≫ est similaire à celle de red :

```
LMap(Init : LBool, Index : IDX , T1 : LTabInt, T2 : LTabInt,
       N_mapped : FUNCTION[LBool,LInt,LInt -> LBool]) : RECURSIVE LBool =
IF(Index=0)
THEN N_mapped(Init,T1(0),T2(0))
ELSE N_mapped(LMap(Init,Index-1,T1,T2,N_mapped), T1(Index), T2(Index))
ENDIF
MEASURE Index
```

EXEMPLE 36 — Soit le nœud prePlus suivant. Il prend un entier elt1 en entrée et définit la valeur de sa sortie elt2 à l'instant t comme étant égale à la somme de la valeur de l'entrée elt1 à l'instant t et à l'instant t-1. À l'instant initial, elt2 vaut la valeur courante de elt1. On a :

```
node prePlus(elt1 : int) returns (elt2 : int);
let
   elt2 = elt1 -> pre(elt1) + elt1;
tel
```

Afin d'utiliser notre définition de map sous forme de réduction, on transforme ce nœud prePlus en l'observateur prePlus_mapped défini par :

```
node prePlus_mapped(acc_in : bool; elt1,elt2 : int) returns (acc_out : bool);
let
   acc_out = acc_in and (elt2 = elt1 -> pre(elt1) + elt1);
tel
```

L'encodage PVS correspondant à ce nœud est alors simplement :

```
prePlus_mapped_type(acc : LBool, elt1, elt2 : LInt) : TYPE = { out : LBool |
        out = acc AND elt2 = elt1   pre(elt1) + elt1}
prePlus_mapped_axiom : AXIOM FORALL (acc : LBool, elt1, elt2 : LInt) :
        EXISTS (out : LBool) : TRUE
prePlus_mapped_node(acc : LBool, elt1, elt2 : LInt) : plus_type(acc, elt1, elt2)
```

On peut maintenant définir en PVS l'expression représentant l'itération T2 = map≪prePlus;Taille≫(T1). On a simplement :

∀ Index, T1, T2. LMap(LTrue,Index,T1,T2,prePlus_mapped_node)

— Fin de l'exemple 36

8.4.2 Stratégie de preuve

Une fois le programme Lustre traduit en PVS, on décrit la propriété à prouver comme dans les exemples précédents. On va ensuite prouver cette propriété par *induction sur l'indice* Index.

Avec les programmes que nous avons utilisés plus haut, nous pouvons exprimer la propriété suivante : *si un tableau* T1 *a tous ses éléments positifs, alors le tableau* T2 *, tel que* T2[i] = T1[i] -> pre(T1[i]) + T1[i]) *a lui aussi ses éléments positifs.* En Lustre, cette propriété est décrite par l'observateur suivant :

```
node (T1 : intˆTaille) returns (ok : bool);
var T2 : intˆTaille;
let
   T2 = map≪prePlus;Taille≫(T1);
   ok = red≪unPositif;Taille≫(true,T1)
        => red≪unPositif;Taille≫(true,T2);
tel
```

où encore (pour retrouver un map exprimé par une réduction) :

```
node (T1,T2 : intˆTaille) returns (ok : bool);
var res : bool
let
   res = red≪prePlus_mapped;Taille≫(true,T1,T2);
   ok = red≪unPositif;Taille≫(true,T1) and res
        => red≪unPositif;Taille≫(true,T2);
tel
```

Suivant les explications données au paragraphe précédent, l'encodage PVS de cette propriété est simplement :

```
P : ∀ (Index : IDX, T1 : LTabInt, T2 : LTabInt).
      A(LRed(LTrue,Index,T1,unPositif_node)
      AND LMap(LTrue,Index,T1,T2,prePlus_node)
      ⇒ LRed(LTrue,Index,T2,unPositif_node))
```

Rappelons (voir 2.3.3) que la fonction A représente la vérité au cours du temps de la propriété qu'elle reçoit comme argument. Par exemple A(P) signifie « P *est toujours vraie* » La propriété P est trivialement vraie, mais représente un exemple assez simple sur lequel nous allons maintenant montrer la démarche à suivre dans PVS pour le prouver.

La première partie de la preuve consiste à appliquer un schéma d'induction sur l'entier Index qui représente l'étage actuel de l'itération. On va alors devoir prouver deux propriétés :

Cas de base – la propriété représentée par l'accumulateur de la réduction est vrai après le premier passage dans l'itération, cas qui est représenté en PVS par la propriété :

```
∀ T1: LTabInt, T2: LTabInt.
  A(LRed(LTrue, 0, T1, unPositif_node) AND
  LMap(LTrue, 0, T1, T2, plus_node)
  ⇒ LRed(LTrue, 0, T2, unPositif_node))
```

Le lecteur aura remarqué qu'on considère comme cas de base, non pas la valeur de l'accumulateur de la réduction avant le premier passage dans l'itération (comme c'est le cas habituellement), mais après ce premier passage. Nous reviendrons sur ce choix au paragraphe 8.5.2.

Invariance – la propriété est préservée par un passage dans l'itération : si elle est vraie à un rang j, alors elle est vraie au rang j+1. En PVS, on doit donc prouver :

```
∀ j < Taille
  (j < Taille - 1 AND
  (∀ T1: LTabInt, T2: LTabInt.
     A(LRed(LTrue, j, T1, unPositif_node) AND
     LMap(LTrue, j, T1, T2, plus_node)
⇒ LRed(LTrue, j, T2, unPositif_node))))
⊨
  (∀ T1: LTabInt, T2: LTabInt.
     A(LRed(LTrue, j + 1, T1, unPositif_node) AND
     LMap(LTrue, j + 1, T1, T2, plus_node)
     ⇒ LRed(LTrue, j + 1, T2, unPositif_node)))
```

L'induction que nous avons mise en place ci-dessus concerne les tableaux. Dans le traitement du cas de base de cette induction (comme dans le traitement du cas général, d'ailleurs), on doit maintenant montrer un invariant du temps sur un étage de l'itération. Cette invariance par rapport au temps est exprimée dans les propriétés ci-dessus par l'opérateur A.

8.4.2.1 Induction sur les tableaux - Cas de base

Pour prouver le cas de base de l'induction sur les tableau, on doit donc prouver (pour obtenir cette propriété, on a simplement expansé la définition de A) :

```
∀ n: nat.
  (LRed(LTrue, 0, T1, unPositif_node) AND
   LMap(LTrue, 0, T1, T2, prePlus_node)
   ⊨ LRed(LTrue, 0, T2, unPositif_node))(n)
```

C'est-à-dire que le cas de base de l'itération est vrai à tout instant de l'exécution du programme. On va pour cela utiliser une induction sur n. On va ainsi prouver que la propriété est vraie à l'instant initial, puis qu'elle est préservée dans le temps : si elle est vraie à un instant t alors elle est vraie à l'instant j+1.

Induction temporelle - Cas de base – On doit donc montrer la propriété suivante :

```
LRed(LTrue, 0, T1, unPositif_node)(0)
AND LMap(LTrue, 0, T1, T2, plus_node)(0)
⊨ LRed(LTrue, 0, T2, unPositif_node)(0)
```

Par définition des itérateurs LRed et LMap on obtient aisément :

```
unPositif_node(LTrue, T1[0])(0)
AND plus_node(LTrue, T1[0], T2[0])(0)
⊨ unPositif_node(LTrue, T2[0])(0)
```

qui est facilement prouvé en utilisant les définitions de plus_node et de unPositif

Induction temporelle - Invariance – On doit montrer que la propriété qui porte sur le premier élément des tableaux est préservée dans le temps, c'est-à-dire que quel que soit l'instant j, si la propriété est vraie en j alors elle est vraie en j+1 ce qui est représenté par :

```
∀ j.
  (LRed(LTrue, 0, T1, unPositif_node) AND
   LMap(LTrue, 0, T1, T2, prePlus_node)
  ⇒ LRed(LTrue, 0, T2, unPositif_node))(j)
⊨
  (LRed(LTrue, 0, T1, unPositif_node) AND
   LMap(LTrue, 0, T1, T2, prePlus_node)
  ⇒ LRed(LTrue, 0, T2, unPositif_node))(j + 1)
```

Comme on est toujours en train de traiter le cas de base de l'itération tableau, les instances de LRed et de LMap sont remplacés par des instances de unPositif_node et plus_node (par définition). Ce cas se prouve alors facilement.

8.4.2.2 Induction sur les tableaux - Invariance

On a montré ci-dessus que la propriété était vraie après un premier passage dans l'itération. On va montrer maintenant que cette propriété est préservée par un passage dans l'itération. On applique les mêmes transformations syntaxiques et on applique une induction temporelle qui nous donne les deux cas suivants à montrer.

Induction temporelle - Cas de base – On doit prouver la propriété suivante :

```
(LRed(LTrue, j, T1, unPositif_node) AND
 LMap(LTrue, j, T1, T2, prePlus_node)
 ⇒ LRed(LTrue, j, T2, unPositif_node))(0)
⊨
(LRed(LTrue, j+1, T1, unPositif_node) AND
 LMap(LTrue, j+1, T1, T2, prePlus_node)
 ⇒ LRed(LTrue, j+1, T2, unPositif_node))(0)
```

On expanse ensuite une fois la définition de LRed et de LMap dans le conséquent de cette implication. On obtient la propriété :

```
(LRed(LTrue, j, T1, unPositif_node) AND
 LMap(LTrue, j, T1, T2, prePlus_node)
 ⇒ LRed(LTrue, j, T2, unPositif_node)(0))
⊨
(unPositif_node(LRed(LTrue, j+1, T1, unPositif_node)) AND
 prePlus_node(LMap(LTrue, j+1, T1, T2, prePlus_node))
 ⇒ unPositif_node(LRed(LTrue, j+1, T2, unPositif_node)))(0)
```

On applique ensuite :

- Un renommage des différentes partie de l'antécédent : LRed(LTrue, j, T1, unPositif_node) en H1, LMap(LTrue, j, T1, T2, prePlus_node) en H2 et LRed(LTrue, j, T2, unPositif_node) en H3 ;
- Une mise à plat de l'implication de la partie conséquence de la propriété.

On obtient ainsi :

```
H1(0) AND H2(0) ⇒ H3(0)
unPositif_node(H1, T1(j+1))(0)
prePlus_node(H2, T1(j+1), T2(j+1))(0)
⊨
unPositif_node(H3, T2(j+1))(0)
```

En remplaçant plus_node et unPositif_node par leur définition, on prouve facilement cette propriété.

Induction temporelle - Invariance – La dernière propriété qui doit être prouvée est l'invariance temporelle du cas général de l'induction sur les tableaux, c'est-à-dire :

```
∀n.
 (LRed(LTrue, j, T1, unPositif_node) AND
 LMap(LTrue, j, T1, T2, prePlus_node)
 ⇒ LRed(LTrue, j, T2, unPositif_node))(n)
⊨
 (LRed(LTrue, j+1, T1, unPositif_node) AND
  LMap(LTrue, j+1, T1, T2, prePlus_node)
  ⇒ LRed(LTrue, j+1, T2, unPositif_node))(n+1)
```

On applique les même transformations syntaxiques que précédement. On obtient ainsi un but facilement prouvé par les stratégies de base de PVS.

8.4.3 Conclusion

Dans le paragraphe précédent, nous avons montré la démarche de preuve à mettre en place dans PVS sur un exemple. Cette démarche est toujours identique quel que soit le programme à traiter. On va tenter de prouver la propriété itérative concernée en appliquant une induction sur l'indice des tableaux. On obtient ainsi un cas pour le premier niveau de l'itération et un cas pour l'invariance de la propriété par application du nœud itéré. Ensuite, on prouve chacun de ces cas par induction temporelle sur les flots LUSTRE. Ce schéma de preuve est donné à la figure 8.4.

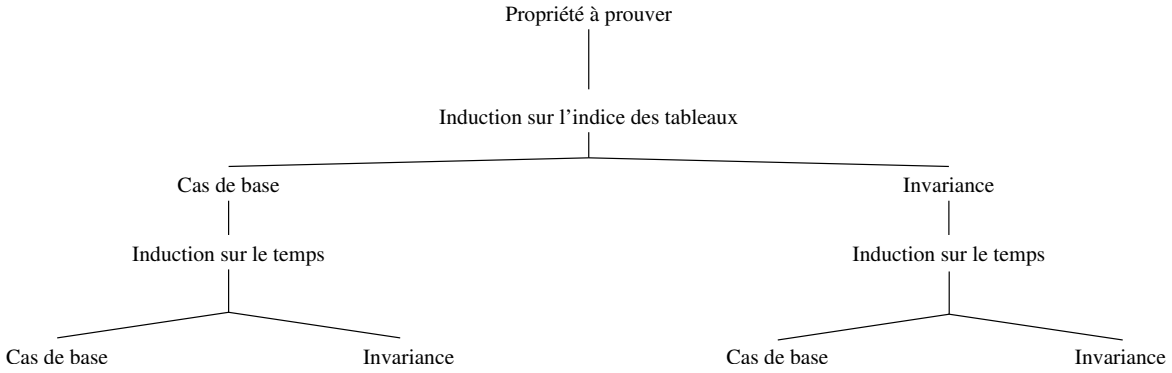


FIG. 8.4 – Schéma de preuve de programmes itératifs dans PVS.

La motivation première pour utiliser un outil tel que PVS était la puissance du moteur d'inférence. La thèse de Cécile Dumas-Canovas [DC00] avait déjà fait ressortir la lourdeur d'utilisation de ces outils et la difficulté représentée par l'obligation d'encoder toute la sémantique du langage LUSTRE en PVS : « l'utilisation de fonctions telles que *count* ou *index* nous éloigne du formalisme initial LUSTRE ». Il serait ainsi difficile pour un utilisateur ne connaissant pas cette sémantique d'interpréter une preuve qui n'aboutit pas.

Nous avons tout de même voulu étudier cette approche pour la preuve de programmes itératifs. Cela nous a fourni un premier terrain d'expérimentation pour découvrir les méthodes à mettre en place.

L'encodage que nous avons proposé n'est pas optimal. Il est certainement possible, pour quelqu'un connaissant mieux les possibilités de PVS de proposer un encodage qui facilite les preuves. Par ailleurs, il paraît indispensable de proposer une compilation de LUSTRE en PVS. Cette compilation doit être facile à réaliser.

Un des problèmes qui se pose est l'automatisation des preuves. Il est hors de question de demander à un utilisateur de faire « à la main » tout le raisonnement qui a été montré ci-dessus, celui-ci demandant

une certaine maîtrise de PVS. Cette automatisation pourrait se faire par l'écriture de stratégies PVS. Nous n'avons pas étudié cette possibilité.

Comme déjà souligné dans [DC00], il se peut que d'autres outils de déduction (tels que Coq ou Isabelle) soit plus adaptés à la description des programmes LUSTRE. Mais dans tous les cas, une telle approche demande des efforts importants, tant pour la traduction depuis LUSTRE vers le langage de l'outil considéré que dans la détermination de stratégies automatiques pour l'outil lui-même.

Finalement, les manipulations que nous souhaitons considérer sont essentiellement syntaxiques. Nous avons trouvé plus simple et plus efficace de développer une sorte de préprocesseur de programmes itératifs qui opère des manipulations directement sur le programme LUSTRE. On peut alors utiliser les outils de validation dédiés à LUSTRE sur les obligations de preuves générées par un tel préprocesseur. C'est cette approche qui est décrite dans le paragraphe suivant.

8.5 Manipulation de programmes itératifs LUSTRE

8.5.1 Introduction

On considère (voir figure 8.5) un programme P utilisant des itérations et une propriété également exprimée avec des itérations. D'autres composants sont éventuellement utilisés dans P, de préférence donnés avec une spécification locale (un contrat). Notre but est de prouver φ sur P.

Dans cette section nous montrons comment on peut exploiter la structure de φ et de P pour obtenir plusieurs sous-objectifs de preuve plus simples à prouver que φ elle-même.

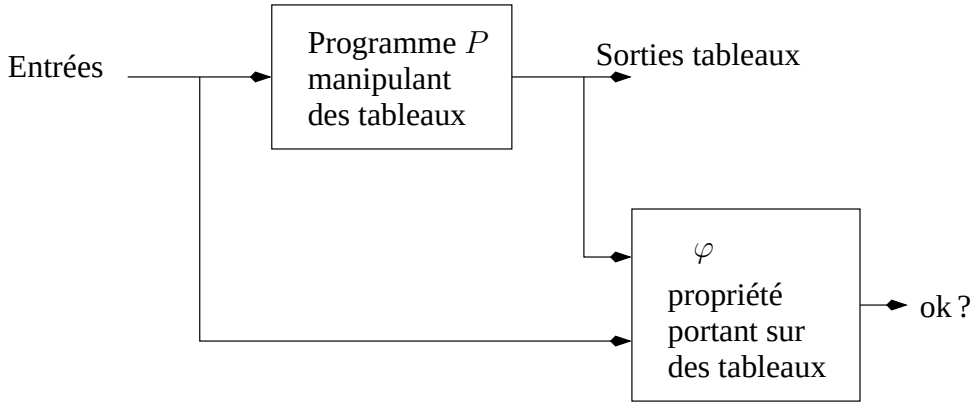


FIG. 8.5 – Position du problème.

D'un point de vue pratique, on va considérer que φ est intégré à P (voir figure 8.6). On se ramène donc à considérer une « boîte » de laquelle sort une valeur booléenne. Le principe du traitement proposé est de regarder comment cette valeur booléenne est calculée pour essayer de découper (« slicer ») la preuve selon la structure en tableaux. L'objectif étant de perdre des dimensions et donc obtenir des preuves plus simples.

Exemple trivial – Considérons une première propriété exprimée à l'aide de l'opérateur forall du paragraphe 8.2.1. Admettons que cette propriété porte directement sur des entrées du programme considéré (voir figure 8.7). On peut découper cette propriété en autant de cas qu'il y a d'éléments dans le tableaux. De plus, comme chaque élément est calculé de la même façon, il n'est pas nécessaire

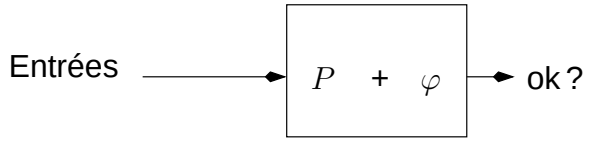


FIG. 8.6 – φ est intégré à P.

de prouver tous ces cas. Il suffit donc de prouver que la propriété itérée dans le forall est vraie pour un élément quelconque du tableau.

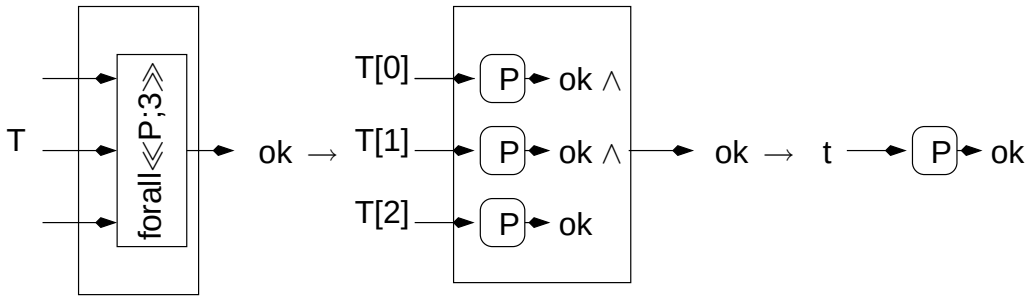


FIG. 8.7 – La propriété est exprimée avec un forall.

Cette première technique, permise par la symétrie de l'opérateur forall, est très intéressante puisqu'on ne perd aucun information : on n'a pas renforcé la propriété initiale.

Considérer que la propriété porte directement sur les entrées du programme n'est pas très intéressant. Mais cette technique s'étend parfaitement au cas où le tableau sur lequel porte la propriété est calculé par un enchaînement de maps (figure 8.8). Notons que, bien qu'il soit extrêment simple, nous avons rencontré ce cas dans de vraies applications (notamment sur le ELMU, présenté au chapitre 7). Il sera présenté plus en détail au paragraphe 8.5.2.1.

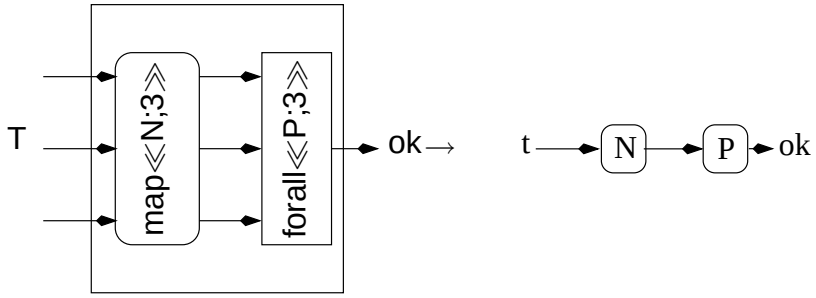


FIG. 8.8 – Traitement d'un enchaînement « map suivi de forall »

Dans la pratique, les propriétés qu'on aura à traiter seront plus complexes. En général elles seront exprimées à l'aide de réductions quelconques (et pas forcément par des forall). Elles pourront mettre en jeu des enchaînements d'itérations plus complexes que les maps que nous avons évoqués. De manière générale, toute la puissance du langage LUSTRE peut être utilisée pour exprimer aussi bien les propriétés que les programmes. Nous donnons ci-dessous les sources de compléxités.

La propriété est calculée par un red **quelconque** – En général, la propriété n'est pas forcément exprimée par un forall mais par une itération red. Nous devons alors traiter une propriété de la forme de celle présentée au paragraphe 8.3.2 dans un cas où nous ne connaissons pas l'invariant.

Nous proposons de « tenter » de prouver cette propriété en supposant qu'elle est un invariant des itérations du programme. On va donc essayer de prouver cette propriété par induction sur les tableaux manipulés. On génère deux objectifs de preuve, le premier permettant de prouver le cas de base (on va prouver que la propriété est vraie au début de l'itération) et le second permettant de prouver l'induction (on prouve que la propriété est préservée par un passage dans la boucle de l'itération).

En procédant de la sorte, on renforce bien sûr la propriété. Dans le cas classique (preuve d'algorithmes itératifs du paragraphe 8.3.2) cette technique est trop forte, mais nos itérateurs servent en général à structurer les programmes et cette approche nous paraît suffisante dans ce contexte. Nous étudierons cet aspect au paragraphe 8.5.2.2.

Les tableaux paramètres de la réduction peuvent aussi être des tableaux de constantes (de type [1,2,3,4]) ou des tableaux explicites (3^t où t est une taille de tableau). Nous montrerons comment tirer partie de cette information pour préciser un peu les objectifs de preuve générés.

Notons que la technique que nous proposons n'a d'intérêt que dans le cas de tableaux à plus d'un élément.

La forme du réseau d'opérateurs est complexe – Dans le cas du forall, nous avons indiqué que les tableaux sur lesquels porte la propriété peuvent être des variables locales calculées par des map. Dans ce cas, la méthode de slice proposée peut être propagée à l'enchaînement de maps. Dans le cas de propriétés exprimées par un red, on peut aussi propager la transformation aux itérations utilisées en amont dans le programme. Pour chaque itération, on complétera les deux objectifs pour le cas de base et pour l'induction. Nous détaillerons ce cas aux paragraphes 8.5.3 et 8.5.4.

Les calculs dans le réseaux sont complexes – Le programme sur lequel porte la propriété peut aussi être plus complexe par la présence de nœuds appelés. L'expansion systématique des appels de noeuds est toujours envisageable. On se ramène alors au cas précédent. Dans le cas où ces noeuds sont décrits à l'aide des contrats, on peut éviter l'expansion en utilisant les contrats. Lorsqu'un noeud est utilisé, on va renforcer les hypothèses de la propriété à prouver avec son contrat et appliquer la méthode précédente sur ce nouveau programme. Cette technique est décrite au paragraphe 8.5.5.

8.5.2 Traitement de propriétés itératives simples

Nous présentons dans ce paragraphe les cas les plus simples de propriétés itératives qu'on peut rencontrer. Dans un premier temps, nous détaillons l'exploitation de la forme forall et donnons un exemple d'application. Ensuite, nous détaillons le cas d'une propriété exprimée par un red. Nous expliquons le principe du renforcement de la propriété à prouver en un invariant et montrons la construction de l'objectif de preuve découpé en deux parties : une première permet de prouver l'initialisation de l'invariance et le second permet de prouver l'induction.

8.5.2.1 Cas du forall

Considérons l'observateur de la figure 8.9. Il exprime la propriété suivante : « *Si tous les éléments du tableau* T1 *sont positifs, et que* T2 *est défini par un map d'un nœud* N*, alors est-ce que les éléments de* T2 *sont aussi tous positifs* ». Pour prouver cette propriété, il suffit de prouver la propriété donnée à

la figure 8.10 qui exprimer que *si un élément quelconque de* T1 *est positif alors un élément de* T2 *est positif*.

```
node obs(T1 : intˆ10) returns (ok : bool);
var T2 : intˆ10;
let
   T2 = map≪N;10≫(T1);
   ok = forall≪unPositif;10≫(T2);
   assert forall≪unPositif;10≫(T1);
tel
```

FIG. 8.9 – Propriété purement symétrique.

```
node obs_bis(elt_T1 : int) returns (ok : bool);
var elt_T2 : int;
let
   elt_T2 = N(elt_T1);
   ok = unPositif(elt_T2);
   assert unPositif(elt_T1);
tel
```

FIG. 8.10 – Obligation de preuve pour la propriété de la fig. 8.9.

8.5.2.2 Principe du traitement pour le red

Nous nous intéressons maintenant à la première forme de complexité que nous avons évoqué plus haut. On considère que la propriété est calculée par un red, ou qu'elle porte sur le résultat d'une réduction. Considérons par exemple l'observateur donné à la figure 8.11. Le noeud phi code la propriété *la valeur de* v *est positive*. La propriété encodée par le noeud Obs est que la somme des éléments d'un tableau T est positive.

Afin de mieux comprendre les renforcements que nous proposons, nous donnons à la figure 8.12 une version impérative du calcul effectué par ce programme pendant un cycle d'exécution. Il s'agit d'une boucle de type for obtenue par compilation de l'itération red comme indiqué au paragraphe 4.3.2.2.

Sur le côté de ce programme sont notés 4 points de repère : α, β, γ et γ'. La propriété ϕ doit être vérifiée au point α dans le programme impératif. Comme nous l'avons vu plus haut, pour prouver ϕ en α, il *suffit* de prouver que ϕ est un invariant de la boucle (c'est une condition suffisante, mais pas nécessaire).

Pour cela, nous pouvons montrer un cas de base stipulant que ϕ est vrai au point β (aucun passage dans la boucle n'a été effectué). Puis, nous montrons que ϕ est conservé par un passage de la boucle : quelle que soit la valeur de i, si ϕ est vrai en γ, alors ϕ est aussi vrai en γ'.

Spécificité du cas de base considéré – En fait, nous ne considérons pas le cas de base de la preuve par induction au point de repère β (avant une première exécution du corps de boucle), mais au point

```
node Obs(T : int^s) returns (ok : bool);
var n : int;
let
   n = red≪plus;s≫(0,T);
   ok = phi(n);
tel

node phi(v : int) returns (Pv : bool);
let
   Pv = v>0;
tel
```

FIG. 8.11 – Une propriété itérative simple.

```
Acc = 0;                          | β
for(int i=0;i<s;i++){             |
                                  | γ
   Accu = plus(Accu,T[i]);        |
                                  | γ'
}                                 |
n = Accu;                         | α
```

FIG. 8.12 – Le calcul de la variable n.

γ' (c'est-à-dire après une première exécution du corps de boucle). Ce relâchement s'appuie essentiellement sur la volonté de ne pas rejeter des propriétés qui sont fausses avant le premier passage de la boucle, mais qui sont pourtant des invariants de la boucle. Il est également possible grâce au fait que *nous ne considérons pas d'itérations sur des tableaux vides*.

Par exemple, considérons l'itération de la figure 8.11. Sous l'hypothèse que *les éléments de* T *sont tous positifs*, ϕ est satisfaite. La règle de preuve que nous proposons mène au cas de base et à l'induction que voici :

Cas de base : à l'initialisation (en β), l'accumulateur Acc de l'itération est positif.

Induction : si Acc est positif en γ, alors Acc est positif en γ'.

On prouve sans problème le cas inductif (car chaque élément de T est supposé strictement positif et que Acc croît à chaque application de plus). Mais le cas de base considéré est trivialement faux car Acc est initialisé à 0.

Nous relâchons volontairement la contrainte sur la cas de base et exigeons maintenant de le considérer en γ' après exactement *un* passage dans le corps de boucle. Dans ce cas, la propriété est satisfaite (une application de plus produit bien une valeur de Acc strictement positive, si l'élément de T considéré est positif comme le stipule l'hypothèse prise plus haut).

8.5.2.3 Règle de manipulation associée

La propriété considérée dans l'exemple ainsi que les repères α et β sont représentés graphiquement à la figure 8.13. Cette figure indique aussi les endroits dans l'itération que l'on considère pour le cas initial et pour l'invariance de la propriété.

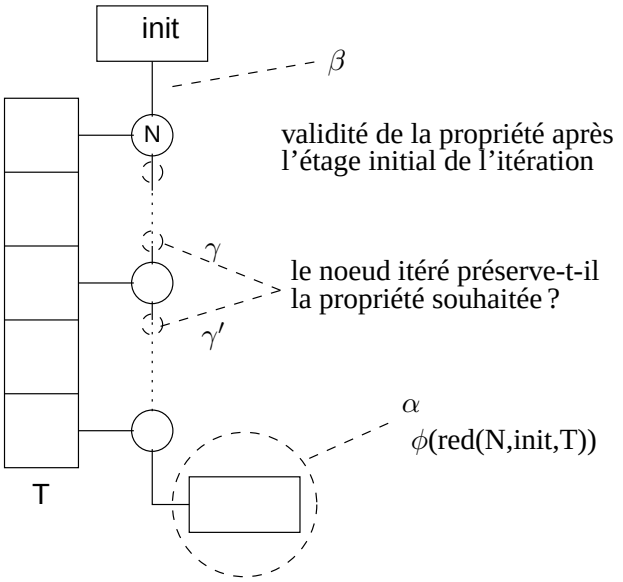


FIG. 8.13 – Propriété itérative et manipulations.

Pour prouver la propriété, on va générer une conjonction de propriétés LUSTRE. La première permet de prouver la validité de la propriété à l'étage initial de l'itération (voir figure 8.14). La seconde, figure 8.15, permet de prouver que la propriété est conservée par un appel au nœud itéré.

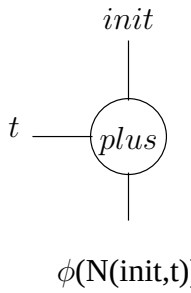


FIG. 8.14 – Cas Initial.

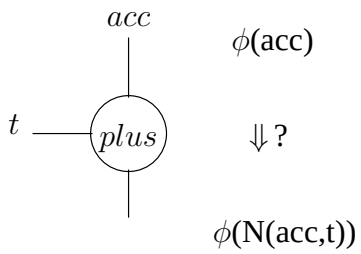


FIG. 8.15 – Invariance.

Cette transformation est représentée par la règle suivante : Étant donnée la propriété ϕ sur une réduction d'un nœud quelconque N, représentée par l'expression suivante (voir figure 8.13) :

$$\lambda \mathsf{init}, \mathsf{T}.\phi(\mathsf{red}(\mathsf{init}, \mathsf{T}, \mathsf{N}))$$

où

- init est l'entrée d'initialisation de la réduction ;
- T est l'entrée tableau ;

$$\frac{\begin{array}{c}\lambda \mathsf{init}, \mathsf{t}.\phi(\mathsf{N}(\mathsf{init}, \mathsf{t})) \\ \lambda \mathsf{acc}, \mathsf{t}.\phi(\mathsf{acc}) \Rightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{t}))\end{array}}{\lambda \mathsf{init}, \mathsf{T}.\phi(\mathsf{red}(\mathsf{N}, \mathsf{T}, \mathsf{init}))}$$

FIG. 8.16 – Règle de construction d'obligations de preuves pour observateurs itératifs simples.

on génère deux observateurs. Le premier correspond à la figure 8.14 et permet de prouver que ϕ est vraie après un passage dans l'itération :

$$\lambda \mathsf{init}, \mathsf{t}.\phi(\mathsf{N}(\mathsf{init}, \mathsf{t})).$$

Le second, correspondant à la figure 8.15 permet de prouver que si la propriété est satisfaite à un étage de l'itération ($\phi(acc)$), alors elle est aussi satisfaite à l'étage suivant (après application du nœud itéré, $\phi(N(acc, t))$) :

$$\lambda \mathsf{acc}, \mathsf{t}.\phi(\mathsf{acc}) \Rightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{t})).$$

La règle est décrite sous la forme donnée à la figure 8.16.

8.5.2.4 Exemple

Voici le résultat de l'application de cette règle sur l'observateur de la figure 8.11. Pour prouver que la propriété est satisfaite après le premier appel au nœud itéré, on va générer deux équations :

```
okInit = phi(v_init);
v_init = plus(0,elt_T_init);
```

La première représente la valeur de vérité de la propriété après un passage dans la boucle de l'itération (variable okInit). La seconde représente le premier passage dans la boucle elle-même (la valeur calculée au premier niveau de la réduction). C'est un appel au nœud itéré avec comme paramètres :

- l'initialisation de l'itération (ici, la constante 0) ;
- une variable, quantifiée universellement (c'est une entrée de l'observateur) représentant le premier élément du tableau T, elt_T_init.

Ensuite, on doit générer les équations décrivant l'invariance de la propriété :

```
propRankN = phi(v_acc);
propRankNp1 = phi(v_inv);
v_inv = plus(v_acc, elt_T_inv);
okInv = propRankN => propRankNp1);
```

Les variables ainsi définies ont la signification suivante :

- La variable propRankN représente la valeur de vérité de la propriété à un rang quelconque N. Pour la définir on applique la propriété phi à une entrée quelconque (v_acc). Cette équation code l'*hypothèse de récurrence* de la propriété ;
- La variable propRankNp1 représente la valeur de vérité de la propriété au rang N+1. Pour la définir on applique la propriété phi à la variable v_inv ;
- v_inv est définie par un appel du nœud itéré dans la réduction sur la variable v_acc (sur laquelle on s'est assuré de la validité de phi dans l'équation précédente) et à une variable elt_T_inv représentant un élément du tableau T ;

- Enfin, okInv représente la règle $\lambda \mathsf{acc}, \mathsf{t}.\phi(\mathsf{acc}) \Rightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{t}))$.

La validité de la propriété est définie comme la conjonction de l'initialisation et de l'invariance :

```
ok = okInit and okInv;
```

Ces équations sont toutes générées dans un nouvel observateur, donné à la figure 8.17.

```
node Obs_dup(v_acc : int; elt_T_init, elt_T_inv : int)
returns (ok : bool);
var okInit : bool; okInv : bool;
    propRankN : bool; propRankNp1 : bool;
    v_init : int; v_inv : int;
let
  okInit = phi(v_init);
  v_init = plus(0, elt_T_init);

  okInv = (propRankN => propRankNp1);
  propRankN = phi(v_acc);
  propRankNp1 = phi(v_inv);
  v_inv = plus(v_acc, elt_T_inv);

  ok = (okInv and okInit);
tel
```

FIG. 8.17 – Observateur résultant de l'application de la règle 8.16.

8.5.2.5 Tableaux Explicites

Les traitements que nous avons proposé tirent partie du fait qu'on peut tout à fait confondre les différents éléments des tableaux sur lesquels porte la propriété à prouver. On abstrait seulement le calcul du premier élément qui est particulier (par l'utilisation de l'expression d'initialisation de l'itération).

Le rang des éléments considérés pour prouver l'invariance n'a ici aucune importance. Par contre, ce rang peut être important lorsque certains des tableaux que l'on considère sont définis de manière explicite (on donne la valeur exacte de chaque élément comme dans l'expression [1,2,3,4]). Dans ce cas, le fait d'employer la solution que nous avons proposé renforce la définition du tableau explicite : on va remplacer les valeurs explicites par des variables représentant la valeur du tableau à des rangs quelconques N et N+1.

Si l'on ne souhaite pas abstraire ces valeurs, deux choix s'offrent à nous. La première solution consiste à découper la propriété en autant de sous-buts qu'il y a d'élément de tableaux. On perd alors l'intérêt de la méthode décrite, mais on garde la définition de la propriété intègre. La seconde solution consiste à découper la propriété en deux sous-buts Init et Inv, en exprimant un choix possible entre toutes les valeurs possibles de l'élément du tableaux explicite abstrait pour Inv. L'exemple suivant illustre cette possibilité.

EXEMPLE 37 — Considérons une propriété exprimée à l'aide d'une itération d'un nœud P qui prend 4 entrées : un accumulateur booléen représentant la validité de la propriété et 3 éléments de tableaux. La propriété est exprimée par :

```
node prop(A : int^5) returns (ok : bool);
let
   ok = red≪P;5≫(true,A,[1,2,3,4,5],[3,4,5,6,7]);
tel

node P(acc_in : bool; elt1, elt2, elt3 : int) returns (acc_out : bool);
let
   acc_out = acc_in and elt1=>elt2 and elt1<=elt3
tel
```

Si l'on abstrait on perd une information importante sur la propriété à prouver. On peut construire l'objectif de preuve suivant, où okInit représente le fait que le premier élément de A soit compris entre 1 et 3 (les premiers éléments des tableaux explicites), et où okInv représente le fait qu'à chaque étage N, si le N-ième élément de A est compris entre les N-ièmes éléments des tableaux explicites alors le N+1-ième élément de A est compris entre les N+1-ième éléments des tableaux explicites.

```
node g_prop2_dup1(A_init, A_inv : int; B_init, B_inv : int;
                                C_init, C_inv : int; propTrueRankN : bool)
returns (ok : bool);
var okInit, okInv : bool; propTrueRankNp1 : bool;
let
   okInit = P(true, A_init, B_init, C_init);
   propTrueRankNp1 = P(propTrueRankN, A_inv, B_inv, C_inv);
   okInv = ((propTrueRankN) => (propTrueRankNp1));
   ok = ((okInv) and (okInit));
tel
```

Mais la propriété exprimée au départ est plus précise. Elle stipule exactement que le N+1-ième élément doit être compris entre 2 et 4, 3 et 5, 4 et 6 ou entre 5 et 7. Afin de préserver cette information, on pourra générer l'observateur suivant :

```
node g_prop2_dup1(A_init, A_inv : int; B_inv : int; C_inv : int;
                              propTrueRankN : bool)
returns (ok : bool);
var okInit, okInv : bool;
    propTrueRankNp1 : bool;
let
   okInit = P(true, A_init, 1, 3);
   propTrueRankNp1 = P(propTrueRankN, A_inv, B_inv, C_inv);
   okInv = ((propTrueRankN) => (propTrueRankNp1));
   ok = ((okInv) and (okInit));
   assert ((B_inv = 2) and (C_inv = 4)) or ((B_inv = 3) and (C_inv = 5))
          or ((B_inv = 4) and (C_inv = 6)) or ((B_inv = 5) and (C_inv = 7))
tel
```

— FIN DE L'EXEMPLE 37

```
node main(T : int^10; initP, initH : bool) returns (ok : bool);
let
   ok = red≪P;size≫(initP,T);
   assert red≪P;size≫(initH,T);
tel
```

FIG. 8.18 – Une propriété itérative accompagnée d'une hypothèse itérative.

8.5.2.6 Prise en compte d'une hypothèse itérative

Nous nous intéressons maintenant au cas où la propriété que nous devons prouver est donnée avec une *hypothèse*. En LUSTRE, on a vu qu'on pouvait spécifier une telle hypothèse en ajoutant à l'observateur une clause booléenne repérée par le mot-clé *assert*. Ce paragraphe s'intéresse au traitement d'une telle hypothèse lorsque l'expression utilisée est une itération.

EXEMPLE 38 — Nous donnons à la figure 8.18 un observateur LUSTRE dans lequel la propriété à prouver est exprimée par une itération portant sur un tableau T :

```
ok = red≪P;size≫(initP,T);
```

et où on donne une hypothèse itérative exprimée elle aussi par une itération portant sur le tableau T :

```
assert red≪H;size≫(initH,T)
```

— FIN DE L'EXEMPLE 38

Nous proposons dans ce cas d'appliquer une variante de la règle de la figure 8.5.2 où l'on prend en compte une hypothèse.

Afin de rendre plus claire l'explication de cette transformation, nous revenons à la forme impérative des itérations calculant la propriété *et* l'hypothèse. Le calcul de l'hypothèse peut-être représenté par une boucle for calculant une variable hypo ; le calcul de la propriété est représenté comme d'habitude par une boucle for calculant la variable ok.

```
hypo = initH;
for(int i=0;i<size;i++){
  hypo = H(hypo,T[i]);
}

ok = initP;
for(int i=0;i<size;i++){
  ok = P(ok,T[i]);
}
Prop = hypo ⇒ ok                α
```

En prenant en compte l'hypothèse itérative de l'observateur, on veut prouver que la propriété Prop = hypo $\Rightarrow$ ok est vraie au point α. Remarquons que ces deux boucles for sont strictement équivalentes à la forme fusionnée suivante :

```
hypo = initH;                          |
ok = initP                             |
for(int i=0;i<size;i++){               |
                                       | γ
  hypo = H(hypo,T[i]);                 |
  ok = P(ok,T[i]);                     |
                                       | γ'
}                                      |
Prop = hypo ⇒ ok                       | α
```

Pour prouver que Prop est vraie, nous appliquons le *même renforcement qu'auparavant*. On prouvera que :

- Prop est vrai après un passage dans la boucle (soit au point γ' avec i=0, hypo=initH et ok=initP) ;
- Prop est préservé par le corps de boucle : si Prop est vrai en γ, alors Prop est vrai en γ', quelle que soit la valeur courante de i.

8.5.3 Propagation aux enchaînements d'itérations

Dans le paragraphe précédent, nous avons étudié le traitement d'une propriété itérative simple où chaque tableau de l'itération calculant la propriété était une entrée de l'observateur.

On souhaite maintenant appliquer la même transformation dans le cas où d'autres itérations sont utilisées dans l'observateur. Une première solution consiste à applique l'algorithme d'optimisation des enchaînements d'itérations proposé au paragraphe 4.5 et d'appliquer ensuite la règle précédente.

Le programme produit par la transformation que nous proposons doit néanmoins être le plus lisible possible (le plus proche possible du programme original) par l'utilisateur. Donc nous avons choisi de conserver la structure d'enchaînements d'itérations et de générer pour ceux-ci des enchaînements d'appels de noeuds.

Nous décrivons dans les paragraphes qui suivent des cas très simples d'enchaînements manipulant au plus deux itérations (la propriété *et* une itération calculant les tableaux sur lesquels porte la propriété. L'algorithme que nous donnons plus loin généralise ces règles en propageant cette transformation à toutes les itérations utilisées pour calculer la propriété étudiée.

8.5.3.1 Enchaînements « *map suivi de red* »

Considérons maintenant l'enchaînement d'itérations de la figure 8.19. Cette fois, le tableau T sur lequel est évaluée la propriété n'est pas une entrée du programme. Il est calculé par une itération d'un nœud plusUn tel que chaque élément de T soit égal à la valeur de l'élément correspondant de T' augmentée de 1 :

$$\forall i.\ T[i] = T'[i] + 1.$$

Nous voulons prouver la même propriété que plus haut et donnons pour cela l'observateur de la figure 8.19. Comme nous l'avons fait dans le paragraphe précédent, nous donnons à la figure 8.20 le code impératif correspondant à ce programme.

La figure 8.21 donne la forme optimisée de cet enchaînement (obtenue par l'algorithme d'optimisation présenté au chapitre 4). On a indiqué les points α, γ et γ'. Sur cette forme, il serait évident d'appliquer la même méthode que précédemment. Comme nous l'avons souligné plus haut, appliquer

```
node Obs(T' : intˆs) returns (ok : bool);
var n : int;
    T : intˆs;
let
  T = map≪plusUn;s≫(T');
  n = red≪plus;s≫(0,T);
  ok = phi(n);
tel
node plusUn(t' : int) returns (t : int);
let
  t = t' + 1;
tel
```

FIG. 8.19 – Un enchaînement d'itérations.

```
for(int i=0;i<s;i++){
   T[i] = plusUn(T'[i]);
}

Acc = 0;
for(int i=0;i<s;i++){
   Accu = plus(Accu,T[i]);
}
n = Accu;
```

FIG. 8.20 – Version impérative du calcul de la variable n.

```
Acc = 0;                          | β
for(int i=0;i<s;i++){             |
                                  | γ
  T[i] = plusUn(T'[i])            |
  Acc = plus(Accu,T[i]);          |
                                  | γ'
}                                 |
n = Accu;                         | α
```

FIG. 8.21 – Version après la fusion des deux boucles.

d'abord les axiomes d'optimisation d'enchaînements d'itérations puis la règle de transformation du paragraphe précédent conduit à une perte totale de la structure du programme et rend plus difficile la relecture du programme produit par l'utilisateur.

On propose plutôt une règle de manipulation similaire à celle proposée plus haut pour une itération simple qui permet de conserver l'enchaînement des itérations au niveau du programme manipulant les noeud Init et Inv. La figure 8.22 illustre cette règle.

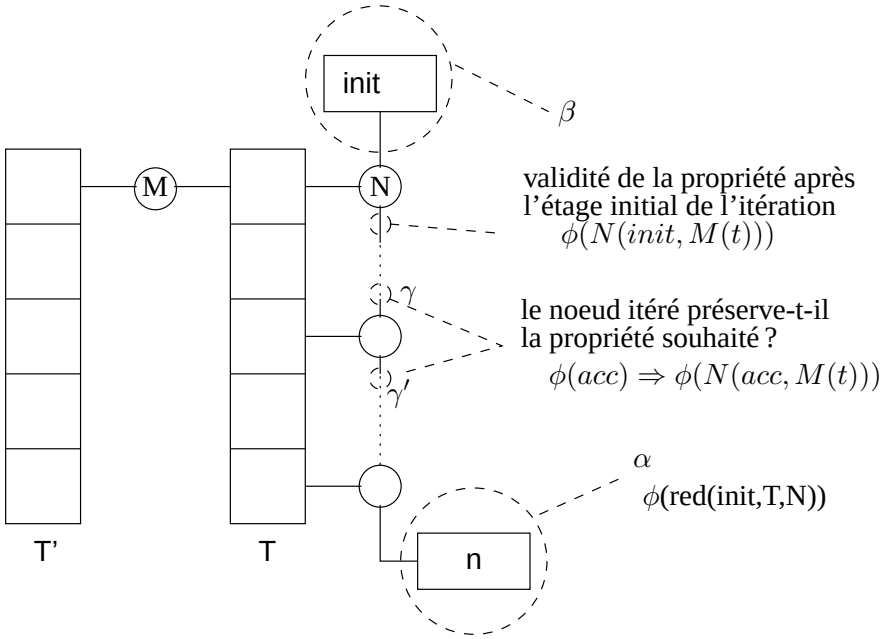


FIG. 8.22 – Représentation graphique du calcul de la variable n.

8.5.3.2 Règle de manipulation associée

À partir de l'observateur suivant décrivant la propriété ϕ :

$$\lambda \mathsf{init}, \mathsf{T}.\phi(\mathsf{red}(\mathsf{init}, \mathsf{map}(\mathsf{M}, \mathsf{T}), \mathsf{N}))$$

on génère deux observateurs. Le premier permet de prouver que ϕ est vrai après un passage dans l'enchaînement d'itérations :

$$\lambda \mathsf{init}, \mathsf{t}.\phi(\mathsf{N}(\mathsf{init}, \mathsf{M}(\mathsf{t}))).$$

Le second permet de prouver que si la propriété est satisfaite à un étage de l'enchaînement d'itération ($\phi(acc)$) alors et est aussi satisfaite à l'étage suivant (après application des deux nœuds itérés, $\phi(N(acc, M(t))$:

$$\lambda \mathsf{acc}, \mathsf{t}.\phi(\mathsf{acc}) \Rightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{M}(\mathsf{t}))).$$

Cette règle est décrite à la figure 8.23.

$$\frac{\begin{array}{c}\lambda \mathsf{init}, \mathsf{t}.\phi(\mathsf{N}(\mathsf{init}, \mathsf{M}(\mathsf{t}))) \\ \lambda \mathsf{acc}, \mathsf{t}.\phi(\mathsf{acc}) \Rightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{M}(\mathsf{t})))\end{array}}{\lambda \mathsf{init}, \mathsf{T}.\phi(\mathsf{red}(\mathsf{N}, \mathsf{map}(\mathsf{N}, \mathsf{T}), \mathsf{init}))}$$

FIG. 8.23 – Règle de construction d'obligations de preuves pour l'enchaînement red(map).

8.5.3.3 Exemple

Pour l'observateur de la figure 8.19, on obtient le nœud Obs_dup de la figure 8.24. On a bien généré :

- une équation pour le cas initial de la propriété :

  okInit = (n_init > 0);
- une équation pour l'invariance de la propriété :

  okInv = propRankN => propRankNp1;
- Les équations nécessaires à la propagation de la transformation (définissant entre autres elt_T_init et elt_T_inv pour le tableau T).

8.5.4 Propagation dans un réseau quelconque

La règle que nous venons de donner permet de transformer un enchaînement de deux itérations. Il est cependant possible que d'autres itérations soient utilisées pour calculer les tableaux utilisés par la propriété. On va donc généraliser le traitement d'enchaînements en découpant toutes les itérations qui sont utilisés dans le programme pour calculer la propriété. Ce slice s'effectue en parcourant le graphe de définition de la propriété vers « l'arrière ». Pour chaque itération, on génère :

– un appel au noeud itéré, utilisé pour calculer le cas init de l'induction ;
– un appel utilisé pour calculer les cas inv.

On s'intéressera seulement aux itérations qui calculent les paramètres tableaux de la propriété. Nous donnons un exemple de traitement plus loin. Par contre, on ne modifiera pas le code utilisé pour calculer les initialisations des itérations. Par exemple, si on a le programme de la figure 8.25, on devra conserver l'itération de N pour calculer la valeur initiale de l'itération de P.

8.5.4.1 Exemple

Considérons la propriété exprimée par l'observateur de la figure 8.29 (donnée à la page 214). On ne s'intéresse pas ici au sens de cette propriété. On ne décrira donc pas le détail des nœuds itérés. La sortie booléenne ok de cet observateur est calculée par une réduction du nœud P appliquée au tableau tab_out. Celui-ci est calculé par un map d'une nœud N sur un tableau T4. T4 est une des sorties d'un map_red du nœud Q. Cette itération calcule aussi la variable res4 et le tableau T5 qui ne nous intéressent pas pour la propriété. Ses paramètres sont :

- la constante 1 qui initialise l'itération ;
- les deux tableaux T2 et T3.

Le tableau T2 est calculé par un map du nœud S sur un tableau T1, lui-même rempli à l'aide de l'opérateur fill initialisé avec une entrée de l'observateur, appelée init_fill. Enfin, le tableau T3 est calculé avec la variable res3, par une itération de type map_red qui utilise le nœud R sur le tableau tab_in (une entrée de l'observateur).

```
node Obs_dup(n_acc : int;accIn_acc : int;
                    elt_T2_init : int; elt_T2_inv : int)
returns (ok : bool);
var
     okInit : bool; okInv : bool;
     n_init : int; n_inv : int;
     propRankN : bool; propRankNp1 : bool;
     elt_T_init : int; elt_T_inv : int;
let
– nouvelle propriété à prouver
   ok = (okInv and okInit);

– propriété vraie après première itération
   okInit = phi(n_init);

– propriété préservée par le nœud itéré
   okInv = (propTrueRankN => propTrueRankNp1);

– propriété vraie au rang N
   propTrueRankN = phi(n_acc);

– propriété vraie au rang N+1
   propTrueRankNp1 = phi(n_inv);
   n_init = g_plus(0, elt_T_init);
   n_inv = g_plus(n_acc4, elt_T_inv);
   elt_T_init = g_plusUn(elt_T2_init);
   elt_T_inv = g_plusUn(elt_T2_inv);
tel
```

FIG. 8.24 – Résultat de l'application de la règle d'enchaînement map(red) à l'exemple de la figure 8.19.

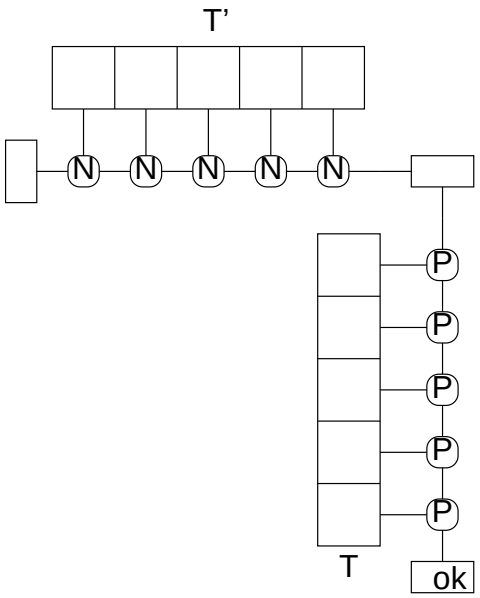


FIG. 8.25 – Cas non traité : initialisation d'une réduction par une autre réduction

Une version graphique du réseau d'opérations constituant cet observateur est donnée à la figure 8.30 (donnée à la page 214).

La transformation de cet observateur se fait progressivement à partir de la réduction calculant la sortie ok. Comme nous l'avons vu précédemment, on commence par générer deux équations okInit et okInv stipulant que :

- la propriété est vraie après l'appel initial au nœud itéré (okInit) ;
- la propriété est préservée par un appel du nœud itéré okInv).

Pour calculer okInv on définit les deux variables :

- propRankN qui est une entrée du nouvel observateur et qui représente toujours l'hypothèse « la propriété est vraie au rang N » ;
- propRankNp1 qui représente le fait que « la propriété est vraie au rang N+1 ».

Ensuite on propage la génération des cas init et inv aux itérations de l'enchaînement jusqu'à atteindre les entrées de l'observateur. Par exemple, pour l'itération :

```
res3, T3 = map_red≪R;size≫(1, tab_in);
```

on va générer les deux équations :

```
res3_init,elt_T3_init = R(1, elt_tab_in_init);
res3_inv,elt_T3_inv = R(accIn_R, elt_tab_in_inv);
```

De la même façon, pour l'itération :

```
varOut, T1 = fill≪T;size≫(init_fill);
```

on va générer les équations :

```
varOut_init,elt_T1_init = T(init_fill);
varOut_inv,elt_T1_inv = T(accIn_T);
```

L'observateur ainsi généré est donné à la figure 8.32. Une version graphique de ce nœud est donné à la figure 8.33. Pour des raisons de lisibilité, ces deux figures sont données à la fin du chapitre, pages 216 et 217.

8.5.5 Prise en compte des contrats

Pour l'instant nous avons ignoré d'éventuels appels de nœuds dans les équations définissant la sortie de l'observateur. Il s'agit en fait de l'approche « *standard* » qui considère que les appels de nœuds éventuels ont déjà été expansés par un pré-processeur. Le nœud qu'on a considéré jusqu'à présent peut ainsi être considéré comme écrit dans un format intermédiaire de type EC (le format intermédiaire du compilateur LUSTRE, dans lequel les appels de nœuds ont tous été expansés) « étendu » pour inclure les itérations.

Cependant, nous souhaitons pouvoir tirer partie de la structuration du programme en nœuds. Dans le cadre de développement que nous proposons dans cette thèse, les nœuds sont spécifiés à l'aide des contrats du chapitre 5 et il paraît intéressant d'exploiter ces contrats dans le cas de la preuve de propriétés itératives que nous traitons ici (dans le chapitre 9 nous reviendrons sur l'exploitation des contrats proprement dite). Dans ce paragraphe, nous nous intéressons à la question suivante :

Si l'on rencontre un nœud à contrat lors de la transformation d'un enchaînement d'itérations, comment peut-on l'exploiter ?

Le traitement que nous proposons maintenant consiste à oublier le corps du nœud rencontré et à ne considérer que son contrat. Comme nous le verrons, on se ramène dans un premier temps au cas où aucun nœud n'est appelé et où on doit manipuler une *hypothèse* dans le traitement de la propriété itérative. Ensuite, on applique la méthode proposée au paragraphe précédent qui prend en compte l'hypothèse générée.

8.5.5.1 Exemple simple

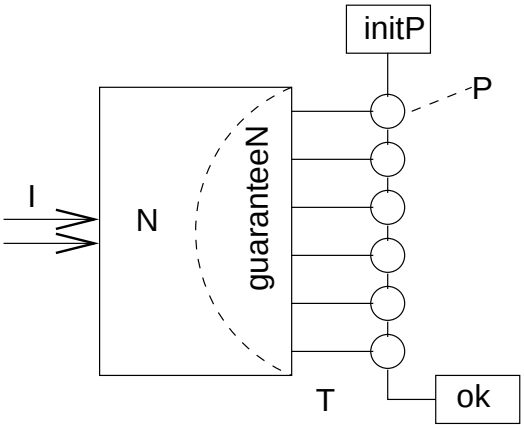
Considérons le programme de la figure 8.26. La sortie ok représente la valeur de vérité d'une propriété calculée par une itération à résultat booléen, appliquée à un tableau T. Ce tableau est l'unique résultat d'un composant N dont on connaît le contrat :

- Son assertion est la propriété true, on ne la considère pas pour l'instant ;
- Sa garantie est représentée par le nœud guaranteeN dont la sortie okGuarantee est calculée par une réduction simple d'un nœud H.

8.5.5.2 Principe

Renforcement d'une hypothèse par le contrat du nœud appelé – Dans un premier temps on va abstraire la présence du nœud N en considérant la variable T comme une nouvelle entrée du système. Nous ne voulons cependant pas oublier complètement N, mais prendre en compte *seulement* son contrat. Nous utilisons donc celui-ci comme hypothèse de la propriété à prouver. Le nœud okGuarantee est donc maintenant appelée pour exprimer une hypothèse de la propriété. Par expansion de ce nœud, cette nouvelle hypothèse peut-être rapportée à une réduction du nœud H. La nouvelle version de l'objectif de preuve à traiter est l'observateur donné à la figure 8.18.

Traitement de l'hypothèse générée – Nous avons montré comment traiter cet observateur particulier : on considère la propriété à traiter comme étant hypo $\Rightarrow$ prop et on cherche à prouver que cette propriété est une invariance de l'ensemble des itérations.




```
node main(I : ... ; initP : bool) returns (ok : bool);
var T : intˆsize;
let
    T = N(I);
    ok = red≪P;size≫(initP,T);
tel

node N(I : ...) returns (T : intˆsize);
%ASSUME:assumeN%
%GUARANTEE:guaranteeN%
let
    ...
tel

node guaranteeN(I : ...; T : intˆsize;initH : bool) returns (okGuarantee : bool);
let
    okGuarantee = red≪H;size≫(initH;T);
tel
```

FIG. 8.26 – Une propriété itérative sur un programme utilisant un nœud N.

```
node main2(I : int^size; T : int^size; initP) returns (ok : bool);
let
   ok = red≪P;size≫(initP,T);
   assert assumeN(I) => guaranteeN(I,T);
tel
```

FIG. 8.27 – Une propriété itérative accompagnée d'une hypothèse itérative.

Forme générale du contrat – La transformation présentée ci-dessus ne s'applique pas seulement dans le cas où la garantie du nœud N est une réduction simple. En général le nœud guaranteeN est quelconque. On se contentera de générer un appel de nœud. Pour l'exemple donné, on obtiendrait l'assertion :

assert guaranteeN(I,T)

À partir de cette hypothèse, une étape d'expansion des appels de nœuds dans la clause assert du nouvel observateur permet de faire apparaître les éventuelles réductions. Cette expansion consiste à :

- générer en hypothèse du nouvel observateur l'expression calculant la sortie booléenne de guaranteeN ;
- ajouter les variables locales de guaranteeN aux variables locales de main ;
- ajouter les équations définissant ces variables locales aux équations de main.

L'observateur de la figure 8.18 est une forme simplifiée où on a expansé l'appel du nœud guaranteeN et opéré d'autres simplifications triviales (suppression les occurrences des variables d'entrées I de N qui sont inutiles dans ce cas).

Prise en compte de la clause assume **du nœud appelé** – Jusqu'ici, nous avons considéré que la clause assume du nœud appelé était équivalente à true : on pouvait sans problème l'oublier. En général, cette clause est représentée par un nœud assumeN quelconque. On devra alors générer une hypothèse de la forme :

assert assumeN(I) $\Rightarrow$ guaranteeN(I,T)

Cette forme décrit le fait que pour prouver une propriété P faisant intervenir un nœud N, on oublie volontairement comment N est réalisé : on se contente de supposer que les entrées/sorties de N satisfont bien son contrat.

8.5.5.3 Exemple

L'exemple suivant illustre les dernières remarques. Considérons le contrat du noeud N donné plus haut, à la figure 8.26. On va traiter le même observateur que dans le paragraphe précédent, en utilisant ce contrat pour N.

Dans un premier temps on remplace l'appel à N par une hypothèse manipulant son contrat. On obtient ainsi le nœud mainBis de la figure 8.27 ressemblant de très près à celui de la figure 8.18. Ensuite, on peut expanser les appels assumeN(I) et guaranteeN(I,T). Après introduction de variables intermédiaires pour rendre le programme plus lisible, on obtient l'observateur de la figure 8.28.

À partir du nœud main3, on applique le traitement du paragraphe 8.5.2. On obtient ainsi l'observateur de la figure 8.31 (donnée à la fin du chapitre, page 215). Nous rappelons brièvement l'utilité de certaines variables de ce nouveau programme :

```
node main3(I,T : int^size; initP : bool) returns (ok : bool);
var
    assumeOK, guaranteeOK : bool;
    sum : int;
    tousPositif : bool;
let
  ok = red<<p;size>>(initP,T);

  assumeOK = red<<unPositif;size>>(true;I);

  guaranteeOK = unPositif(sum) and tousPositif;
  sum = red<<+;size>>(0;T);
  tousPositif = red<<unPositif;size>>(true;T);

  assert assumeOK => guaranteeOK;
tel
```

FIG. 8.28 – Le nœud N a été remplacé par son contrat.

- ok représente la nouvelle propriété à prouver ; Elle est calculée comme la conjonction de okInit et okInv ;
- okInit représente la validité de la propriété après *un* passage dans l'itération initiale. Elle est définie comme l'implication hypo_init => okInv c'est-à-dire « *si l'hypothèse est vraie après un passage dans la boucle* alors *la propriété est vraie après un passage dans la boucle* ». hypo_init et okInit sont calculées en découpant les équations définissant les variables assumeOK et guaranteeOK et ok selon l'algorithme classique ;
- okInv représente la préservation de la propriété hypo => prop le long de l'itération. On doit prouver que si hypo => prop est vraie à un étage quelconque de l'itération (hypothèse représentée par la variable HPAcc) alors la propriété est vraie à l'étage suivant de l'itération (représenté par la variable HPInv).HPAcc sont HPInv sont calculées en propageant l'algorithme présenté plus haut aux variables calculant l'hypothèse assumeOK => guaranteeOK et la propriété ok ;
- Les autres variables sont générées par la propagation décrite au paragraphe 8.5.3.

8.5.6 Conclusion

Dans le chapitre 10, nous montrerons l'usage de ces transformations appliquées aux exemples des chapitres 6 et 7. Nous verrons que cette méthode est essentiellement intéressante sur les structures très régulières, mais que ce sont ces structures que nous avons le plus rencontrées dans les études de cas.

Nous l'avons déjà souligné plus haut, cette approche présente certaines limitations évidentes. Une première limitation est due au fait qu'on renforce nécessairement la propriété à traiter en un invariant des itérations du programmes. Il existe des techniques de calcul d'invariant à partir des propriétés à prouver que nous n'avons pas étudiées, puisque la technique proposée suffit pour exemples que nous avons étudié.

Par ailleurs, nous avons souligné des limitations dans la prise en compte de la structure du réseau d'opérateurs du programme. On pourrait imaginer des extensions à la méthode proposée pour prendre en compte des formes plus générales, en traitant par exemple les itérations utilisées pour calculer les initialisations des itérations utilisées pour la propriété. Ce travail pourrait s'orienter vers des méthodes

de slice plus générales sur les programmes comme celles déjà mis en place dans LUDIC [Gau03].

Une autre extension possible pourrait consister, lorsque le réseau paraît trop compliqué pour appliquer la méthode décrite au paragraphe précédent, à encourager l'utilisateur à découper l'application en sous-composants et à donner des contrats pour ces sous-composants. On pourra alors utiliser ces contrats dans la preuve pour la simplifier (comme suggéré au paragraphe 8.5.5). A l'inverse, on peut proposer l'expansion simple des noeuds utilisés si l'utilisateur la juge plus utile que l'utilisation du contrat.

Globalement, notre méthode est intrinsèquement incomplète. Elle pourrait être raffiné pour prendre en compte d'autres cas de structures de programmes, mais resterait incomplète. Nous donnons l'algorithme implémenté dans notre prototype dans l'annexe B.

```
node prop(init_fill : int; tab_in : int^size)
returns (ok : bool);
var tab_out : int^size;
      T1, T2, T3, T4, T5 : int^size;
      varOut, res3, res4 : int;
let
    varOut, T1 = fill<<T;size>>(init_fill);
    T2 = map<<S;size>>(T1);
    res3, T3 = map_red<<R;size>>(1, tab_in);
    res4, T4, T5 = map_red<<Q;size>>(0, T2, T3);
    tab_out = map<<N;size>>(T4);
    ok = red<<P;size>>(true, tab_out);
tel
```

FIG. 8.29 – Propriété mettant en jeu diverses itérations.

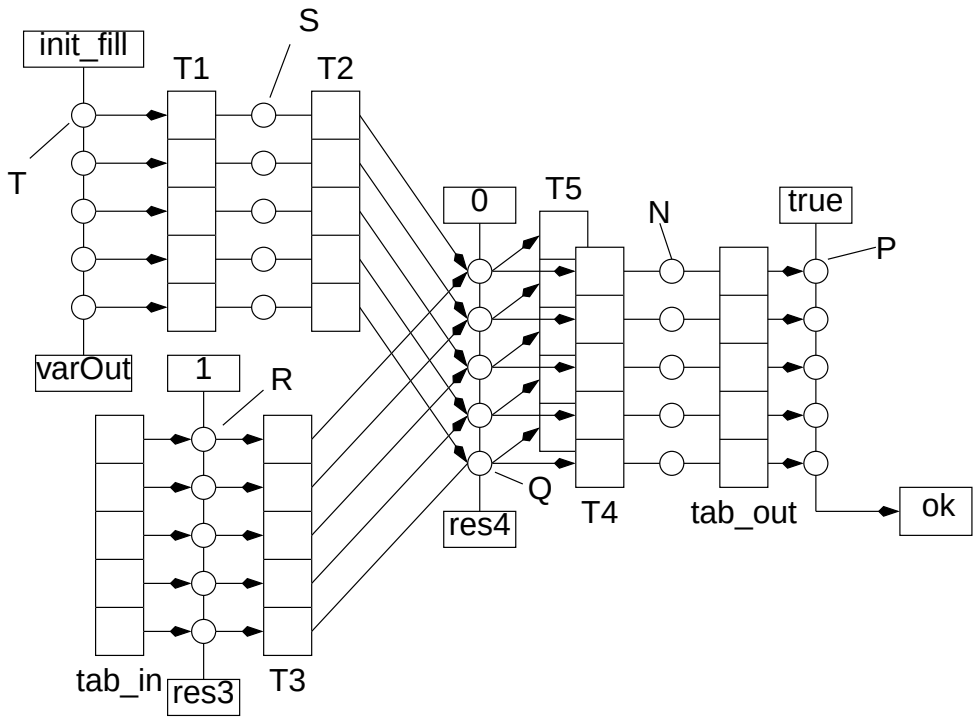


FIG. 8.30 – Réseau d'itérations de l'observateur prop.

```
node g_cascade_dup1(initP_init : bool; initP_inv : bool;
                    elt_T_init : int; elt_T_inv : int;
                    propTrueRankN : bool; assumeOK_acc : bool;
                    accIn_acc : bool; elt_I_init : int;
                    elt_I_inv : int; sum_acc : int;
                    tousPositif_acc : bool)
returns (ok : bool);
var okInit, okInv : bool;
    propTrueRankNp1 : bool;
    hypoTrueRankN : bool; hypoTrueRankNp1 : bool;
    hypo_init, ok_init : bool;
    assumeOK_init : bool; assumeOK_inv : bool;
    guaranteeOK_init : bool; guaranteeOK_inv : bool;
    guaranteeOK_acc : bool;
    sum_init : int; sum_inv : int;
    tousPositif_init : bool; tousPositif_inv : bool;
    HPAcc : bool; HPInv : bool;
let
– la propriété à prouver
  ok = (okInit and okInv);

– la propriété est vrai après un passage dans l'itération
  okInit = (hypo_init => ok_init);
  ok_init = g_unPositif(initP_init, elt_T_init);
  hypo_init = (assumeOK_init => guaranteeOK_init);

– la propriété est préservée par le nœud itéré
  okInv = (HPAcc => HPInv);
  HPAcc = (hypoTrueRankN => propTrueRankN);
  HPInv = (hypoTrueRankNp1 => propTrueRankNp1);
  hypoTrueRankN = (assumeOK_acc => guaranteeOK_acc);
  hypoTrueRankNp1 = (assumeOK_inv => guaranteeOK_inv);
  propTrueRankNp1 = g_unPositif(propTrueRankN, elt_T_inv);

– propagation du découpage des itérations
  assumeOK_init = g_unPositif(true, elt_I_init);
  assumeOK_inv = g_unPositif(assumeOK_acc, elt_I_inv);
  guaranteeOK_init = (g_unPositif(true, sum_init) and tousPositif_init);
  guaranteeOK_inv = (g_unPositif(true, sum_inv) and tousPositif_inv);
  guaranteeOK_acc = (g_unPositif(true, sum_acc) and tousPositif_acc);
  tousPositif_init = g_unPositif(true, elt_T_init);
  tousPositif_inv = g_unPositif(tousPositif_acc, elt_T_inv);
  sum_init = g_plus(0,elt_T_init);
  sum_inv = g_plus(sum_acc,elt_T_inv);
tel
```

FIG. 8.31 – Résultat de l'algorithme appliqué à l'exemple de la figure 8.28.

```
node prop_dup
(tab_out_acc33 : int;      accIn_Q : int;      init_fill : int;    elt_tab_in_init : int;
 propRankN : bool;         accIn_T : int;      accIn_R : int;      elt_tab_in_inv : int;)
returns (ok : bool);
var                                          let
  okInit : bool;             okInv : bool;       okInit = P(true, tab_out_init);
                                                 okInv = (propRankN => propRankNp1);
  propRankNp1 : bool                             propRankNp1 = P(propRankN, tab_out_inv);
  tab_out_init : int;        tab_out_inv : int;  tab_out_init = N(elt_T4_init);
                                                 tab_out_inv = N(elt_T4_inv);

  -- sorties de S                                elt_T2_init = S(elt_T1_init);
  elt_T2_init : int;                             elt_T2_inv = S(elt_T1_inv);
  elt_T2_inv : int;

  -- sortie de Q
  res4_init : int;           elt_T4_init : int;  res4_init,elt_T4_init,elt_T5_init = Q(0, elt_T2_init, elt_T3_init);
  elt_T5_init : int;
  res4_inv : int;            elt_T4_inv : int;   res4_inv,elt_T4_inv,elt_T5_inv = Q(accIn_Q, elt_T2_inv, elt_T3_inv)
  elt_T5_inv : int;

  -- sorties de T
  varOut_init : int;         elt_T1_init : int;  varOut_init,elt_T1_init = T(init_fill);
  varOut_inv : int;          elt_T1_inv : int;   varOut_inv,elt_T1_inv = T(accIn_T);

  -- sortie de R
  res3_init : int;           elt_T3_init : int;  res3_init,elt_T3_init = R(1, elt_tab_in_init);
  res3_inv : int;            elt_T3_inv : int;   res3_inv,elt_T3_inv = R(accIn_R, elt_tab_in_inv);
```

FIG. 8.32 – En-tête du nœud obtenu pour le nœud prop.

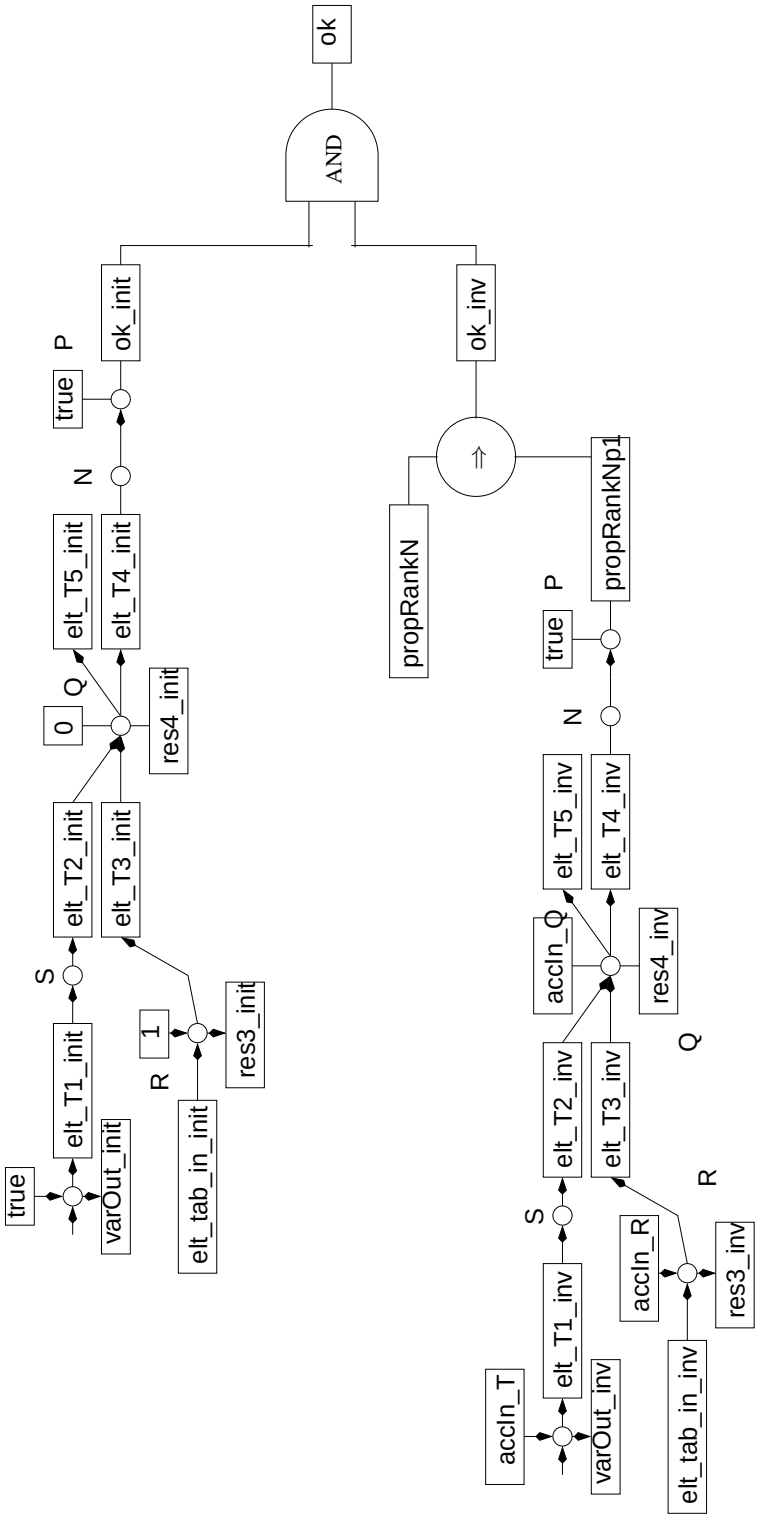


FIG. 8.33 – Réseau d'opérateurs de l'observateur prop_dup.

Chapitre 9

Manipulations de contrats

Dans ce chapitre, nous montrons comment tirer partie des contrats dans la validation des systèmes. On s'intéressera à deux formes de validation, d'un côté les validations locales (comme l'adéquation d'une implémentation vis-à-vis de son contrat) de l'autre les validations globales (liées essentiellement à la composition des nœuds à contrats). On étudiera aussi l'utilisation des contrats pour la validation des itérations. Enfin, on tentera de montrer l'impact des contrats et des méthodes de validation associées dans une idée de cycle de développement des systèmes.

9.1 Introduction

Nous avons déjà insisté (voir chapitre 5) sur l'utilité des contrats dans le processus de construction des systèmes. Nous souhaitons maintenant présenter les apports de l'approche proposée en termes de validation. En terme de simulation, on va pouvoir utiliser les contrats LUSTRE pour déterminer plus facilement l'origine d'un bogue. Nous revenons sur ce point à la section 9.3, dans lequel nous présentons le travail de magistère de Marc Vareille autour du débogueur LUDIC de Fabien Gaucher.

Dans cette thèse, nous nous intéressons plus à l'utilisation des contrats dans la vérification des systèmes. Nous avons identifié deux niveaux de prise en compte.

Prise en compte locale – Premièrement, les contrats vont nous aider à valider l'implémentation d'un composant donné (voir paragraphe 9.4.1). On ne s'intéresse pas ici à savoir si le composant est correctement utilisé dans son environnement, mais seulement à savoir si l'implémentation qu'on en donne est conforme à la spécification donnée par le contrat. Nous verrons plus loin comment vérifier qu'un corps de composant satisfait bien son contrat, ou comment s'assurer qu'un contrat est implémentable (c'est-à-dire qu'il est possible de trouver une implémentation le réalisant).

Prise en compte globale – Ensuite, les contrats permettent de mettre en place une méthodologie de développement mêlant des approches ascendante et descendante de conception. On pourra considérer des systèmes en cours de développement où certains composants sont entièrement implémentés et d'autres seulement partiellement. Tous ces composants sont décrits par leur contrats. Les différentes définitions données au paragraphe 9.4.2 sont des outils qui nous permettrons de donner au développeur une plus grande confiance dans les connexions qui existent entre différents composants. Il s'agit par exemple de vérifier que deux contrats peuvent être composés, de déterminer l'adéquation d'un

découpage en sous-composants (dont on connaît les contrats) vis-à-vis de la spécification (elle aussi donnée par contrat) du composant global.

Itérations et contrats – Enfin, nous nous sommes intéressés aux possibilités de validation apportées par l'utilisation des contrats dans le cas des itérations. On montrera comment utiliser les contrats pour s'assurer qu'une itération est, dans une certaine mesure, compatible avec le contrat du noeud itéré. La section 9.4 présente ces différents aspects de l'utilisation des contrats pour la vérification des composants.

Contrats et méthodologies de développement – La section 9.5 tente de présenter les apports des contrats dans la vision d'une méthodologie de développement générale pour les systèmes réactifs.

9.2 Travaux connexes : modèles compositionnels

Dans les travaux qu'on a présentés au chapitre 5, on ne parlait quasiment jamais de l'utilisation faite des contrats. La seule qui était proposée (le plus souvent) était l'exécution défensive des composants. Nous nous intéressons dans ce paragraphe aux travaux portant sur l'exploitation des contrats ou de formes de spécifications partielles des composants dans le processus de validation.

Les auteurs des différents travaux présentés maintenant s'intéressent à ce qui est communément appelé les « *modèles compositionnels* », c'est-à-dire une description des systèmes en un agencement de sous-systèmes *composés* à l'aide de règles de compositions *formelles*. Ce découpage permet de faciliter la validation d'un système (au niveau global) en la ramenant à la validation des composants constituant le système. Les propriétés globales sont alors déduites en composant des propriétés prouvées localement sur les différents composants.

Ceux-ci sont en général spécifiés à l'aide de deux propriétés, A et G (nos *assume* et *guarantee*) spécifiant ce que le composant suppose sur son environnement (A) et ce que celui-ci peut attendre du composant (G). Lorsque l'on compose deux composants $P1$ (spécifié par $(A1, G1)$) et $P2$ (spécifié par $(A2, G2)$) en un composant $P1 \| P2$, on doit montrer que $G1$ ne viole pas $A2$ et inversement que $G2$ ne viole pas $A1$. Si l'on prouve ces deux obligations, on peut en conclure que $P1 \| P2$ peut être décrit par un couple (A, G) où G est une conséquence de A, $G1$ et $G2$.

Cette méthode est appelée *raisonnement assume-guarantee*. Nous présentons les travaux les plus significatifs dans ce domaine au paragraphe suivant. Le paragraphe 9.2.2 présente plusieurs applications de cette approche au model-checking.

9.2.1 Raisonnement assume-guarantee

La preuve compositionnelle consiste à ramener la validation d'une composition de composants à la validation des composants eux-mêmes. Cette méthode a été proposée la première fois par Misra et Chandy dans [MC81]. Ils proposent une *règle de composition* qui permet de déduire la spécification d'un réseau de composants à partir de la spécification des composants eux-mêmes. Cette règle, reprise dans les travaux de Stark [Sta85] puis d' Abadi et Lamport [AL93, AL95] permet d'établir que le composant $P1 \| P2$ satisfait une garantie G tant que l'assertion A est valide lorsque les conditions suivantes sont validées :

- Chaque composant Pi satisfait G_i sachant A_i ;
- Chaque assertion A_i peut être déduite de l'assertion A et des garanties G_j des autres composants ($A \wedge \bigwedge_j G_j \Rightarrow A_i$) ;

- G peut-être déduite de l'assertion A ainsi que des garanties G_j de chaque composant ($A \wedge \bigwedge_j G_j \Rightarrow G$).

Ce principe de composition est important pour nous puisque d'après [AL93], « *il est valide si toutes les clauses* assumes *et tous les clauses* guarantee *sont des propriétés de sûreté* », ce qui est le cas en LUSTRE, où chacune de ces propriétés est décrite avec un observateur.

Dans [KL93], Lamport et Kurshan appliquent une variante de la règle de décomposition à la vérification d'un multiplicateur. Pour cet exemple, les auteurs utilisent un model-checker pour vérifier que chaque composant satisfait sa spécification, puis un theorem prover pour montrer que le composant complet satisfait sa spécification si chacun des composants satisfait la sienne.

Dans [Sha98], Shankar présente une alternative au raisonnement *assume-guarantee* de Abadi-Lamport appelée « *lazy compositional verification* ». Cette méthode se différencie de celle de [AL93] par plusieurs aspects liés à la forme de composition utilisée. Nous commentons maintenant ces deux approches et identifions les points qui nous semblent importants pour notre travail.

- Abadi/Lamport proposent d'utiliser la conjonction comme opération de composition. Cette méthode a l'avantage de préserver les propriétés des composants au niveau de la composition. Shankar souligne cependant un problème associé : chaque transition de la composition doit aussi être une transition de chaque composant. Cet aspect pose des problèmes d'explosion de l'espace d'état dans le cas de systèmes asynchrones (puisqu'on va générer des transitions inutiles pour certains composants) mais est tout à fait naturel pour des systèmes synchrones. De fait, et comme nous l'avons vu plus haut, la composition de composants synchrones est la simple conjonction des spécifications ;
- Dans le travail de Abadi et Lamport, la règle de composition donnée plus haut génère des obligations de preuve pour chaque composition : on doit notamment montrer que l'ensemble des garanties $\bigwedge_j G_j$ implique l'assertion A_i d'un composant. Dans [Sha98], on propose de renforcer les assertions de chaque composant par les garanties de ses 'voisins'. Aucune obligation de preuve n'est générée, les hypothèses utilisées pour prouver chaque composant sont simplement renforcées. Notre approche correspond plus à celle de Abadi-Lamport puisque nous tentons de prouver qu'un composant implémente bien son contrat. Une fois cette preuve assurée, on n'utilise plus le corps du composant pour prouver le reste de l'application.

9.2.2 Model-checking compositionnel

Dans les méthodologies que nous venons de présenter, il n'est pas précisé comment sont prouvés les contrats des sous-systèmes. Plusieurs travaux ont proposés d'appliquer ce découpage de la preuve d'un système directement au coeur des outils de model-checking.

Pnueli [Pnu84] puis Grumberg et Long [GL94][1] se sont intéressés à l'utilisation de règles de composition (comme celle présentée dans le travail de Abadi et Lamport) directement dans le model-checking de formules CTL ou de ses variantes.

Mocha (outil développé par Henzinger et al. [AHM+98, AdAG+01]) est un model-checker pour systèmes spécifiés à l'aide des *Reactive Modules* [AH96] dont nous avons déjà parlé au chapitre 5. Comme nous l'avons vu alors, les Reactive Modules ne permettent pas vraiment la séparation claire d'un *assume* et d'un *guarantee* depuis une spécification. Dans [HQR98], Henzinger et al. proposent

[1] Un nombre considérable de travaux ont porté, durant les 20 dernières années sur le model-checking compositionnel. Nous ne présentons ici que ceux qui nous paraissent les plus importants, de manières relativement subjective.

entre autres méthodes de vérification une règle *assume/guarantee* leur permettant d'assurer la correction d'une décomposition d'un système en sous-systèmes.

Dans [McM97], McMillan propose une règle de vérification pour le raffinement de designs matériels. Le raffinement consiste simplement à proposer les composants Si qui peuvent décomposer un composant S donné. On peut définir des « *refinement maps* » (voir [AL91]), c'est-à-dire un ensemble de règles qui permette de construire les spécifications des Si à partir de la spécification de S. Le travail présenté dans [McM97] nous intéresse car il porte sur la validation de systèmes matériels (donc possédant certaines contraintes en commun avec les programmes LUSTRE).

Dans [McM00], McMillan propose une méthodologie pour la vérification des systèmes matériels. Cette méthodologie repose sur un ensemble de techniques de preuve, parmi lesquelles, on trouve :

- l'utilisation de relations de raffinement (similaire à celles proposées dans [McM97] ;
- l'utilisation du principe de raisonnement *assume/guarantee* circulaire tel que vu plus haut ;
- l'utilisation de techniques de réduction de symétrie basées sur l'utilisation de scalarset (identiques à la méthode (évoquée au chapitre 8) proposée par Dill et Ip dans Murφ [ID93a]).

Ce raisonnement permet ainsi de prouver d'un côté que $S1$ satisfait $P1$ et que $S2$ satisfait $P2$ et d'en déduire que la composition de $P1$ avec $P2$ satisfait bien la conjonction de $P1$ avec $P2$.

[McM99] étend ces techniques (tout d'abord développées pour les systèmes matériels) aux systèmes d'états infinis. Dans [RM01] , il propose l'application de sa méthodologie à la vérification de micro-architectures et dans [McM01], il étudie la vérification d'un protocole de cohérence de cache.

9.2.3 Autres travaux

De nombreuses personnes se sont attachées à appliquer la vérification compositionnelle à différents types de systèmes. Kim Larsen et al. [JLS00, LL95] ont intégré des règles de preuve compositionnelle à l'outil Uppaal dédié à la vérification des systèmes temps-réels. Dans [ZS01], Zulkernine et al. proposent la spécification de contrôleurs par couples *assume/guarantee*. Le contrôle d'un système complexe est réalisé à l'aide de contrôleurs décrits pour chaque composant du système.

Des règles *assume/guarantee* ont été proposé pour des réseaux flots de données dans les travaux de Ketil Stølen [Stø96] ou de Manfred Broy [Bro98]. Ces travaux considèrent des composants similaires aux nœuds LUSTRE et proposent des contrats formées de deux propriétés : un *assume* portant sur l'histoire des valeurs d'entrées et un *guarantee* portant sur l'histoire des valeurs d'entrées *et* de sortie.

Les *Interface Automata* que nous avons présenté plus permettent aussi l'application du raisonnement *assume-guarantee*. Dans [dAH01a], la définition formelle des *Interface Automata* donnée par DeAlfaro et Henzinger permet notamment de définir des vérifications de compositions et d'implémentation (dans le cadre du raffinement) similaires à celle que nous proposons plus loin.

Dans [Hol00], Leszek Holenderski a proposé une méthodologie de vérification compositionnelle pour les réseaux synchrones. Holenderski propose des stratégies de preuve tout à fait similaires à notre approche : la composition de composants synchrones est décrite par une conjonction et les règles de preuve de validité d'une composition (essentiellement basées sur des manipulations purement logiques des formules *assume/guarantee*) sont similaires à celles que nous proposons au paragraphe 9.4.

Ces derniers travaux s'appliquent au même type de système que ceux développés en LUSTRE. Par ailleurs, ils donnent des définitions formelles de la composition similaire à la composition des composants LUSTRE ainsi qu'une notion de raffinement, proche de notre notion d'implémentabilité (voir paragraphe 9.4.1.1). Notons que ces deux notions sont toute fois différentes. Jan Mikac a étudié dans

son DEA à Verimag une notion de raffinement pour les programmes LUSTRE [Mik02]. Néanmoins, ils appliquent tous le raisonnement *assume-guarantee* (preuve compositionnelle) avec son aspect « *global* » : ils cherchent à déduire une propriété globale d'un ensemble de propriétés locales. Nous ne proposons pas de méthode de preuve compositionnelle globale, mais d'utiliser les contrats pour diverses vérification locales ainsi que pour vérifier la cohérence des compositions.

9.3 Débogage défensif

Dans [Var02], Marc Vareille a étudié l'interprétation des contrats dans le débogueur LUDIC de Fabien Gaucher. Le fonctionnement de LUDIC a rapidement été présenté au paragraphe 2.3.5.

Les contrats n'était pas prévus dans la conception originale de LUDIC. Le travail de M. Vareille a donc tout d'abord consisté à rendre possible leur prise en compte.

Il a fallu définir un format pour l'association des nœuds *assume/guarantee* du contrat d'un nœud N au nœud N lui-même. Dans ce format, on donne les nœuds *assume/guarantee* et les nœuds décrivant l'implémentation des corps séparément. On spécifie l'association de ces nœuds entre eux à l'aide d'un fichier « *lien* ».

Ce format est ensuite traduit en un programme LUSTRE interprétable par LUDIC où les informations du fichier *lien* en variables intermédiaires définies à l'aide d'appel aux nœuds *assume* et *guarantee*. L'analyseur syntaxique de LUDIC a enfin été modifié pour prendre en compte ces variables locales : celles-ci ne sont pas utilisées pour calculer une sortie du nœud N et sont donc normalement oubliées par LUDIC.

M. Vareille a ensuite étendu LUDIC selon deux approches. L'algorithme de « *débogage algorithmique* » implémenté dans LUDIC consiste à parcourir le graphe d'appel des nœuds d'un programme pour déterminer quel nœud est la source d'un bogue. Lorsqu'il trouve un appel de nœud, l'algorithme demande à l'utilisateur si les valeurs qu'il a calculé à l'instant où le bogue a été découvert sont correctes ou pas. M. Vareille a proposé de remplacer l'oracle humain par une évaluation à la volée de l'expression booléenne assume $\Rightarrow$ guarantee.

La seconde proposition de M. Vareille tente de répondre à la question souvent posé dans le domaine de l'interprétation des contrats : « *lorsqu'un contrat est violé, quel composant doit être incriminé : l'appelant ou l'appelé ?* » :

- Si une clause assume d'un nœud N est fausse, on regarde si la clause *assume* du nœud *appelant* N est fausse aussi, auquel cas, on reporte la faute sur lui. Si la clause *assume* du nœud appelant N est vraie, le bogue vient nécessairement de l'instanciation de N ;
- Si une garantie d'un nœud M est violée, on évalue les garanties des nœuds appelés par M. Si une de ces garanties est violée, l'erreur doit être attribuée au nœud appelant correspondant. Si aucune garantie appelée n'est violée, c'est M qui est fautif (qui viole son propre contrat).

9.4 Validation

Nous décrivons maintenant les différentes vérifications possibles autour des contrats. On distingue deux grandes catégories : les vérifications *locales* et *globales*.

- Au niveau *local*, on va s'intéresser à la validité du contrat considéré indépendamment de l'utilisation qui en est faite : le contrat est-il implémentable ? l'implémentation qu'on en a donné est-elle valide (c'est-à-dire réalise-t-elle bien le contrat donné) ? etc. ;
- Au niveau *global*, on va s'intéresser à la validité du contrat dans son environnement. Est-il possible de composer deux composants d'après leurs contrats ? Peut-on utiliser un composant donné

dans la description d'un composant plus gros ?

Dans les paragraphes suivants, nous décrirons ces vérifications en terme de step-relations. Pour chaque vérification, on obtient une *obligation de preuve* que nous pouvons exprimer directement en LUSTRE. Cela nous permet de profiter de toute la puissance des outils de vérification dédiés à LUSTRE.

Typiquement, chaque vérification décrite ci-dessous va être effectuée en deux étapes :

- On commence par générer un observateur LUSTRE représentant la propriété à vérifier ;
- On utilise par exemple un model-checker pour vérifier cette propriété.

Ces vérifications s'inscrivent dans la méthodologie de développement que nous préconisons au paragraphe 9.5. Les vérifications locales font en quelques sortes office de validation *unitaire* des composants. Cette étape de validation sera mise en place très tôt dans le processus de développement.

Les vérifications globales permettent une validation *d'intégration* des composants. Cette étape peut être entamée une fois que les composants les plus simples ont été validés : on vérifie la composition des contrats. On propage les contraintes apportées par des contrats de composants appelant vers les contrats des composants qu'ils utilisent, afin d'évaluer leur compatibilité.

9.4.1 Vérifications locales

On propose principalement deux vérifications locales. La première permet de vérifier qu'un contrat donné *peut* être implémenté. La seconde permet de vérifier qu'un corps de composant implémente bien le contrat associé.

9.4.1.1 Implémentabilité d'un contrat

Dans la suite, on utilise les notations introduites au paragraphe 3.4.6.3, page 59.

Définition 25 — Un contrat est-il implémentable ?

Soit un composant $\mathcal{C} = (I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$, on dit que le contrat spécifié par (Ja, Sa) et (Jg, Sg) est implémentable *(c'est-à-dire qu'il existe au moins un corps le satisfaisant) si et seulement si la condition suivante est vérifiée :*

$$(Sa, Ja)\lfloor I \rfloor \lceil I \cup O \rceil \subseteq (Sg, Jg)\lfloor I \cup O \rfloor \tag{9.1}$$

La condition 9.1 signifie simplement que les comportements autorisés par l'assertion du contrat doivent aussi être autorisés par la garantie.

Les traces décrites par (Sa, Ja) sont des traces sur les variables de $I \cup A$, alors que celle décrites par (Sg, Jg) sont sur $I \cup O \cup G$. On ne peut donc pas les comparer directement. On compare donc l'ensemble des comportements sur les variables $I \cup O$ autorisés par (Sa, Ja) (c'est-à-dire où les variables de O peuvent prendre n'importe qu'elle valeur) et les comportements sur $I \cup O$ autorisés par (Sg, Jg).

Notons que la notion d'implémentabilité n'est pas comparable avec celle de cohérence présentée au paragraphe 5.3.4. Les deux exemples suivants l'illustrent.

EXEMPLE 39 — Considérons un composant spécifié par le contrat suivant :

- $I = \{i\}$ et $O = \{o\}$;
- $Ja = Jg = true$;
- $Sa = i \geq 0$;
- $Sg = (o \geq 2 \times i) \wedge (o \leq 100)$.

Sg implique une contrainte sur i ($i < 50$) qui est plus forte que Sa. Ce contrat est implémentable : on peut construire un composant qui prend un i dans $[0, 50]$ et rend une sortie o dans $[0, 100]$. Mais il n'est pas cohérent puisque $i \geq 0 \not\subseteq (\exists o.(o \geq 2 \times i) \wedge (o \leq 100))$.

— FIN DE L'EXEMPLE 39

EXEMPLE 40 — Considérons maintenant un composant spécifié par :
- $I = \{i\}$ et $O = \{o\}$;
- $Ja = Jg = true$;
- $Sa = true$;
- $Sg = (o > 50) \wedge (o \leq 49)$.

Ce contrat est cohérent (puisque il n'y a simplement pas de contrainte exprimée sur i dans Sa, ni dans Sg. Par contre, ce contrat n'est pas implémentable, puisque la contrainte Sg est contradictoire.

— FIN DE L'EXEMPLE 40

Traduction en obligation de preuve LUSTRE – Soit N un nœud possédant l'interface :

node N(I) **returns** (O);

Son contrat est spécifié par les nœuds assumeN et guaranteeN qui ont pour interface :

node assumeN(I) **returns** (assumeOK : bool);

node guaranteeN(I,O) **returns** (guaranteeOK : bool);

Pour prouver que le contrat de N est implémentable, on vérifie que l'ensemble des comportements autorisés par l'assertion est contenu dans l'ensemble des comportements autorisés par la garantie. On génère pour cela l'observateur suivant :

```
node contractEstImplemetable(I) returns (ok : bool);
var O;
let
   ok = assumeN(I) => guaranteeN(I,O);
tel
```

Notons que ce nœud possède une variable locale indéfinie. Celle-ci signifie que l'on souhaite en fait montrer que, pour chaque valeur de I, il est possible de trouver au moins *une* valeur pour la variable O. Comme nous l'avons déjà souligné, un tel observateur ne pourra pas être fourni tel que à un model-checker. On pourra par contre envisager (voir perspectives, chapitre 13) de le simuler.

9.4.1.2 Vérifier qu'un nœud implémente son contrat

Une fois qu'une implémentation pour un composant est connue, on va vérifier que cette implémentation est valide vis-à-vis du contrat du composant. Pour cela, on va montrer que lorsqu'on fournit au composant des valeurs d'entrées autorisées par l'assertion du contrat, alors le composant produit bien des sorties qui sont conformes à la garantie du contrat.

Définition 26 — Un corps de composant satisfait-il son contrat ?

> *Soit un composant* $\mathcal{C} = (I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$*, On dit que le corps de ce composant, spécifié par (*Jb*,*Sb*) implémente le contrat spécifié par* (Ja, Sa) *et* (Jg, Sg) *si et seulement si, pour chaque trace d'entrées satisfaisant* (Ja, Sa) *toutes les traces de sorties satisfaisant* (Jb, Sb) *satisfont aussi* (Jg, Sg)*. c'est-à-dire, si et seulement si :*
>
> $$(Sa, Ja)\lfloor I \rfloor \lceil I \cup O \rceil \cap (Sb, Jb)\lfloor I \cup O \rfloor \subseteq (Sg, Jg)\lfloor I \cup O \rfloor$$

Traduction en obligation de preuve LUSTRE – Soit N un nœud possédant l'interface :

node N(I) **returns** (O);

Son contrat est spécifié par les nœuds assumeN et guaranteeN qui ont pour interface :

node assumeN(I) **returns** (assumeOK : bool);

node guaranteeN(I,O) **returns** (assumeOK : bool);

Pour prouver que N implémente bien son contrat, on vérifie que lorsqu'on fournit à N des entrées pour lesquelles assumeN répond vrai, alors N produit des sorties pour lesquelles guaranteeN répond vrai.

```
node contratEstSatisfait(I) returns (ok : bool);
var O;
let
   O = N(I)
   ok = assumeN(I) => guarantee(I,O);
tel
```

EXEMPLE 41 — On considère le composant rMFF décrit au paragraphe 5.3.1. Nous avons considéré un contrat constitué :

- d'une assertion stipulant que l'entrée du composant est toujours vraie pendant au moins 2 instants consécutifs ;
- d'une garantie stipulant que si la sortie du composant est toujours vraie pendant au moins 3 instants consécutifs.

Chacune de ces propriétés était spécifiée par un automate. Les nœuds LUSTRE correspondant était donnés à la figure 5.14. Nous avons aussi proposé un automate pour une implémentation de ce contrat (voir figure 5.8). Le nœud LUSTRE correspondant est donné à la figure 9.1. Pour vérifier que rMFF implémente bien son contrat, on génère l'obligation de preuve représentée par le nœud lustre verifContract de la figure 9.2. Cet observateur est ensuite envoyé (par exemple) au model-checker *lesar* qui, dans ce cas, répond « *propriété vraie* ».

— FIN DE L'EXEMPLE 41

9.4.2 Vérifications globales

On appelle « vérifications globales » les propriétés que l'on souhaite vérifier lorsque des composants sont connectés les uns aux autres.

```
node rMFF(a : bool) returns (b : bool);
var q0, q1, q2 : bool;
let
   q0 = true -> (pre(q0) and not pre(a)) or (pre(q2) and not pre(a));
   q1 = false -> (pre(q0) and pre(a)) or (pre(q1) and pre(a)) or (pre(q2) and pre(a));
   q2 = false -> (pre(q1) and not pre(a));
   b = false -> pre(q1) or pre(q2);
tel
```

FIG. 9.1 – Le nœud LUSTRE rMFF.

```
node verifContract(a : bool) returns (contractOK : bool);
var b : bool;
     assume, guarantee : bool;
let
   b = rMFF(a);
   assume = assumeRMFF(a);
   guarantee = guaranteeRMFF(b);
   contractOK = assume => guarantee;
tel
```

FIG. 9.2 – Obligation de preuve construite pour vérifier l'implémentation rMFF.

On propose principalement trois vérifications globales. La première permet de vérifier que deux composants sont compatibles, c'est à dire qu'il peuvent être composé. La seconde permet de la cohérence d'une décomposition d'un composant en sous-composants. La troisième vérification est utilisée dans le cadre des itérations : on va vérifier que le contrat du nœud itéré autorise bien les connexions induites par l'itération.

9.4.2.1 Compatibilité de deux composants

Au chapitre 5, nous avons présenté la composition synchrone de composants. Or, avant même de composer deux composants, on peut se poser la question de savoir si les contrats correspondants « autorisent » cette composition. Le cas le plus simple est celui d'une composition purement séquentielle où il n'y a pas de rebouclage de définition.

EXEMPLE 42 — Considérons deux nœuds C1 et C2 possédant les interfaces suivantes :

node C1(i1 : int) **returns** (o1, o2 : int);
node C2(i2, i3 : int) **returns** (o3 : int);

Un contrat est associé à chacun des nœuds. Nous ne donnons que la garantie de C1 :

$$G1 : o1 > 0 \text{ and } o2 <= 0$$

et l'assertion de C2 :

$$A2 : i2 \leq 0 \text{ and } i3 > 0$$

Admettons que l'on souhaite composer C1 et C2 en identifiant :
- o1 avec i2 ;
- o2 avec i3 ;

Pour vérifier que les contrats de C1 et C2 autorisent cette composition, on va devoir vérifier que G1 $\Rightarrow$ A2 en identifiant les variables concernées par la composition comme indiqué ci-dessus. On doit donc montrer que :

$$\mathsf{o1} > 0 \text{ and } \mathsf{o2} \leq 0 \Rightarrow \mathsf{o1} \leq 0 \text{ and } \mathsf{o2} > 0;$$

qui est trivialement fausse. Si maintenant on veut composer C1 et C2 en identifiant :
- o1 avec i3 ;
- o2 avec i2 ;

on doit vérifier que :

$$\mathsf{o1} > 0 \text{ and } \mathsf{o2} \leq 0 \Rightarrow \mathsf{o2} \leq 0 \text{ and } \mathsf{o1} > 0;$$

qui est trivialement vraie. Ainsi, la deuxième composition est possible tandis que la première ne l'est pas.

— FIN DE L'EXEMPLE 42

En général, la composition de deux composants peut impliquer des rebouclages : une entrée de C1 peut dépendre d'une sortie de C2 et une entrée de C2 peut en même temps dépendre d'une sortie de C1. C'était le cas notamment de la composition présentée au paragraphe 5.3.5 (figure 5.13, page 5.13).

On donne maintenant une règle permettant de vérifier que les contrats de deux composants qu'on compose sont *compatibles*. Nous appelons *variables de branchements* les variables de C1 et C2 qui servent à composer les deux composants. Comme nous l'avons vu dans la composition, on suppose que les *variables de branchements* portent des noms identiques dans les deux composants. Dans la pratique, ce ne sera pas nécessairement le cas. On suppose que les composants sont branchés *directement*, c'est-à-dire sans composant intermédiaire.

On dit que C1 et C2 peuvent être composés seulement si la garantie de C1 (resp. de C2) ne contraint pas les variables de branchements que l'assertion A2 (resp. de A1) reçoit en entrée, plus que A2 (resp. A1) elle-même.

Définition 27 — Compatibilité de deux Composants

On dit que 2 composants $\mathcal{C}_i = (I_i, O_i, L_i, A_i, G_i, Sa_i, Ja_i, Sg_i, Jg_i, Sb_i, Jb_i)$ (avec $i = 1, 2$), sont compatibles pour la composition $\mathcal{C}_1 \times \mathcal{C}_2$ définie au paragraphe 5.3.5 si :

$$(Sg_2, Jg_2)\lfloor O2 \cap I1 \rfloor \subseteq (Sa_1, Ja_1)\lfloor I1 \cap O2 \rfloor$$

et

$$(Sg_1, Jg_1)\lfloor O1 \cap I2 \rfloor \subseteq (Sa_2, Ja_2)\lfloor I2 \cap O1 \rfloor$$

Cette notion de compatibilité ne nous permet pas de déclarer que deux composants ne « *peuvent* » pas être connectés l'un à l'autre, mais simplement d'avoir une plus grande confiance dans la composition, en cas de réponse positive. Considérons par exemple deux composants branchés en séquence avec une variable x (voir figure 9.3). Si la garantie G_1 du premier composant n'implique pas l'assertion A_2 du second composant, on peut prévenir le développeur que le producteur de x ne suffit pas à assurer les contraintes du second composant. La validité de A_2 devra donc venir du contexte

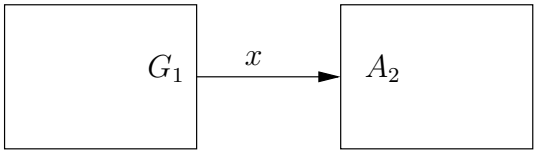


FIG. 9.3 – Exemple de composition.

d'utilisation. Si la compatibilité peut être assurée, on sait que les contrats des deux composants seront respectés dans un environnement satisfaisant le contrat de la composition.

EXEMPLE 43 — À la figure 5.13 (paragraphe 5.3.5), nous avons donné la composition de deux composants $\mathcal{C}_1$ et $\mathcal{C}_2$ définis par :

$$
\begin{array}{l}
Ja_1(i_1, j_1) \ : true \\
Sa_1(i_1, j_1) \ : j'_1 \geq 0 \vee (j'_1 < 0 \wedge i'_1 < 0); \\
Jg_1(i_1, j_1, o_1, j_2) \ : true \\
Sg_1(i_1, j_1, o_1, j_2) \ : j'_2 > 0 \wedge o'_1 > 0. \\
Ja_2(i_2, j_2) \ : true \\
Sa_2(i_2, j_2) \ : j'_2 > 0 \vee (j'_2 \leq 0 \wedge i'_2 < j'_2); \\
Jg_2(i_2, j_2, o_2, j_1) \ : true \\
Sg_2(i_2, j_2, o_2, j_1) \ : j'_1 > 0 \wedge o'_2 > j'_1.
\end{array}
$$

Pour vérifier que les deux composants sont compatibles pour cette composition, on va générer une obligation de preuve vérifiant que :

- la partie de la garantie de $\mathcal{C}_1$ portant sur $j2$ implique bien la partie de l'assertion de $\mathcal{C}_2$ portant sur $j2$, c'est-à-dire que :
$$(\exists i_2, o_2 . j'_1 > 0 \wedge o'_2 > j'_1) \Rightarrow (\exists i_1 . j_1 \geq 0 \vee (j_1 < 0 \wedge i'_1 < 0)) ;$$
- la partie de la garantie de $\mathcal{C}_2$ portant sur $j1$ implique bien la partie de l'assertion de $\mathcal{C}_1$ portant sur $j1$, c'est-à-dire que :
$$(\exists o_1, i_2, i_1 . \ j'_1 > 0 \wedge o'_2 > j'_1) \Rightarrow (\exists i_2 . j'_2 > 0 \vee (j'_2 \leq 0 \wedge i'_2 < j'_2)) ;$$

— **FIN DE L'EXEMPLE 43**

Traduction en obligation de preuve LUSTRE **–** Dans l'exemple précédent, on obtient des contraintes existentielles, du fait qu'on « cache » les variables qui ne sont pas utilisées dans la connexion. Comme nous l'avons déjà dit plus haut, ces contraintes ne vont pas pouvoir être exprimées par un observateur LUSTRE classique : ces variables quantifiées existentiellement correspondraient à des variables locales indéfinies. On peut par contre toujours décrire ces contraintes par un observateur à variables locales indéfinies dont la sémantique est donnée au paragraphe 3.4.3.

Cela implique une limitation sur les possibilités d'exploitation de ces observateurs. On ne pourra notamment pas les fournir tels quels aux outils de vérification (puisque ceux-ci utilisent des observateurs déterministes).

La génération d'observateurs LUSTRE (qui nous concerne spécialement pour l'instant) est aussi simple que dans les cas précédents. Nous en donnons un exemple ci-dessous. Par contre, l'utilisation qui peut être faite de ces observateurs est différente. Nous reviendrons plus loin sur ce point.

EXEMPLE 44 — Nous considérons à nouveau la composition décrite à la figure 5.13. Nous avons exprimé ci-dessus les contraintes permettant de vérifier que la composition de $\mathcal{C}_1$ et $\mathcal{C}_2$ est possible

au vu de leur contrat. La figure 9.4 donne un observateur à variables locales indéfinies permettant de vérifier la compatibilité des contrats de $\mathcal{C}_1$ et $\mathcal{C}_2$. Nous utilisons deux variables intermédiaires pour faciliter la lecture de la propriété. branch1OK représente le fait que la garantie de $\mathcal{C}_2$ implique l'assertion de $\mathcal{C}_1$ vis à vis de la variable de branchement j_1. branch2OK représente le fait que la garantie de $\mathcal{C}_2$ implique l'assertion de $\mathcal{C}_2$ vis à vis de la variable de branchement j_2. On notera la comparaison utilisant deux valeurs successives de j_1 (j1 et pre(j1)) correspondant à la présence de l'opérateur pre dans la composition.

— FIN DE L'EXEMPLE 44

```
node verifComposition(j1,j2 : int) returns (compoOK : bool);
var i1,o1,i2,o2 : int;
    branch1OK, branch2OK : bool;
let
   compoOK = branch1OK and branch2OK;
   branch1OK = true -> guaranteeC2(i2,j2,o2,j1) => assumeC1(i1,pre(j1));
   branch2OK = true -> guaranteeC1(i1,pre(j1),o1,j2) => assumeC2(i2,j2);
tel
```

FIG. 9.4 – Vérification de la compatibilité de $\mathcal{C}_1$ et $\mathcal{C}_2$.

9.4.2.2 Propagation des contraintes exprimées par un contrat

On s'intéresse ici à la propagation des contraintes exprimées par des contrats à l'intérieur de nœuds. Considérons par exemple un composant N dont le contrat est exprimé par un couple de nœuds assumeN et guaranteeN. N utilise éventuellement un nœud M pour lequel on a aussi défini un contrat par les nœuds assumeM et guaranteeM.

Les branchements de M avec d'autres composants à l'intérieur de N peuvent être étudiés grâce aux propriétés que nous avons évoquées portant sur la compatibilité de deux composants. Par contre, on peut aussi vouloir « comparer » les contrats de M et N.

Imaginons (figure 9.5) qu'une variable d'entrée i de N soit directement utilisée comme entrée de M. Les contrats de N et M représentent tous les deux des contraintes sur i et il paraît intéressant de « comparer » ces contraintes, par exemple pour savoir si assumeN est une contrainte assez forte pour que l'on soit sûr que M est utilisé avec des valeurs pour l'entrée i acceptables. On pourra utiliser ce type de comparaison pour évaluer si des valeurs potentiellement produites par M (acceptées par sa garantie) peuvent violer la garantie de N. De manière générale, cela permet de se faire une idée partielle de la bonne utilisation d'un composant par son environnement direct.

On s'intéresse à la comparaison des contraintes exprimées dans les clauses *assume* et *guarantee* des deux contrats. On propose deux formes de *propagation* de contraintes : une version « *ascendante* » et une version « *descendante* ». Dans le premier cas, on va propager les contraintes exprimées sur i par M dans le contrat de N. Dans le second cas, on va propager les contraintes exprimées sur i par N dans le contrat de M. On pourra ensuite vérifier, par exemple, que le contrat ainsi obtenu est toujours implémentable (au sens de ce que nous avons décrit au paragraphe 9.4.1.1). Nous donnons maintenant les définitions des propagations ascendantes et descendantes d'un contrat.

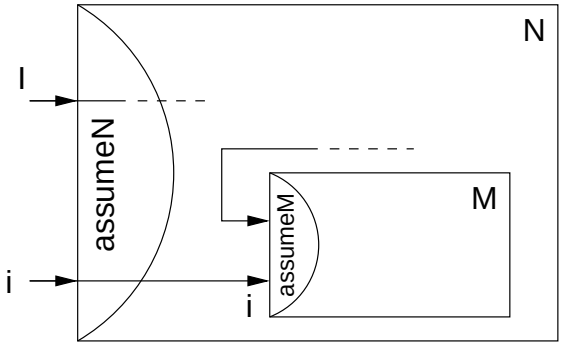


FIG. 9.5 – Illustration de la propagation de contraintes.

Définition 28 — Propagation ascendante d'un contrat

Soient 2 composants $\mathcal{C}_i = (I_i, O_i, L_i, A_i, G_i, Sa_i, Ja_i, Sg_i, Jg_i, Sb_i, Jb_i)$ *(avec* $i = 1, 2$*) tels que* $\mathcal{C}_1$ *utilise* $\mathcal{C}_2$*. La propagation ascendante du contrat de* $\mathcal{C}_2$ *vers* $\mathcal{C}_1$*, notée* $\mathcal{C}_{1\uparrow\mathcal{C}_2}$ *est définie par :*

$$\begin{aligned}\mathcal{C}_{1\uparrow\mathcal{C}_2} = (&I_1, O_1, L_1,\\ &A_1 \cup (I_2 \setminus I_1),\\ &G_1 \cup (I_2 \setminus I_1) \cup (O_2 \setminus O_1),\\ &Sa_1 \cap Sa_2\lfloor I_2 \cap I_1\rfloor,\\ &Ja_1 \cap Ja_2\lfloor I_2 \cap I1\rfloor,\\ &Sg_1 \cap Sg_2\lfloor (I_2 \cap I1) \cup (O_2 \cap O_1)\rfloor,\\ &Jg_1 \cap Jg_2\lfloor (I_2 \cap I_1) \cup (O_2 \cap O_1)\rfloor,\\ &Sb_1, Jb_1)\end{aligned}$$

$\mathcal{C}_{1\uparrow\mathcal{C}_2}$ est appelé propagation ascendante puisqu'il représente le composant appelant ($\mathcal{C}_1$) dont on a renforcé le contrat avec le contrat d'un nœud appelé ($\mathcal{C}_2$). Notons que la propagation ne concerne pas l'implémentation de $\mathcal{C}_1$ (qui n'est donc pas modifiée) mais uniquement son contrat. Inversement, on propose la propagation descendante du contrat de $\mathcal{C}_2$ par celui de $\mathcal{C}_1$. Ces définitions sont illustrées par l'exemple suivant.

Définition 29 — Propagation descendante d'un contrat

Soient 2 composants $\mathcal{C}_i = (I_i, O_i, L_i, A_i, G_i, Sa_i, Ja_i, Sg_i, Jg_i, Sb_i, Jb_i)$ *(avec* $i = 1, 2$*) tels que* $\mathcal{C}_1$ *utilise* $\mathcal{C}_2$*. Le propagation descendante du contrat de* $\mathcal{C}_1$ *vers* $\mathcal{C}_2$*, notée* $\mathcal{C}_{2\downarrow\mathcal{C}_1}$ *est définie par :*

$$\begin{aligned}\mathcal{C}_{2\downarrow\mathcal{C}_1} = (&I_2, O_2, L_2,\\ &A_2 \cup (I_1 \setminus I_2),\\ &G_1 \cup (I_1 \setminus I_2) \cup (O_1 \setminus O_2),\\ &Sa_2 \cap Sa_1\lfloor I_1 \cap I2\rfloor,\\ &Ja_2 \cap Ja_1\lfloor I_1 \cap I2\rfloor,\\ &Sg_2 \cap Sg_1\lfloor (I_1 \cap I_2) \cup (O_1 \cap O_2)\rfloor,\\ &Jg_2 \cap Jg_1\lfloor (I_1 \cap I_2) \cup (O_1 \cap O_2)\rfloor,\\ &Sb_2, Jb_2)\end{aligned}$$

EXEMPLE 45 — Considérons les nœuds M et N composés comme décrit par le schéma de la figure 9.6 et spécifiés par les contrats assumeM, guaranteeM et assumeN, guaranteeN donnés à la figure 9.7.

— FIN DE L'EXEMPLE 45

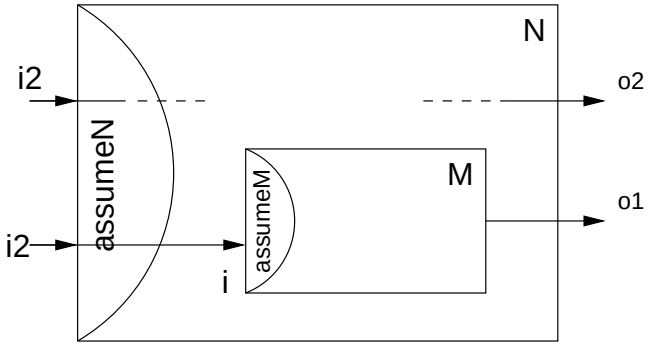


FIG. 9.6 – Une utilisation d'un nœud N par un nœud M.

```
node assumeM(i1, i2 : int) returns (assumeOK : bool);
let
   assumeOK = i1 < 0 and i2>0;
tel

node guaranteeM(i1, i2, o1, o2 : int) returns (guaranteeOK : bool);
let
   guaranteeOK = o1>0 and o2<0;
tel

node assumeN(i : int) returns (assumeOK : bool);
let
   assumeOK = i<=0;
tel

node guaranteeN(i, o : int) returns (guaranteeOK : bool);
let
   guaranteeOK = o>=0;
tel
```

FIG. 9.7 – Les contrats des composants M et N de la figure 9.6.

Affaiblissement/Renforcement d'un contrat – A plusieurs moments dans la validation d'une application on peut renforcer ou affaiblir le contrat d'un composant. Soit en utilisant la propagation de contraintes que nous venons de décrire, soit simplement parce qu'on souhaite raffiner (ou généraliser) son comportement. A chaque fois il faudra s'assurer la validité du nouveau contrat par rapport à son implémentation.

Deux cas particuliers peuvent être traités très facilement. Si on renforce l'assertion d'un contrat dont on a prouvé la validité, celle-ci est conservée pour le nouveau contrat. Un composant qui suppose un ensemble de comportements (représenté par une contrainte A) sur ses entrées pour fonctionner correctement, fonctionnera correctement aussi si il est placé dans un environnement représenté par une contrainte A' telle que A'⊆A). Un exemple trivial est celui où A n'exprime aucune contrainte sur les entrées du composant (A = true). On peut placer le composant dans un environnement n'importe quelle contrainte A', il fonctionnera de la même façon.

De manière symétrique, on peut sans problème affaiblir la garantir G en la remplaçant par une contrainte G' telle que G⊆G'. Par exemple, si les comportements des sorties du composant satisfont G alors ils satisfont aussi une G' si G' = true. Donc, si on *renforce l'assertion* ou que l'on *affaiblit la garantie* d'un contrat, il n'est pas nécessaire de re-valider l'implémentation du contrat vis-à-vis du nouveau contrat. Dans tous les autres cas (affaiblissement de l'assertion, renforcement de la garantie), on devra à nouveau prouver la validité du contrat.

9.4.2.3 Manipulations des contrats dans le contexte des itérations

Nous avons présenté au paragraphe 5.3.9 comment on peut calculer le contrat d'un composant « *itération* » à partir du contrat du composant itéré. Le contrat d'une itération de taille n est construit itérativement en composant des contrats 1) du composant itéré 2) d'une itération de taille $n-1$. Le contrat d'une itération de taille 1 est celui du composant itéré lui-même.

Or, lors d'une composition de composant à contrat, on a proposé (paragraphe précédent) de vérifier que cette composition était possible « *vis-à-vis des contrats des composants que l'on compose* ». Dans le cas d'une itération, on doit pouvoir vérifier que le composant utilisé est bien *itérable* c'est-à-dire que les contraintes représentées par son contrat sont bien compatibles avec la composition imposée par l'itération.

Vérifier qu'un contrat est itérable – Dans le cas d'une itération *map*, aucune vérification n'est nécessaire, puisque les instances du nœud itéré ne sont pas connectées. Dans le cas des autres itérations, par contre, on devra s'assurer que les contraintes imposées par la garantie du nœud itéré sur sa sortie *accumulateur* et les contraintes imposées par son assertion sur son entrée *accumulateur* sont *compatibles*. Ces variables accumulateurs sont en effet les seules variables utilisées pour connecter les différentes instances du composant entre elles. Il suffit de vérifier une composition de deux instances du nœud itéré, puisque les branchements dans l'itération sont tous identiques. Nous donnons maintenant une définition formelle de cette notion. Ensuite, nous montrons comment cette propriété peut-être traduite en un observateur LUSTRE.

Définition 30 — Un contrat est-il itérable ?

Soit 1 composant $\mathcal{C} = (I, O, L, A, G, Sa, Ja, Sg, Jg, Sb, Jb)$, tel que $I = \{a_{in}\}$ et $O = \{a_{out}\} \cup f$ (où e (resp. f) représente les entrées (resp. sorties) correspondant aux éléments de tableaux et a_{out} correspond à l'accumulateur de sortie et soit s un entier quelconque. on dit que l'itération $iter(\mathcal{C}, s)$ (où $iter \in \{red, fill, map\_red\}$) définie au paragraphe 5.3.9 est possible vis-à-vis du contrat de $\mathcal{C}$ si :

$$((Sg, Jg)\lfloor\{a\_out\}\rfloor)_{[a\_out \leftarrow a]} \subseteq ((Sa, Ja)\lfloor I1 \cap \{a\_in\}\rfloor)_{[a\_in \leftarrow a]}$$

Traduction en obligation de preuve LUSTRE – Soit N un nœud possédant l'interface :

node N(a_in, I) **returns** (a_out, O);

Son contrat est spécifié par les nœuds assumeN et guaranteeN qui ont pour interface :

```
node assumeN(a_in, I) returns (assumeOK : bool);

node guaranteeN(a_in, I, a_out, O) returns (assumeOK : bool);
```

Pour prouver que le nœud N est itérable, on vérifie que l'ensemble des comportements de la variable accumulateur de sortie (la variable a_out) autorisés par la garantie du contrat sont tous des comportements autorisés pour la variable accumulateur d'entrée (a_in) par l'assertion de N. On génère ainsi l'observateur suivant :

```
node contractEstIterable(a_out) returns (ok : bool);
var I,O, a_in
let
   ok = guaranteeN(a_in, I, a_out, O) => assumeN(a_out, I);
tel
```

9.5 Méthodologie de développement

Dans le chapitre 5, nous avons montré l'intérêt d'une méthodologie de spécification par contrat dans la description des systèmes. Les manipulations que nous avons proposées ci-dessus s'inscrivent dans une approche de validation par morceaux des systèmes et de manière générale de développement de programmes corrects.

On va notamment pouvoir les utiliser à tous les stades du développement d'une application afin de valider les choix de composants. On utilisera pour cela les techniques que nous avons présentés dans ce chapitre.

La principale utilisation que nous souhaitons explorer est l'exécution précoce de composants non-déterministes décrits seulement par leur contrat. L'exécutabilité des spécifications est alors primordiale. L'un des avantages de la spécification par contrat est qu'un réseau complexe de composants peut être exécuté (simulé) très tôt dans le processus de développement, avant même que tous les composants n'aient été programmés. Pour chaque composant décrit par un contrat, les valeurs des sorties ne peuvent pas être calculées de manière déterministe en fonction des entrées. Étant données ces valeurs d'entrées, il faut générer des valeurs de sorties compatibles avec le contrat.

Cette génération automatique de sorties est parfaitement définissable grâce à la définition formelle que nous avons des clauses assume et garantie. Nous reviendrons sur cette approche dans les perspectives, chapitre 13.

Chapitre 10

Retour sur Exemples

Dans ce chapitre, nous revenons sur les exemples des chapitres 6 et 7. Nous montrons comment appliquer les méthodes proposées aux chapitres 8 et 9 et le gain apporté.

10.1 Gyroscope

Au chapitre 6, nous avons présenté l'application du Gyroscope. Nous avons montré le découpage de cette application en plusieurs composants principaux décrits d'abord à l'aide de contrats avant d'être implémentés par des noeuds LUSTRE. Nous allons maintenant voir comment les techniques présentées aux chapitres 8 et 9 peuvent être utilisées pour aider à valider cette application.

Concernant la manipulation des contrats, nous allons tout d'abord valider les implémentations vis-à-vis des contrats. Ensuite, nous montrerons les vérifications à mettre en place lors de la composition des composants de base et illustrerons l'implication que peut avoir cette phase de validation sur le développement des composants. Concernant la manipulation des itérations, nous montrerons comment appliquer les techniques proposées sur l'itération du nœud EvaluateAxis.

10.1.1 Manipulation des contrats

Nous commençons par étudier les composants Channel et Voter. Pour valider les contrats de ces deux composants, nous avons généré des observateurs en suivant la proposition du paragraphe 9.4.1.2. Ensuite, nous étudierons la connexion de ces composants au sein du nœud EvaluateAxis.

10.1.1.1 Validation du nœud Channel

Comme nous l'avons déjà souligné au paragraphe 6.7, la spécification du Channel et sa validation ont été progressives : nous avons validé différentes versions du contrat avec les implémentations correspondantes. En suivant la méthode du paragraphe 9.4.1.2, on génère à chaque fois un observateur de la forme suivante :

```
node isContractFor_g_Channel_Fulfilled(inChannel : Faulty_ChannelT;
                                       delta : int; ideal : int;
                                       delta_to_ideal : int)
returns (contractIsFulfilled : bool);
var outChannel : Valid_ChannelT;
    assume_isValid : bool;
    guarantee_isValid : bool;
let
  outChannel = Channel(inChannel, delta, ideal, delta_to_ideal);
  assume_isValid = assumeChannel(inChannel, delta, ideal, delta_to_ideal);
  guarantee_isValid = guaranteeChannel(inChannel, delta, ideal,
                                       delta_to_ideal, outChannel);
  contractIsFulfilled = ((assume_isValid) => (guarantee_isValid));
tel
```

La première version ne prenait pas en compte la propagation temporelle des erreurs ni la détection des fautes croisées. Nous avons écrit une version du Channel qui n'utilise pas Maintain ni Cross-FailDetect. Le contrat correspondant est plus simple que celui donné au chapitre 6. Il stipule que :

- Soit une erreur locale est détectée par le channel (ici, cette erreur est instantanée) ;
- Soit la valeur courante de la sortie n'est pas trop loin des valeurs courantes des entrées.

La seconde version que nous avons développée prend en compte les erreurs de capteurs. Elle utilise donc le nœud Maintain. Le contrat utilisé est celui donné au paragraphe 6.4.3.

La dernière version est celle que nous avons présentée au chapitre 6. Elle prend en compte à la fois les erreurs de capteurs *et* les erreurs de liaisons. Le contrat du Channel pour cette nouvelle version est légèrement modifié. La garantie est identique, mais l'assertion est renforcée : on suppose maintenant que parmi les trois channels voisins du channel qu'on traite, au moins 1 n'a pas émis d'erreur locale à l'instant précédent. Comme l'assertion du contrat a été renforcée, celui-ci reste valide et ne doit pas être vérifiée à nouveau (voir paragraphe 9.4.1.2).

Les trois versions du Channel ont été validées à l'aide de l'outil nbac [Jea00]. Cet outil est bien adapté à pour cette preuve car le contrat de Channel manipule des variables numériques. Les model-checkers classiques abstraient ces variables et ne sont pas capables de prouver la propriété.

10.1.1.2 Validation du nœud Voter

Pour le Voteur, nous n'avons développé qu'une version, donnée au paragraphe 6.4.4. L'observateur généré pour prouver la compatibilité de l'implémentation avec le contrat est le suivant :

```
node isContractFor_Voter_Fulfilled(channels : Valid_ChannelT^4;
                                   ideal : int; delta_to_ideal : int)
returns (contractIsFulfilled : bool);
var vote : int; assume_isValid : bool; guarantee_isValid : bool;
let
  vote = Voter(channels, ideal, delta_to_ideal);
  assume_isValid = assumeVoter(channels, ideal, delta_to_ideal);
  guarantee_isValid = guaranteeVoter(channels, ideal, delta_to_ideal, vote);
  contractIsFulfilled = ((assume_isValid) => (guarantee_isValid));
tel
```

Nous avons tout d'abord essayé de prouver cette propriété à l'aide de l'outil *nbac*, mais celui-ci ne gère pas (pour l'instant) les divisions par des valeurs non constantes (comme celle utilisée dans le Voter pour calculer la valeur de vote).

Nous avons par contre prouvé cette propriété à l'aide des outils GLOUPS[1] et PVS. GLOUPS permet, à partir d'une propriété LUSTRE de générer des objectifs de preuve PVS. Il a fallu ensuite prouver ces objectifs de preuve dans PVS.

10.1.1.3 Composition du Channel et du Voter

L'étape suivante dans la validation du Gyroscope consiste à vérifier que la composition des instances de Channel avec le Voter est valide vis-à-vis de leurs contrats. On doit tout d'abord construire un contrat pour l'itération map_red≪Channel;4≫(mapredInit,[0,1,2,3],channels). On obtient le contrat suivant grâce à la définition donnée au paragraphe 5.3.9.4 :

Assertion – On suppose que parmi les previousOutChannel, le nombre de channel déclarés en failure est inférieur à 3. On n'a fait que propager l'assertion due à la détection des fautes de transmission (nécessaire au nœud CrossFailDetect).

Garantie – La garantie de l'itération est déduite facilement de la garantie du nœud Channel. On garantit que, pour chaque channel, soit il déclare une erreur, soit la valeur calculée n'est pas trop loin des valeurs qu'il reçoit en entrée (modulo le décalage temporel introduit par Maintain).

Lorsqu'on compose l'itération map_red≪Channel;4≫ avec le Voter, on doit vérifier l'inclusion des traces de la garantie du premier dans les traces de l'assertion du second, avec la méthode proposée au chapitre 9. On se rend compte que les contrats ne sont pas complètement compatibles : Pour chaque channel, la contrainte sur les entrées de Voter est : local_failure or local_value-ideal<delta_to_ideal, alors que la contrainte sur les sorties de Channel est : local_value - valuea < delta AND local_value - valueb < delta. De plus, Voter suppose qu'au moins un channel fonctionne bien (que sa valeur local_failure est fausse), ce qui n'est pas du tout garantie par Channel.

Dans l'application originale, le fait qu'au moins un channel fonctionne bien est garantie par les channels eux-mêmes. Ils sont implémentés de telle façon que lorsque 3 channels tombent en panne, alors le quatrième ne peut pas tomber un panne (il passe dans un mode de type « sauvetage »).

On pourrait très bien prendre en compte cette caractéristique. On modifierait pour cela le contrat ainsi que l'implémentation du Channel. Dans cette nouvelle version, la composition du Channel avec le Voter sera valide (au sens du paragraphe 9.4.2.1). Nous n'avons néanmoins pas implémenté cette version.

10.1.1.4 Validation du contrat de EvaluateAxis

Dès lors qu'on a validé la composition ci-dessus, l'implémentation de EvaluateAxis satisfait son contrat. Pour prouver cela, nous avons simplement à remarquer que la propriété garantie par le Voter est exactement celle que doit garantir EvaluateAxis. Nous n'avons cependant pas encore développé cette version.

[1] L'outil GLOUPS est disponible sur http://www-verimag.imag.fr/∼mikac/Gloups/Gloups-index.html. Il implémente les idées dévelopées dans [DC00]

10.1.1.5 Conclusion

Les contrats des composants Channel et Vote nous ont permis de nous rendre compte qu'ils n'étaient pas entièrement compatibles. Cette étude des contrats permet d'illustrer la méthodologie que nous avons présentée au paragraphe 9.5. Après la phase de spécification *descendante* que nous avons mis en place sur l'exemple au chapitre 6, nous venons de montrer comment mettre en place la validation *ascendante* correspondante. On a commencé par valider localement les composants, sans nous intéresser au contexte dans lequel ils étaient utilisés. On a ensuite étudié leur composition et nous avons montré que les contrats n'était pas compatibles. Cette phase de validation de *l'intégration* des composants n'a pas été détaillée, mais elle met en jeu une série de modifications, de *spécialisation* des composants afin de les adapter à l'environnement dans lesquels ils sont utilisés.

Comme nous l'avons dit dans le chapitre 9, nous ne prétendons pas montrer la validité d'une propriété globale sur le système. Par contre, toutes ces vérification locales, et ces vérifications des compositions permettent de s'assurer d'une cohérence partielle entre les composants de l'application. Ces vérifications sont des outils qui permettent à l'utilisateur de raisonner sur les différents composants.

Le programme complet est trop complexe pour que les outils « classiques » de validation puissent être utilisés pour prouver une propriété globale. Il est indispensable de pouvoir découper le système et de le valider partie par partie. C'est l'intérêt principal de la méthodologie que nous avons mise en place.

10.1.2 Manipulation autour de l'itération principale

La propriété principale présentée au paragraphe 6.6 est calculée à l'aide d'une itération appliquée sur le tableau secure_values, lui-même calculé par un map de EvaluateAxis. On reconnaît une forme intéressante pour la méthode proposée au chapitre 8.

Au paragraphe 8.5.2.5, nous avons vu que la présence de tableaux explicites pouvait être traitée de différentes façons. Nous étudions maintenant ces différentes solutions sur l'exemple.

10.1.2.1 Abstraction des tableaux explicites

Si l'on choisit d'abstraire ces tableaux explicites, on se retrouve à prouver la propriété suivante :

```
node propOnGyroscope(axis : Faulty_Array; deltas, deltas_to_ideals, ideals : int^3)
returns (isValid : bool);
var secure_values : int^3;
let
   secure_values = FSTGyroscope(axis);
   isValid = red≪ValueIsSecure;3≫(true,secure_values,
   ideals,deltas_to_ideals)
tel
```

On peut commencer par expanser le nœud FSTGyroscope, pour obtenir l'objectif de preuve suivant :

```
node propOnGyroscope(axis : Faulty_Array; deltas, deltas_to_ideals, ideals : int^3)
returns (isValid : bool);
var secure_values : int^3;
let
   secure_values = map<<EvaluateAxis;3>>(axis,deltas,deltas_to_ideals,ideals);
   isValid = red<<ValueIsSecure;3>>(true, secure_values,
                                    ideals,deltas_to_ideals)
tel
```

L'itération map≪EvaluateAxis;3≫ peut être encapsulée dans un nœud mapEvaluateAxis dont on construit le contrat grâce à la définition donnée au paragraphe 5.3.9.2. On obtient le contrat suivant, spécifié par les nœuds assumeMAP et guaranteeMAP.

Assertion – assumeMAP exprime que toutes les entrées de l'itération doivent satisfaire l'assertion du nœud itéré EvaluateAxis. Il est défini par :

```
node assumeMAP(axis : Faulty_Array, delta_to_ideals, ideals : int^3)
returns (okAssume : bool);
let
   okAssume = red<<iteratedAssume;3>>(true, axis,delta_to_ideals, ideals);
tel
```

où le nœud iteratedAssume est construit à partir de l'assertion du nœud EvaluateAxis :

```
node iteratedAssume(acc_in : bool; oneAxe : Faulty_ChannelT^4,
                    delta,delta_to_ideal, ideal : int)
returns (acc_out : bool);
let
   acc_out = acc_in and assumeEvaluateAxis(oneAxe,delta,delta_to_ideal,ideal);
tel
```

Garantie – de la même façon, on construit la guarantee en itérant iteratedGuarantee :

```
node guaranteeMAP(axis : Faulty_Array, delta_to_ideals,
                  ideals : int^3; secure_values : int^3)
returns (okGuarantee : bool);
let
   okGuarantee = red<<iteratedGuarantee;3>>(true, axis,delta_to_ideals, ideals);
tel
```

où le nœud iteratedGuarantee est construit à partir de la garantie : du nœud EvaluateAxis :

```
node iteratedGuarantee(acc_in : bool; oneAxis : Faulty_ChannelT^4,
                       delta,delta_to_ideal, ideal : int;
                       secure_values : int)
returns (acc_out : bool);
let
   acc_out = acc_in and guaranteeEvaluateAxis(oneAxis,delta,
                                              delta_to_ideal,ideal,secure_values);
tel
```

Le nouvel observateur qu'on doit considérer est simplement :

```
node propOnGyroscope(axis : Faulty_Array; deltas, deltas_to_ideals, ideals : intˆ3)
returns (isValid : bool);
var secure_values : intˆ3;
let
   secure_values = mapEvaluateAxis(axis,deltas,deltas_to_ideals,ideals);
   isValid = red≪ValueIsSecure;3≫(true, secure_values,
                                        deltas_to_ideals,ideals);
tel
```

On peut maintenant appliquer l'algorithme de manipulation des itérations. Celui-ci va rencontrer le nœud mapEvaluateAxis et proposer d'utiliser son contrat. Il va dans un premier temps générer le nouvel objectif de preuve suivant :

```
node propOnGyroscope(axis : Faulty_Array;
                     deltas, deltas_to_ideals, ideals : intˆ3;
                     secure_values : intˆ3)
returns (isValid : bool);
var assumeMAPV, guaranteeMAPV : bool;
let
   isValid = red≪ValueIsSecure;3≫(true,secure_values,
                                  ideals,deltas_to_ideals,
   assumeMAPV = red≪iteratedAssume;3≫(true,axis,deltas,deltas_to_ideals);
   guaranteeMAPV = red≪iteratedGuarantee;3≫(true,axis,
                                            deltas,deltas_to_ideals,ideals,secure_values);
   assert assumeMAPV
          => guaranteeMAPV;
tel
```

L'assertion a été construite à partir du contrat de mapEvaluateAxis. On peut alors appliquer l'algorithme proposé au paragraphe 8.5. On obtient ainsi l'observateur de la figure 10.1. Cet observateur a la structure classique que nous avons présentée au paragraphe 8.5.4.1. La propriété (représentée par la variable ok) est découpée en deux sous-propriétés okInit pour le cas de base et okInv pour l'invariance de l'induction. A chaque fois la propriété générée est une implication Hypothèse => Propriété où Hypothèse représente la validité du contrat que nous avons transformé ci-dessus en implication assumeMAPV => guaranteeMAPV et où Propriété représente la validité de la propriété représentée par ValueIsSecure.

Cas de base – On doit prouver que la propriété est vraie après un passage dans l'itération, on a donc généré un appel du nœud ValueIsSecure avec les variables init correspondant à chaque variable du programme : deltas_to_ideals_init, ideals_init, deltas_init et secure_values_init.

Invariance – On doit prouver que la propriété est préservée par un passage dans l'itération. On génère pour cela la variable HPAcc qui représente la validité de l'implication Hypothèse => Propriété à un rang N de l'itération, ainsi que la variable HPInv qui représente la validité de l'implication au rang N+1.

```
node propOnGyroscope(secure_values_init : int; secure_values_inv : int;
                     deltas_to_ideals_init, deltas_to_ideals_inv : int;
                     ideals_init, ideals_inv : int;
                     deltas_init, deltas_inv : int;
                     propTrueRankN : bool;
                     assumeMAPV_acc, guaranteeMAPV_acc : bool;
                     accIn_acc_in : bool;
                     elt_axis_init, elt_axis_inv : Faulty_ChannelT^4)
returns (ok : bool);
var isValid_init : bool;
    propTrueRankNp1 : bool;
    hypo_init : bool;
    assumeMAPV_init, assumeMAPV_inv : bool;
    guaranteeMAPV_init : bool, guaranteeMAPV_inv : bool;
    hypoTrueRankN : bool, hypoTrueRankNp1 : bool;
    HPAcc : bool, HPInv : bool;
    okInv : bool, okInit : bool;
let
  ok = ((okInit) and (okInv));
– cas de base de l'induction
  okInit = ((hypo_init) => (isValid_init));
  hypo_init = ((assumeMAPV_init) => (guaranteeMAPV_init));
  isValid_init = ValueIsSecure(true, secure_values_init, deltas_to_ideals_init, ideals_init);
– invariance
  okInv = ((HPAcc) => (HPInv));
  HPAcc = ((hypoTrueRankN) => (propTrueRankN));
  HPInv = ((hypoTrueRankNp1) => (propTrueRankNp1));

  propTrueRankNp1 = g_ValueIsSecure(propTrueRankN, secure_values_inv,
  deltas_to_ideals_inv, ideals_inv);
  hypoTrueRankN = ((assumeMAPV_acc) => (guaranteeMAPV_acc));
  hypoTrueRankNp1 = ((assumeMAPV_inv) => (guaranteeMAPV_inv));
  assumeMAPV_init = iteratedAssume(true, elt_axis_init, elt_deltas_init,
                                   deltas_to_ideals_init, ideals_init);
  assumeMAPV_inv = iteratedAssume(assumeMAPV_acc, elt_axis_inv, elt_deltas_inv,
                                   deltas_to_ideals_inv, ideals_inv);
  guaranteeMAPV_init = iteratedGuarantee(true, elt_axis_init, elt_deltas_init,
                                  deltas_to_ideals_init, ideals_init, secure_values_init);
  guaranteeMAPV_inv = iteratedGuarantee(guaranteeMAPV_acc1, elt_axis_inv, elt_deltas_inv,
                                  deltas_to_ideals_inv, ideals_inv, secure_values_inv);
tel
```

FIG. 10.1 – Résultat des manipulations tableaux sur le gyroscope.

10.1.2.2 Prise en compte des tableaux explicites

Si l'on ne souhaite pas abstraire les tableaux explicites, la technique dont nous avons parlé au paragraphe 8.5.2.5 peut être appliquée. On commence par appliquer les même transformations que précédemment (construction du contrat de l'itération, etc.). On doit alors traiter l'observateur suivant :

```
node propOnGyroscope(axis : Faulty_Array;
                     secure_values : intˆ3)
returns (isValid : bool);
var assumeMAPV, guaranteeMAPV : bool;
let
   isValid = red≪ValueIsSecureII;3≫(true,
        secure_values,
        [DELTA_TO_IDEAL_ROLL, DELTA_TO_IDEAL_PITCH, DELTA_TO_IDEAL_YAW],
        [IDEAL_ROLL,IDEAL_PITCH, IDEAL_YAW]);
   assumeMAPV = red≪iteratedAssume;3≫(true,axis,
       [DELTA_ROLL, DELTA_PITCH, DELTA_YAW],
       [DELTA_TO_IDEAL_ROLL, DELTA_TO_IDEAL_PITCH, DELTA_TO_IDEAL_YAW]);
   guaranteeMAPV = red≪iteratedGuarantee;3≫(true,
        axis,[DELTA_ROLL, DELTA_PITCH, DELTA_YAW],
        [DELTA_TO_IDEAL_ROLL, DELTA_TO_IDEAL_PITCH, DELTA_TO_IDEAL_YAW],
        [IDEAL_ROLL,IDEAL_PITCH, IDEAL_YAW],
        secure_values);
   assert assumeMAPV
          => guaranteeMAPV;
tel
```

À partir de cet observateur, on peut générer un observateur ressemblant à celui de la figure 10.1 où les instances des variables deltas_init, deltas_to_ideals_init et ideals_init ont été remplacées par les valeurs constantes correspondantes pour le premier passage dans l'itération.

En ce qui concerne le cas init, on doit générer une assertion indiquant qu'elles valeurs peuvent être utilisées, en même temps pour ces différentes variables.

10.1.2.3 Symétrie

La propriété que nous avons étudiée jusqu'ici est en fait parfaitement symétrique. On veut en fait montrer que la valeur produite par le système pour *chaque* axe n'est pas trop loin de la valeur idéale correspondante. On peut ainsi l'exprimer à l'aide de l'opérateur forall. On écrirait l'observateur suivant :

```
node propOnGyroscope(axis : Faulty_Array) returns (isValid : bool);
var secure_values : intˆ3;
let
   secure_values = FSTGyroscope(axis);
   isValid = forall≪ValueIsSecure_bis;3≫(true,
             secure_values,
             [DELTA_TO_IDEAL_ROLL,DELTA_TO_IDEAL_PITCH,DELTA_TO_IDEAL_YAW],
             [IDEAL_ROLL,IDEAL_PITCH,IDEAL_YAW]);
tel
```

où ValueIsSecure_bis est le nœud suivant, construit à partir du nœud ValueIsSecure utilisé dans le paragraphe 6.6 :

```
node ValueIsSecure_bis(secure_value : int;
                       delta_to_ideal, ideal : int)
returns (is_valid : bool);
let
   is_valid = (abs((secure_value - ideal)) < delta_to_ideal);
tel
```

À partir de ce nouvel objectif de preuve, on peut appliquer la technique proposée au paragraphe 8.5.2.1. Comme nous l'avons présenté ci-dessus, on peut au choix :

– On obtient alors l'objectif de preuve suivant :

```
node propOnGyroscope(elt_axis : Faulty_ChannelT^4;
                     ideals, deltas_to_ideals : int)
returns (ok : bool);
let
   ok = ValueIsSecure_bis(elt_axis, deltas_to_ideals, ideals);
tel
```

– les prendre en compte : auquel cas on rajoute à l'objectif de preuve précédent l'assertion suivante :

```
assert (deltas_to_ideals = DELTA_TO_IDEAL_ROLL and ideals = IDEAL_ROLL)
       or(deltas_to_ideals = DELTA_TO_IDEAL_PITCH and ideals = IDEAL_PITCH)
       or(deltas_to_ideals = DELTA_TO_IDEAL_YAW and ideals = IDEAL_YAW);
```

10.1.2.4 Conclusion

Les différents objectifs de preuve que nous venons de décrire peuvent ensuite être fournis aux outils de preuve standard. Le gain apporté ici peut paraître minime, puisqu'au lieu d'un observateur portant sur 3 variables (les 3 éléments du tableau axis), on doit maintenant prouver une propriété sur l'équivalent de 2 éléments (voir 1 élément si l'on spécifie la propriété à l'aide d'un forall).

Mais la taille de l'objectif de preuve généré est en fait *constante par rapport aux nombres d'éléments des tableaux* manipulés. Elle ne dépend que du nombre de variables du programmes original. Le gain peut être très important (voir les exemples de propriétés dans l'ELMU, paragraphe 10.2.2).

La validité de la preuve ne dépend pas non plus de la taille des tableaux, on sait que si on prouve les objectifs de preuve générés, la propriété sera valide. La preuve est donc *générique* : on peut prouver la propriété sans connaître exactement la taille des tableaux manipulés. Plus globalement, c'est un argument en faveur de l'utilisation des itérations.

10.2 ELMU

Nous revenons maintenant sur l'application du ELMU que nous avons présentée au chapitre 7. Nous avons montré alors l'intérêt des contrats et des itérations en terme de spécification. Comme nous venons de le faire pour le Gyroscope, nous tentons de montrer maintenant ce qu'apportent ces deux aspects en terme de validation de l'application.

10.2.1 Manipulation des contrats

Nous avons évoqué, au chapitre 7, l'existence, dans la spécification du ELMU d'une bibliothèque de composants de base, intensivement utilisés dans toute l'application. L'exemple du rMFF introduit au paragraphe 5.3.1 est notamment tiré de cette étude de cas.

Les manipulations des contrats qu'on a proposé au chapitre présentent ici un grand intérêt : elles permettent de valider l'utilisation qui est faite de chacun de ces composants. A chaque endroit où un composant est utilisé, on vérifie la cohérence de l'appel. On peut aussi propagée les contraintes décrites dans ces contrats pour exprimer des contraintes sur des paramètres de composants appelant. Cette propagation permet de renforcer la description qu'on a de telle ou telle partie de l'application.

Par exemple, on peut se rendre compte que des contraintes exprimées sur des paramètres d'un noeud N sont contradictoires avec l'utilisation qui en est fait par un noeud spécifié par contrat et ce de manière semi-automatique (en appliquant les méthodes du chapitre 9). L'application telle que nous l'avons étudié ne permet pas cette démarche : même si certains composants sont spécifiés, ils le sont de manière trop informelle pour que l'information correspondante soit exploitée.

Les méthodes que nous proposons ne permettent pas une validation globale de l'application. Mais à chaque fois que le développeur fait l'effort de donner un contrat pour un composant, il en tirera une plus grande confiance dans l'utilisation qui est faite du composant.

10.2.2 Manipulation des itérations

La régularité de structure mise en évidence dans le paragraphe 7.4 se prête tout à fait à la méthode proposée au chapitre 8. Au niveau global de l'application, les itérations utilisées (extractions d'informations et reconstitution d'informations) n'effectuent aucun calcul sur les données du système, seulement des réarrangements de tableaux.

Les traitements des charges et des générateurs, que nous avons mis en évidence avec les nœuds traiteChg et traiteGen sont des nœuds de calculs. Ils sont implémentés de manière symétrique (on réalise un traitement identique pour chaque charge, ou pour chaque générateur). Nous les avons décrits avec des map des nœuds traite_charge et traite_gen.

Pour chacun de ces traitements on propose maintenant l'étude d'une propriété exprimée à l'aide d'une réduction.

10.2.2.1 Traitement des charges

Une propriété simple sur le traitement des charges, est que si une charge a une priorité de niveau maximal (PRIO_CHG_FORTE), il ne doit pas être possible de la décharger, donc l'état sortant de traitChg correspondant à cette charge doit être toujours EC_ON (dans tous les cas, il est possible, à un moment, d'éteindre cette charge. Cette propriété est symétrique par rapport aux charges, on peut donc l'exprimer à l'aide de l'opérateur forall. On décrit donc l'observateur suivant :

```
node prop(TabInfoChgIndiv : T_InfoChgIndivˆNB_CHARGES;
          TabInfoChgGlob : T_InfoChgGlobˆNB_CHARGES)
returns (ok : bool);
var TabEtatCharge : T_EtatChargeˆNB_CHARGES;
let
   TabEtatCharge = traiteChg(TabInfoChgIndiv, TabInfoChgGlob);
   ok = forall≪ifPrioThen_ON;NB_CHARGES≫(TabInfoChgIndiv, TabEtatCharge);
tel
```

Le nœud ifPrioThen_ON permet d'exprimer la propriété « Si une charge a une prioriété forte, alors elle doit toujours être laissée à EC_ON » :

```
node ifPrioThen_ON(acc_in : bool; eltChg : T_InfoChgIndiv; etatCharge : T_EtatCharge)
returns (acc_out : bool);
let
   acc_out = true -> (eltChg.priorite_chg=PRIO_CHG_FORTE) => (etatCharge=EC_ON);
tel
```

On commence par expanser le nœud traiteChg pour se rendre compte qu'il calcule ses sorties à l'aide d'une itération du nœud traite_charge. On obtient une propriété exprimée par un forall et portant sur des tableaux tous calculés par une itération de type map. La méthode présentée au chapitre 8 permet alors d'obtenir l'objectif de preuve suivant :

```
node prop(elt_TabInfoChgIndiv : T_InfoChgIndiv;
          elt_TabInfoChgGlob : T_InfoChgGlob)
returns (ok : bool);
var TabEtatCharge : T_EtatCharge;
let
   elt_TabEtatCharge = traite_charge(elt_TabInfoChgIndiv, elt_TabInfoChgGlob);
   ok = ifPrioThen_ON(elt_TabInfoChgIndiv, elt_TabEtatCharge);
tel
```

Le nombre de charges est très important, on doit manipuler des tableaux d'une vingtaine d'éléments. Le gain apporté par cette méthode dans ce cas est donc aussi très important (réduction d'un facteur 20 de la taille du programme à valider).

10.2.2.2 Traitement des générateurs

Sur le traitement des générateurs, on va vouloir exprimer la propriété suivante : Si une charge est déclarée EC_ON, elle ne doit pas être éteinte, il est donc important que le traitement des générateurs, qui identifie les charges à délester ou à éteindre en cas de surcharge d'un générateur n'éteigne pas ces charges. On va donc commencer par décrire la propriété « Si une charge est déclarée EC_ON alors la commande qui doit lui être fournie par le traitement des générateurs doit être COM_ON », grâce au nœud IsAssociatedToOne :

```
node IsAssociatedToOne(acc_in : bool; etatCharge : T_EtatCharge;
                       comChg : T_ComChg)
returns (acc_out : bool);
let
   acc_out = acc_in or (etatCharge=EC_ON => comChg=COM_ON);
tel
```

Le tableau TabEtatCharge reçu par traiteGen doit donc être parcouru une fois pour chaque générateur. A chaque fois, on doit comparer le tableau entier avec la sous-partie du tableau de commande AllTabComVal calculé par traiteGen. Au vérifiera ainsi qu'*au moins* un générateur émet une commande COM_ON pour une charge qui est déclarée à EC_ON. Le nœud OrOnOneGenerateur vérifie que, pour une générateur donné, dont on donne le tableau de commande par tabComChg, il est possible que si une charge est à EC_ON, alors le générateur lui associe la commande COM_ON.

```
node OrOnOneGenerateur(acc_in : bool; tabEtatCharge : T_EtatChargeˆNB_CHARGES;
                       tabComChg : T_ComChgNBC)
returns (acc_out : bool);
let
   acc_out = acc_in or red≪IsAssociatedToOne;NB_CHARGES≫(false,
                                                          tabEtatCharge,
                                                          tabComChg);
tel
```

Pour obtenir la propriété globale désirée, on itère à nouveau ce nœud sur l'ensemble des générateurs. On utilise :

```
node allCOM_ON_IF_EC_ON(tabEtatCharge : T_EtatChargeˆNB_CHARGES;
                        allTabComChg : T_ComChgNBCˆNB_GENE)
returns (ok : bool);
let
   ok = red≪OrOnOneGenerateur;NB_GENE≫(false, tabEtatChargeˆNB_GENE,
   allTabComChg);
tel
```

Finalement la propriété est exprimée en appliquant le nœud précédent sur le tableau TabEtatCharge reçu par traiteGen en entrée et sur le tableau AllTabComChg rendu en sortie par ce même nœud.

```
node prop(TabIndiceGen  : INTNBG;
          TabInfoGenIndiv : T_InfoGenIndivˆNB_GENE;
          TabInfoGenGlob  : T_InfoGenGlobˆNB_GENE;
          TabEtatCharge : T_EtatChargeˆNB_CHARGES)
returns (ok : bool);
var AllTabComChg  :T_ComChgNBCˆNB_GENE;
    AllTabComVal  : BOOLNBCˆNB_GENE;
let
   AllTabComChg, AllTabComVal = traiteGen(TabIndiceGen,TabInfoGenIndiv,
                                          TabInfoGenGlob,TabEtatCharge);
   ok = allCOM_ON_IF_EC_ON(TabEtatCharge, AllTabComChg);
tel
```

La propriété exprimée à l'aide des itérations n'est pas un invariant : on veut prouver qu'*il existe* un rang de l'itération où une propriété P est vraie. En renforçant cette propriété en un invariant, on va essayer de prouver que P est vrai *pour tous* les rangs de l'itération.

10.2.2.3 Commentaires

Cet exemple est particulièrement intéressant pour le traitement des itérations : on a identifié des calculs totalement réguliers sur lesquels l'algorithme de traitement des itérations du chapitre 8 peut apporter un gain en terme d'efficacité de la preuve.

La taille des tableaux utilisés justifie, plus que dans l'exemple du gyroscope, l'application de cet algorithme : la version que nous avons étudiée est constitué de 4 générateurs et de 20 charges. L'indépendance de la taille de la preuve vis-à-vis de la taille des tableaux manipulés est ici très intéressante.

Cet exemple montre aussi les limitations de l'approche choisie au chapitre 8. Nous ne proposons pas une méthode complète pour prouver des algorithmes itératifs complexes, mais seulement une méthode qui peut tirer partie de certaines formes de *symétries* d'un système pour faciliter la preuve.

Chapitre 11

Implémentation

Dans ce chapitre, nous présentons le prototype qui a été développé durant cette thèse. Nous avons implémenté les algorithmes présentés dans les chapitre précédents, tant pour le traitement de programmes itératifs que pour les manipulations de contrats. Nous avons intégré ces algorithmes dans un outil graphique permettant de les appliquer sur un programme donné.

11.1 Introduction

Dès le début de ce travail, nous nous sommes fixé comme objectif le développement d'un prototype implémentant les manipulations de programmes proposées. Cette implémentation nous paraissait indispensable pour valider ces algorithmes ainsi que la méthodologie globale évoquée plus haut.

Le logiciel dans sa version « initiale » devait permettre le traitement des programmes LUSTRE avec itérateurs et la génération des obligations de preuve liées aux itérations. Les manipulations se font essentiellement sur un arbre abstrait (tel que celui utilisé au paragraphe 8.5). Les premières manipulations ont été testées directement sur de tels arbres abstraits (les exemples étaient codés « en dur » dans le programme). Nous n'avons pas développé notre propre analyseur, car à l'époque où nous avons commencé ce travail le front-end du nouveau compilateur LUSTRE (dont nous avons déjà parlé au paragraphe 4.7.2) était sur le point d'être développé. Il nous a paru plus intéressant d'attendre que celui-ci soit disponible pour pouvoir l'utiliser directement (analyses lexicales et syntaxiques, vérification des types, des horloges, etc.).

Ce front-end a été développé en 2003-2004 par Youssef Bouzouzou. Nous avons alors connecté notre prototype à cet analyseur. Ce branchement a été rendu possible par le fait que les deux outils sont implémentés en Java. Il a ainsi suffit de rassembler les classes nécessaires du front-end et du prototype pour faire une connexion directe. Cela nous a permis de profiter de l'analyseur développé par Y. Bouzouzou et de réserver le temps de développement pour les aspects vraiment fondamentaux de notre travail (transformations liées aux itérations et aux contrats).

En même temps, l'idée de la gestion d'une preuve d'une propriété (comme elle est mise en place dans des outils de preuve formelle comme PVS) nous a paru attrayante. Nous avons donc envisagé un logiciel qui permettrait aussi de gérer une preuve en terme de sous-buts. On peut par exemple représenter le fait que pour prouver qu'une propriété itérative est vraie, il suffit que les cas Init et Inv soient satisfaits. Nous verrons plus loin que l'arbre qu'on utilise n'a pas toujours une sémantique simple d'arbre *ET*.

Nous avons ainsi intégré dans notre outil la notion d'arbre de preuve : chaque nœud de l'arbre

représente une propriété à prouver, ses fils représentant les propriétés qu'il suffit (ou qu'il faut, selon les cas) prouver pour prouver la propriété mère.

Cette notion d'arbre de preuve est assez visuelle. Nous avons donc commencé à réfléchir à une interface graphique permettant de manipuler cet arbre (et facilitant aussi l'identification des itérations à traiter, etc.).

Lorsque nous nous sommes intéressé aux contrats pour LUSTRE, l'interface graphique s'est vraiment montrée indispensable. Elle permet en effet l'application des manipulations de contrats sur tous les nœuds constituant un programme, à la demande de l'utilisateur (ce qui lui accorde une grande souplesse dans l'utilisation de ces transformations).

Ces réflexions ont menés à un prototype appelé GOUPIL, pour *Générateur d'Obligation de Preuve pour programmes Itératifs* LUSTRE. Les paragraphes suivants décrivent chacun des points particuliers des fonctions implémentées dans GOUPIL. Nous commençons par présenter l'interface graphique du logiciel (section 11.2). Dans la section 11.4, on présente les manipulations d'itérations effectivement implémentées.

11.2 Analyse syntaxique

GOUPIL est invoqué en lui fournissant un nom de fichier LUSTRE et un nom de nœud principal. Il commence par effectuer une analyse syntaxique et de type du programme. Cette analyse est exactement celle réalisée par le nouveau compilateur LUSTRE. Pour cela, nous avons directement utilisé les classes Java développées par Y. Bouzouzou.

Cet analyseur construit un premier arbre syntaxique. Comme nous l'avons déjà indiqué, le front-end du compilateur LUSTRE-V6 n'existait pas lorsque nous avons commencé à implémenter nos algorithmes de transformation. Nous avions à cette époque développé notre propre structure d'arbre abstrait pour représenter les programmes LUSTRE. Au lieu de re-implémenter ces algorithmes pour utiliser les arbres du compilateur LUSTRE-V6, nous avons simplement décrit la transformation entre les arbres LUSTRE-V6 et les arbres utilisés dans les algorithmes implémentés dans GOUPIL.

Sur les différentes versions de LUSTRE **–** Plusieurs versions de LUSTRE ont été développées. Au chapitre 2 nous avons présenté LUSTRE-V4 et les formats intermédiaires utilisés par les différents outils de validation. Nous avons aussi mentionné le format V4 étendu avec les itérateurs) que le débogueur LUDIC accepte en entrée. Le compilateur développé par Y. Bouzouzou traite des programmes en LUSTRE-V6. L'outil GOuPIL accepte en entrée des programmes LUSTRE-V6 mais ne prend pas en compte les éventuel packages, ni certaines formes syntaxiques particulières dont nous n'avons pas fait mention ici. Les obligations de preuve générées par GOuPIL sont toujours compatibles LUSTRE-V6. De plus, on peut demander de les générer en LUSTRE compatible avec LUDIC. De toute façon, à partir d'un programme compatible avec le compilateur LUSTRE-V6, il est toujours possible d'obtenir un programme équivalent compatible avec LUDIC. De la même manière, on peut générer du code LUSTRE-V4, ce qui permet d'utiliser les outils LESAR et NBAC. La figure 11.1 montre les compatibilités de formats de tous ces outils (version étendue par rapport à celle donnée au chapitre 2).

11.3 Interface utilisateur

Une fois les analyses syntaxiques et sémantiques effectuées, l'interface graphique de GOuPIL est lancée. On donne un exemple d'une session à la figure 11.2. L'interface principale est séparée en 3

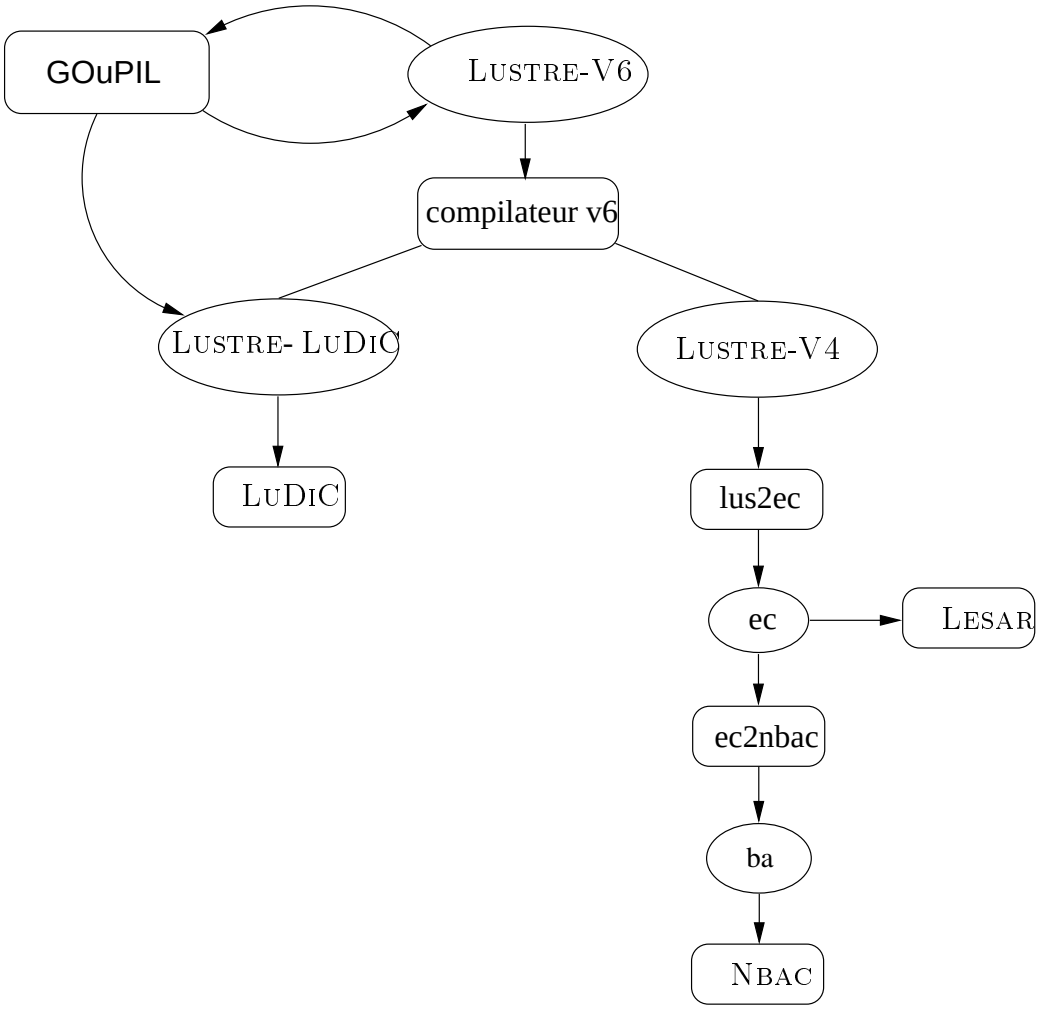


FIG. 11.1 – Compatibilités de GOuPIL avec LUSTRE-V6 et LUDIC.

parties principales.

La première (à droite), nommée *Program* est une fenêtre où est affiché le programme source couramment traité. On peut aussi y voir tous les nœuds qui ont été générés par les manipulations effectuées par l'outil. Le programme que l'on a lancé dans la capture d'écran présentée à la figure 11.2 est MRTRIG : une variante du nœud MTRIG que nous avons décrit plus au (voir paragraphe 8.2.4.1).

La seconde partie, nommée *Proof Tree* donne une représentation graphique de l'arbre de preuve. Nous reviendrons plus loin sur l'utilisation de cet arbre de preuve.

Enfin, la petite partie en bas à gauche est un panneau où des messages sont affichés pour l'utilisateur. On y indique par exemple que la dernière opération demandée a bien été effectuée.

La barre principale de l'interface contient simplement un menu qui permet de quitter le logiciel (pour l'instant, il n'est pas possible de fermer le programme courant et d'en ouvrir un autre). On peut aussi noter un menu déroulant qui suit l'inscription « *show code for node :* ». Ce menu permet de sélectionner un des nœuds du programme pour le voir affiché dans la fenêtre.

Deux menus contextuels permettent de commander l'outil. Le premier peut être activé sur l'arbre de preuve, le second directement sur l'affichage du programme. Le menu de l'arbre de preuve permet en fait simplement de générer les fichiers LUSTRE correspondant aux programmes générés par les différents algorithmes implémentés. On peut choisir de générer du code compatible avec le compilateur LUSTRE-V6 ou avec LUDIC. On peut aussi lancer directement LUDIC sur le programme associé à l'arbre de preuve sélectionné.

Le menu contextuel de la partie *Program* est plus intéressant. C'est à partir de là qu'on va pouvoir commander toutes les transformations sur le programme. Son contenu dépend de l'endroit où l'on clique dans le programme.

Si on clique sur la signature d'un nœud, ou sur la définition de son contrat (les chaînes de caractères commençant par % ASSUME ou % GUARANTEE), le logiciel nous propose d'opérer deux manipulations (figure 11.3) :

- La première consiste à générer un observateur permettant de vérifier que le nœud sélectionné implémente bien le contrat associé, comme nous l'avons vu au paragraphe 9.4.1.2 ;
- La seconde consiste à générer un observateur (nœud avec une variable locale indéfinie) permettant de vérifier que le contrat du nœud sélectionné est bien implémentable, comme nous l'avons vu au paragraphe 9.4.1.1.

Si on clique sur un appel de nœud (par exemple N) , le logiciel propose deux actions différentes (voir figure 11.4) :

- La première consiste à expanser l'appel de N, c'est-à-dire à le remplacer par les équations qui le définissent ;
- La seconde consiste à renforcer le contrat appelant par le contrat du nœud N, comme le propose le paragraphe 9.4.2.2. Il s'agit de la propagation ascendante.

Enfin, si on clique sur une itération, le logiciel propose 4 actions (figure 11.5) :

- On peut ensuite demander au logiciel de remplacer l'itération considérée par un nœud l'encapsulant. En effectuant cette encapsulation, GOUPIL, on construit un contrat correspondant à l'itération, comme cela a été proposé au paragraphe 5.3.9. On peut alors manipuler ce nœud comme n'importe quel autre, en particulier oublier qu'il est réalisé par une itération, et n'utiliser que le contrat nouvellement construit ;
- La troisième action possible sur une itération consiste à vérifier que l'itération est possible vis-à-vis du contrat du nœud itéré. Pour cela, on génère une obligation de preuve comme indiqué au paragraphe 9.4.2.3 ;

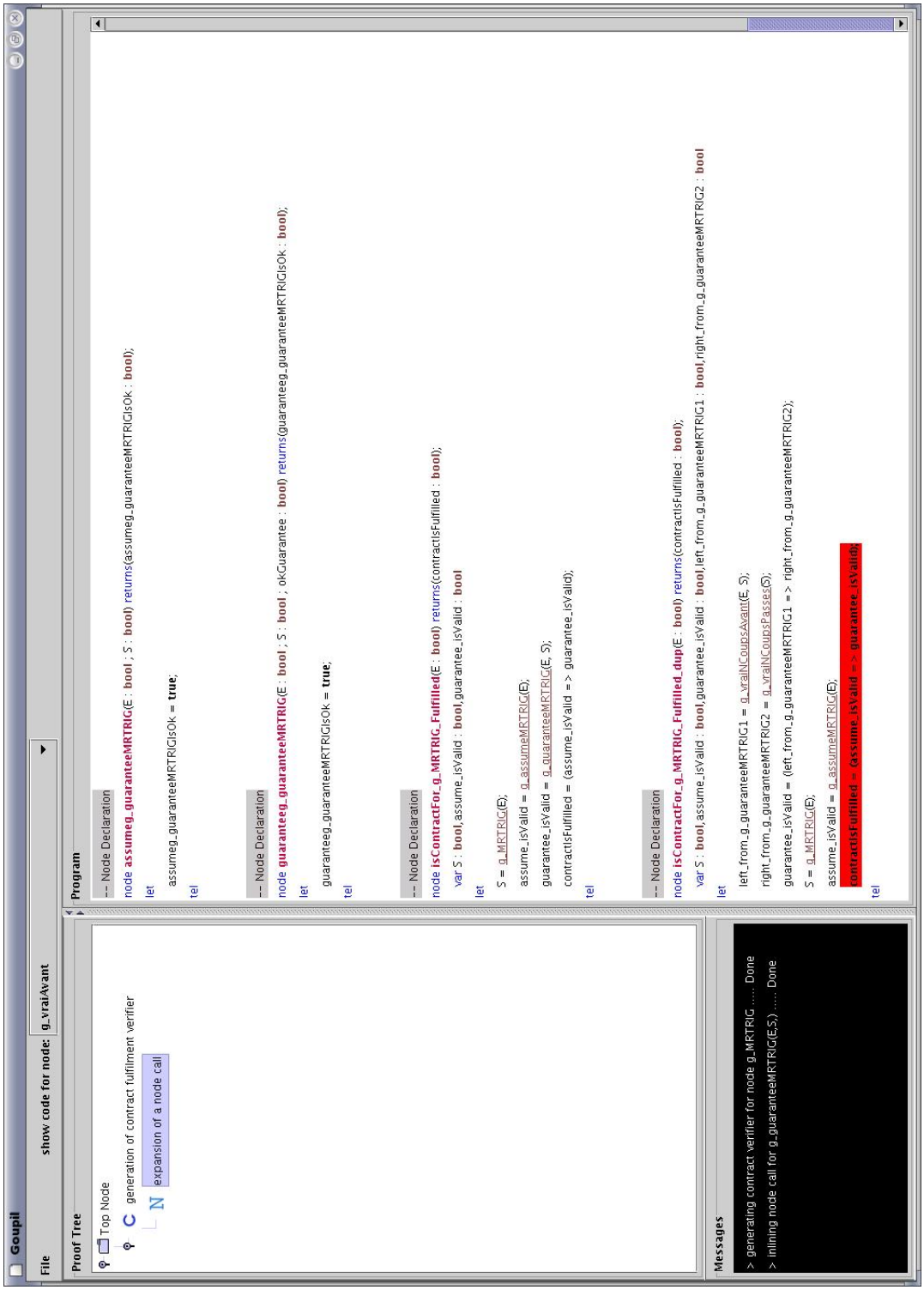


FIG. 11.2 – L'interface utilisateur de GOUPIL.

- La dernière action proposée consiste en l'application de l'algorithme de manipulation d'itérations donné au paragraphe 8.5. Nous y revenons plus loin.

Pour chacune des transformations proposées, le résultat est ajouté à l'arbre syntaxique du programme. On peut par la suite demander à GOUPIL de générer le code LUSTRE correspondant, comme indiqué plus haut.

11.4 Traitement des itérations

Le traitement sur les itérations proposé au chapitre 8 est complètement implémenté dans GOUPIL, traitant tous les cas mentionnés :

- Propriété calculée par une réduction unique ;
- Propriété calculée par une cascade quelconque d'itérations ;
- Une hypothèse peut être exprimée par une clause assert contenant une itération (dont les paramètres peuvent être calculés par d'autres itérations du nœud) ;
- Prise en compte des appels de nœud.

La prise en compte des appels de nœud se fait de la manière suivante. Pendant le déroulement « arrière » de l'algorithme, on détecte le premier appel de nœud rencontré. On propose alors à l'utilisateur, au moyen d'une fenêtre graphique (voir la figure 11.6), trois options :

- La première consiste simplement à ne rien faire. C'est l'action sélectionné par défaut lorsque la fenêtre s'ouvre. Cela doit notamment permettre à l'utilisateur de réfléchir posément à la situation ;
- La seconde permet à l'utilisateur de faire expanser complètement le nœud appelé. Cela peut-être utile notamment si il n'y a pas de contrat associé au nœud ;
- Enfin, la troisième option permet d'appliquer l'algorithme : GOuPIL commence alors par générer un nouvel observateur avec une assertion calculée à partir du contrat du nœud appelé. Ensuite, ce nouveau nœud est traité comme d'habitude, en appliquant la règle de transformation du paragraphe 8.5.2.6. L'observateur intermédiaire (avec hypothèse itérative) n'est pas conservé. On pourrait imaginer qu'il le soit, pour rendre plus clair l'algorithme.

Le traitement de l'opérateur forall a aussi été implémenté dans les algorithmes de transformations. Néanmoins, celui-ci ne fait pour l'instant pas partie de la syntaxe officielle de LUSTRE-V6, il n'est donc pas encore possible de l'utiliser.

11.5 Note d'implémentation

GOuPIL a été entièrement développé en JAVA-1.4. Pour tous les algorithmes implémenté, nous avons utilisé la notion de visiteur [GHJV95]. Chaque type de nœud de l'arbre abstrait est représenté par une classe et les manipulations liées à chacune de ses classes sont rassemblées dans un même fichier *visiteur* pour chaque algorithme.

À partir d'un fichier LUSTRE, on fait générer un arbre abstrait par les classes développées par Y. Bouzouzou. Cet arbre abstrait supporte aussi une manipulation à base de visiteur. Ainsi, pour la traduction de l'arbre abstrait LUSTRE-V6 vers l'arbre abstrait GOuPIL, tout est géré dans une seule classe qui spécifie, pour chaque type de nœud de l'arbre abstrait d'origine, comment générer un nœud équivalent dans l'arbre abstrait cible.

FIG. 11.3 – Menu contextuel activé sur une en-tête de nœud.

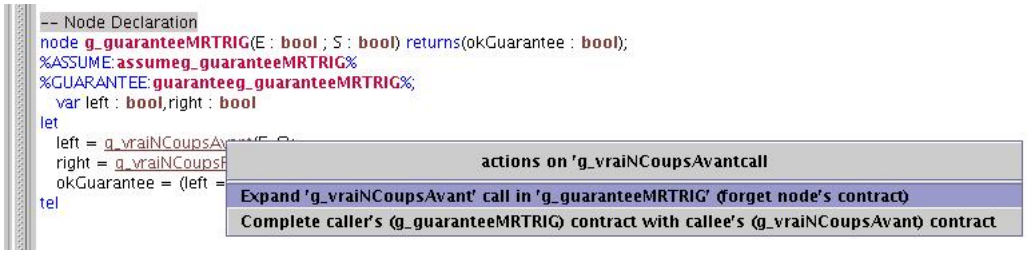


FIG. 11.4 – Menu contextuel activé sur un appel de nœud.

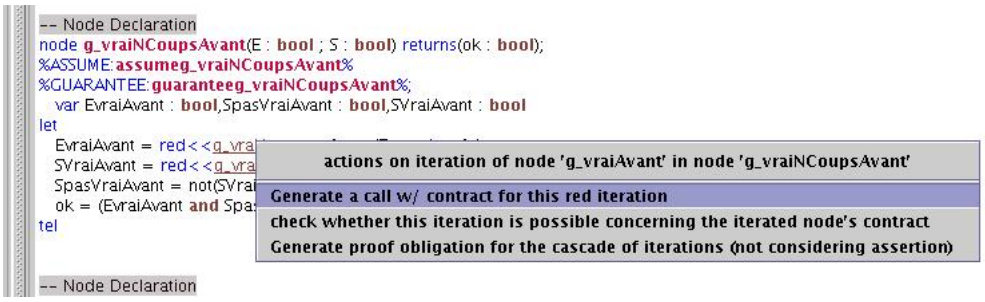


FIG. 11.5 – Menu contextuel activé sur une itération.

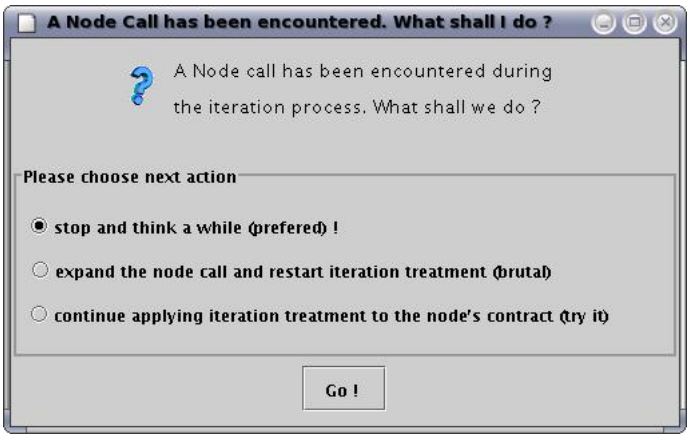


FIG. 11.6 – Choix de l'action à effectuer pour un nœud rencontré.

11.6 Arbre de preuve

La notion d'arbre de preuve que nous avons évoquée plus haut n'a pas été réellement exploitée dans ce travail. Au départ, nous avons développé l'idée d'un arbre de preuve strict où il existe une relation logique forte entre un nœud et ses fils. On peut dans ce cas distinguer au moins deux types de nœuds :

- les nœuds « *ou* » : la propriété qu'un nœud *ou* représente est vérifiée si la propriété représentée par au moins un de ses fils est vérifiée ;
- les nœuds « *et* » : la propriété qu'un nœud *et* représente est vérifiée si la propriété représentée par chacun de ses fils est vérifiée.

Ce type d'organisation de la preuve est notamment utile pour représenter le traitement des itérations. On cherche à prouver une propriété globale (impliquant des itérations) et on se ramène à des sous-propriétés suffisantes qu'il suffit de prouver. La structure arborescente est dans ce cas correcte.

Pour les traitements liées aux contrats, nous avons choisi d'ajouter un nœud d'arbre de preuve à chaque vérification demandée par l'utilisateur. Il n'y a alors pas forcément de dépendance entre les objectifs de preuve que l'on créé. Si l'on considère un arbre de preuve initial dont la racine représente la validité du système (par rapport à une propriété globale), un nœud de l'arbre de preuve représentant la validité d'un contrat n'a pas de signification par rapport à la propriété globale.

On pourrait par contre envisager d'opter pour une notion plus large d'arbre de preuve. La racine de l'arbre pourrait représenter la validité de l'application étudiée, comprise au sens large. On considérerait que l'application est validée lorsque :

- Pour chaque composant doté d'un contrat, on a vérifié la compatibilité du contrat avec l'implémentation donnée ;
- Pour chaque branchement entre deux composants on a vérifié les compositions des contrats ;
- Toutes les propriétés globales (de l'application) ou partielles (de certaines parties de l'application) ont été vérifiées.

Nous n'avons pas exploré plus en avant l'exploitation d'un tel arbre de preuve, mais l'approche que nous venons d'évoquer permettraient de rendre plus aisée notre méthodologie de développement et de validation.

Partie V

Conclusions et perspectives

Chapitre 12

Conclusion

12.1 Contributions

Le travail qui est rapporté dans cette thèse a consisté à étudier l'exploitation de certaines formes de structuration de programmes pour aider à leur spécification et à leur validation.

Nous nous sommes principalement intéressés à deux aspects langages : d'un côté des opérateurs permettant l'expression de programmes réguliers (appelés *itérateurs*) , de l'autre une forme de spécification locale de composants (appelée *contrat*).

Itérateurs de tableaux – Les itérateurs sont une forme de description de calculs réguliers sur des tableaux ou des listes proposés en premier lieu pour les langages fonctionnels. Leur introduction dans le langage LUSTRE a été motivée par plusieurs étude de cas, dont celle du ELMU présentée au chapitre 7, où la régularité des programmes, bien que visible dans la spécification n'était pas formellement exprimée et donc n'était pas exploitée par les outils. Nous avons proposé un schéma de compilation vers du code impératif efficace, c'est-à-dire dans lequel :

- on conserve les tableaux utilisés dans le code LUSTRE (au lieu de les expanser en variables indépendantes) ;
- on génère du code avec boucles de type for pour les itérations.

Nous avons aussi proposé des règles d'optimisations permettant de supprimer les tableaux intermédiaires inutiles dans des enchaînements d'itérations. Une des conséquences positives de ce travail a été le développement d'un compilateur par Jean-Louis Colaço, de la société Esterel Technologies, qui prend en compte les itérateurs et implémente les règles d'optimisations que nous proposons. Les itérateurs ont aussi été introduits dans la nouvelle version du langage LUSTRE-V6 académique. Enfin, le débogueur LUDIC prend lui aussi en compte les itérateurs dans une version étendue de LUSTRE-V4.

Nous avons ensuite étudié les possibilités de prise en compte des itérations dans les phases de validation des systèmes. Après avoir expérimenté une technique basée sur l'outil PVS, dans lequel nous avons étendu une sémantique de LUSTRE proposée par Cécile Dumas-Canovas pour prendre en compte les itérateurs de tableaux, nous avons proposé une méthode de renforcement des propriétés itératives en des propriétés d'invariance. Nous avons présenté un algorithme de *slice tableaux* permettant, à partir d'une propriété sur des itérations d'extraire deux propriétés *suffisantes* :

- La première est le cas de base du renforcement de la propriété initiale ;
- La seconde est le cas d'invariance correspondant.

La méthode proposée n'est pas complète : si l'on arrive à prouver la validité des deux propriétés extraites, alors la propriété itérative de départ est validée. Dans le cas contraire, on ne peut pas conclure : soit la propriété originale n'est pas vérifiée, soit la simplification qu'on a utilisée est trop forte.

Contrats pour composants réactifs synchrones – Parallèlement à ce travail sur les itérateurs de tableaux, nous avons proposé l'application du paradigme de *spécification par contrat* aux composants réactifs synchrones. Dans la continuité de travaux et réflexions déjà en cours au sein de l'équipe synchrone depuis quelques temps [Var02], Nous avons proposé une notion de contrat qui allie :

- les avantages pratiques rencontrés notamment dans le domaine des langages orientés objets : séparation claire entre assertion et garantie ; facilité et puissance d'expression par l'utilisation du même langage qu'on utilise pour développer les composants ;
- les avantages théoriques rencontrés dans les modèles formels de composants : sémantique formelle des contrats ; applicabilité des méthodes de vérification formelles classiques.

Notre proposition regroupe ces deux avantages, puisqu'on donne une définition formelle de la notion de contrat, ainsi qu'un support langage puissant : les contrats sont exprimés dans le même langage que les programmes.

Nous avons appuyé notre travail sur les outils de validation existants, en montrant qu'on peut ramener les validations des contrats à des objectifs de preuve exprimés à l'aide d'observateurs. On peut ainsi facilement tirer partie de toute la puissance de ces outils.

Méthodologie – Ces deux derniers points s'inscrivent plus largement dans une méthodologie de développement correct de systèmes. Ce sont des outils de description qui peuvent être utilisés lors des différentes étapes d'un processus de développement classique, comme nous l'avons suggéré au paragraphe 9.5.

Application à des études de cas réalistes – Dans les chapitres 6, 7 et 10, nous avons montré l'intérêt des itérations et des contrats sur deux exemples de taille relativement importante et surtout trouvant leur origine dans des collaborations avec des utilisateurs industriels. Nous avons d'abord présenté l'utilité de l'approche proposée à un niveau « langage » en insistant notamment sur la possibilité d'une méthodologie de conception progressive facilitée par les contrats. Nous avons aussi présenté les gains apportés en terme de validation. Pour les itérateurs, le gain peut être très important en terme de taille de l'observateur.

Les deux études de cas que nous avons commentés ont montré que cette approche était souvent suffisante, étant donnés que les itérations sont le plus souvent utilisées pour exprimer une régularité de donnée et de programme sur laquelle les propriétés à prouvées sont *symétriques*.

Implémentation – D'un point de vue compilation, les itérations ont déjà fait leur preuve (application des schémas de compilation et d'optimisation dans le compilateur expérimental d'Esterel Technologies). En ce qui concerne l'aspect validation de notre travail, nous avons implémenté les algorithmes de traitement de propriétés itératives, ainsi que les manipulations de contrat, dans le prototype GOUPIL, présenté au chapitre 11. Cette implémentation nous a permis de nous assurer de la faisabilité de nos propositions, mais aussi de nous rendre compte de l'intérêt de ces propositions, puisque nous avons appliqué ces manipulations directement sur les études de cas des chapitres 6 et 7.

D'un point de vu développement logiciel, elle a aussi ouvert une réflexion plus large concernant une plate-forme de développement dédiée à LUSTRE, mettant en œuvre la gestion du développement et de la validation d'une application notamment autour de la notion d'arbre de preuve évoquée en 11.6.

12.2 Bilan

12.2.1 Tableaux

Le bilan sur l'aspect langage des tableaux est très satisfaisant : l'étude de cas du ELMU a montré leur utilité et les améliorations apportées en terme de taille du code généré. Le transfert de la méthode de compilation vers le compilateur expérimental d'Esterel Technologies (même s'il ne s'agit pas encore du compilateur utilisé dans SCADE) est un point très positif de cette partie du travail.

D'un point de vue validation, la méthode que nous avons proposée pour *slicer* des propriétés itératives s'adapte assez bien aux propriétés qu'on rencontre dans les programmes LUSTRE : en général, les itérations servent à structurer les programmes de manière *régulière* et les propriétés décrites sont souvent invariantes.

D'un point de vue efficacité de la preuve, la taille de la preuve à traiter *après* slicing est constante par rapport à la taille des tableaux ce qui confère à la preuve un caractère générique sur cette taille. De plus, le traitement que nous proposons ne dépend pas non plus de la taille des tableaux.

En effectuant cette transformation au niveau du langage source, on permet aussi de garder à disposition toutes les méthodes de validation classiques pour prouver les objectifs de preuve résultants. Si on avait choisi d'implémenter ces manipulations à un niveau plus structurel (comme sur le graphe d'états représentant le programme, dans un model-checker par exemple), on aurait été dépendant d'une représentation abstraite d'un programme et on aurait donc réduit les outils cibles.

Comme nous l'avons déjà souligné plusieurs fois, nous n'avons pas cherché à faire de la preuve de programmes itératifs généralisée. La difficulté d'une telle approche réside en général dans la détermination d'un invariant pour la boucle concernée (la construction d'un invariant satisfait par une boucle quelconque est en général indécidable). Nous nous sommes cantonné à considérer que la propriété à prouver *est* invariante.

On pourrait imaginer de compléter la méthode par une analyse de la propriété itérative. L'introduction de l'opérateur forall va dans ce sens : permettre au développeur d'exprimer des propriétés sous certaines formes bien identifiables que l'on sait traiter d'une manière spécifique. On pourrait imaginer une analyse syntaxique de la propriété itérative qui permette de déterminer si la méthode qu'on a proposé sera « *juste* » ou pas. Le problème est de savoir identifier ces formes. Dès le départ, nous avons choisi de ne pas explorer plus avant cette piste, notamment parce que le renforcement que nous proposons nous a semblé en général suffisant pour les types de propriétés qu'on souhaitait exprimer.

Bien sûr, les constructions langages que nous manipulons sont spécifiques à LUSTRE. Les manipulations que nous proposons sont donc elles aussi liées au langage. La méthode n'est ainsi pas facilement applicable à d'autres langages.

12.2.2 Contrats

Les contrats *assume-guarantee* ont potentiellement une portée plus grande que les itérateurs puisqu'ils ne sont pas limités par une syntaxe précise. Nous les avons décrits avec des observateurs, mais, comme le montre le chapitre 3, on pourrait choisir n'importe quelle forme de spécification.

Les manipulations de contrats que nous proposons ont un but très simple : faciliter le processus de développement en automatisant certaines preuves sur les composants. Nous n'avons pas étudié une méthodologie de model-checking compositionnel (comme celle proposée dans certains travaux que nous avons présentés au chapitre 5).

Le model-checking compositionnel a pour but de prouver une *propriété globale* d'un système en
- découpant le système en sous-systèmes ;
- découpant la propriété en sous-propriétés portant sur ces sous-programmes.

Notre méthode n'a pas ce caractère « *total* ». On ne cherche pas à valider une propriété globale (sur le système entier) en la découpant, mais seulement à faciliter la réflexion du développeur en lui fournissant certains outils de base pour valider des parties de son application.

Un des avantages de notre approche est qu'elle repose sur l'utilisation d'outils standards. Les différentes propriétés à valider sont générées sous la formes d'observateurs LUSTRE qui peuvent être fournis aux outils de preuve.

Par ailleurs, l'étude conjointe des contrats avec les itérations a été utile : lorsqu'on cherchait à décrire des contrats sur des programmes réguliers, nous avons trouvé indispensable de pouvoir exprimer ces propriétés à l'aide d'itérations. L'introduction de l'opérateur forall paraît même indispensable pour faciliter l'écriture de ces contrats.

Chapitre 13

Perspectives

Durant les vingt dernières années, l'équipe Synchrone de Verimag a joué un rôle moteur dans la définition des langages synchrones, en proposant et en développant le langage LUSTRE. Parallèlement, elle a aussi contribué au développement des méthodes de validation, notamment le model-checking, l'interprétation abstraite, le test, la simulation et le débogage.

Ces thématiques continuent bien sûr à être explorées aujourd'hui. En parallèle, l'équipe étudie la portée de la « boîte à outils » LUSTRE pour le « *développement de composants hétérogènes pour les systèmes embarqués* ». L'hétérogénéité est à prendre ici au sens large puisqu'on souhaite étudier des systèmes composés de composants *logiciels* ou *matériels*, qui peuvent être décrits de manière *détaillée* ou *abstraite* par rapport au temps, aux données, au mode de communication utilisé (synchrone vs. asynchrone). Le projet *Alidecs* que nous avons déjà présenté plus haut s'inscrit exactement dans cette thématique.

Dans ce contexte, les contrats seront étudiés comme moyen de description des composants. Nous présentons maintenant quelques problèmes que nous souhaiterions aborder dans des travaux futurs. Nous présentons plusieurs sujets précis que nous souhaitons traiter à court terme ainsi qu'une vision plus large de développement correct de logiciel réactifs critiques que nous souhaitons développer à plus long terme.

Nous considérons que le travail autour des tableaux est maintenant terminé. On pourrait étudier d'autres structures régulières, en suivant la même démarche. Mais nous ne considérons pas ces aspects comme les plus intéressants.

13.1 Perspectives concernant les contrats

La première question que nous voulons étudier est l'utilisation qui peut être faite des contrats dans le cadre de la simulation des systèmes (voir paragraphe 13.1.1). Dans cette thèse, nous nous sommes limités à l'étude des contrats pour des composants qui communiquent de manière synchrone. Une seconde perspective intéressante concerne l'extension de la notion de contrat à des modes de communication asynchrones. Nous précisons cette idée au paragraphe 13.1.2. Au paragraphe 13.1.3, nous évoquons une perspective à plus long terme concernant la définition de contrats « dynamiques ». Enfin, nous proposerons l'étude d'une plate-forme de développement de systèmes réactifs intégrant les différentes notions présentées jusque là.

13.1.1 Exécution précoce

Dans les méthodes de test présentées au paragraphe 2.3.4, on considère classiquement le programme à tester comme une boîte noire que l'on peut exécuter pas à pas. On donne aussi une description de l'environnement que l'on utilise pour déterminer les valeurs successives des entrées (voir figure 13.1).

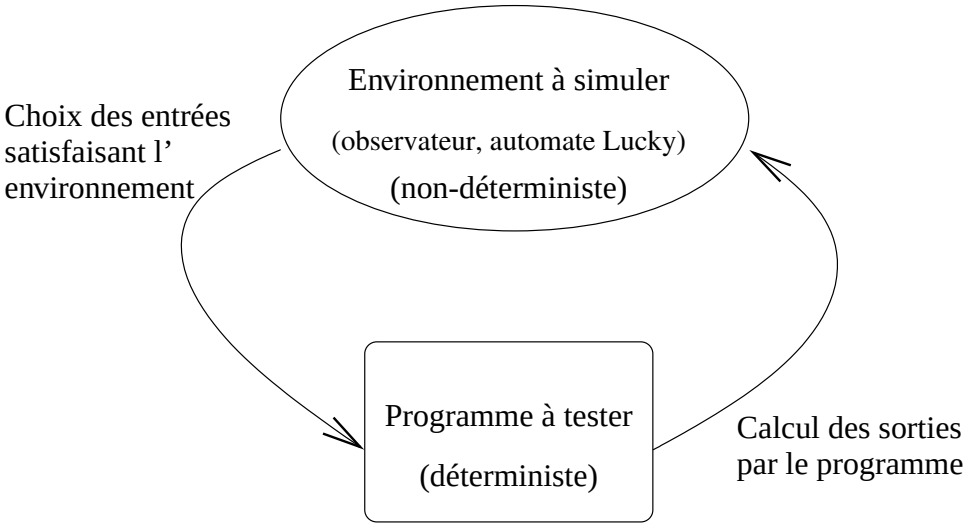


FIG. 13.1 – Schéma de test standard.

L'environnement peut être décrit de plusieurs façons. La plus simple consiste à utiliser un observateur qui décrit une contrainte sur les valeurs d'entrées du programme. Le logiciel de test choisit alors des valeurs qui satisfont cette contrainte. On exécute ensuite le programme à tester, et on récupère les valeurs des sorties correspondantes. Dans sa thèse[Rou04], Yvan Roux a proposé le langage Lutin pour décrire des environnements. Nous en avons présenté les caractéristiques principales au paragraphe 3.4.5.

De manière générale, on voudrait utiliser ces descriptions dans un cadre étendu de simulation. On ne souhaite plus simuler un système seul, en interaction avec son environnement, mais un agencement de plusieurs composants, certains étant décrits de manière totalement déterministe (des noeuds LUSTRE), d'autres dans d'autres modèles synchrones éventuellement non-déterministes (en Lutin, etc.), tous étant branchés pour former un réseau flot-de-donnée. La figure 13.2 donne un exemple de configuration possible.

Dans ce contexte, nous souhaitons étudier les contrats comme un nouveau moyen de description de comportements non-déterministes. Nous voulons par exemple simuler des systèmes dont certains composants sont décrits par des noeuds LUSTRE, d'autres par des programmes Lutin et enfin certains par leur contrat (et de manière générale, par n'importe quelle forme de spécification parmi celles du paragraphe 3.4). La question qui nous intéresse particulièrement est donc de savoir comment simuler un composant spécifié par un contrat *assume-guarantee*.

En pratique, nous avons choisi de décrire nos contrats par des couples d'observateurs. Le premier décrit une contrainte sur les entrées, le second une contrainte sur les entrées/sorties du composant. Une technique pour simuler un tel composant C pourrait être mise en place en utilisant les outils de test de LUSTRE : étant donné les valeurs d'entrées fournies C , on commence par déterminer si l'assertion du contrat est satisfaite. Si c'est le cas, on peut utiliser un solveur de contraintes pour calculer des valeurs de sorties satisfaisant la garantie du contrat du composant. Si l'assertion du contrat est violée,

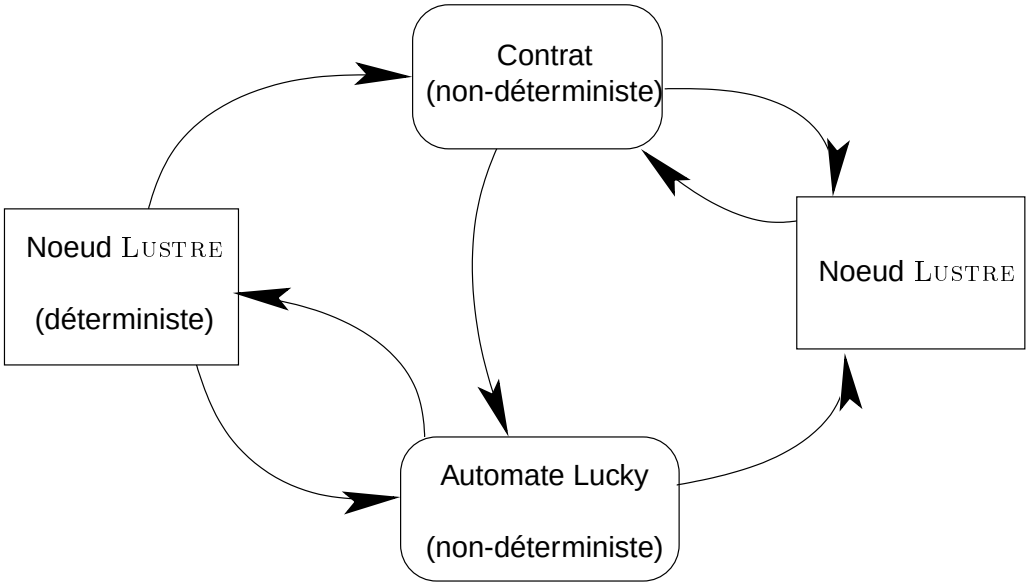


FIG. 13.2 – Schéma de simulation avec des composants décrits dans différents styles.

la simulation est arrêtée et l'on peut indiquer à l'utilisateur que les composants qui fournissent ses entrées à C violent le contrat de C.

Dans un avenir proche, nous souhaitons étudier cet aspect simulation concernant les contrats et mettre en pratique cette approche pour LUSTRE.

13.1.2 Contrats pour composants asynchrones

La seconde perspective directe de notre travail concerne l'extension de notre notion de contrats à des composants asynchrones. Dans [HB02], Halbwachs et Baghdadi ont proposé une méthode pour modéliser des systèmes asynchrones à l'aide de formalismes synchrones. Il ont notamment montré comment modéliser différents types de communications asynchrones (mémoire partagée, rendez-vous, FIFO, etc.) en utilisant les horloges de LUSTRE. Le cadre général de ces travaux est celui des GALS (*Systèmes Globalement Asynchrones Localement Synchrones*) dans lesquels on décrit des systèmes asynchrones par des processus synchrones composés de manière asynchrone. Dans la continuation de cette thèse, nous aimerions étudier la signification des contrats pour de tels systèmes.

Dans ce que nous avons présenté, les entrées/sorties d'un noeud dépendent éventuellement d'horloges différentes de l'horloge de base. Dans le contrat d'un composant, on parle des entrées/sorties en fonction de leur horloge respective. Par ailleurs les contrats de synchronisation font partie des contrats fonctionnels : on peut décrire dans nos contrats des contraintes sur les horloges puisque ce sont des entrées/sorties du composant.

Lorsqu'on travaille dans un mode de communication asynchrone, la synchronisation n'est pas forcément contrôlée par le composant. Par exemple, il est possible que le composant soit construit pour être connecté à un autre composant par l'intermédiaire d'une file FIFO. Si le composant ne gère pas lui-même cette FIFO, il doit l'indiquer dans son contrat de synchronisation. D'un autre côté, on utilise des composants spécifiques qui mettent en place les modes de communications nécessaires (FIFO, mémoire partagée, rendez-vous, etc.). On souhaitera alors spécifier les contrats de ces composants.

De manière générale, on veut déterminer l'impact d'un mode de communication asynchrone du

point de vue de la spécification par contrat des composants.

13.1.3 Contrats « dynamiques »

Dans plusieurs travaux, Frédérique Boussinot a étudié de manière générale la programmation réactive, basée sur 3 notions fondamentales :

- la *concurrence* : les différents composants d'un système représentent des processus qui s'exécutent en parallèle, de manière concurrente ;
- Une notion d'*instants* : il est possible de définir une échelle de temps commune à tous les processus ;
- la *création dynamique* de processus : il est possible de créer dynamiquement des processus, qui interagissent alors avec les processus déjà en place dans leur environnement.

Plusieurs implémentations ont été proposées, parmi lesquelles on peut citer Reactive-C [Bou89], les SugarCubes [BS97], les FairThreads [Bou01], ou encore Reactive-ML [MP].

On peut adapter ces idées au modèle flot-de-donnée du langage LUSTRE. Pour cela, on doit fournir une construction dynamique de réseau par ajout de noeuds (cette approche a été étudiée par Jacques Ndjeng Ndjeng dans son DEA à Verimag, en 2003). Lorsqu'on ajoute un composant en cours d'exécution, certaines entrées sont branchées sur d'autres composants, d'autres constitueront des entrées du système global.

Nous aimerions étudier une notion de contrat qui porte sur cet aspect *dynamique* du comportement d'un système. Par exemple, on pourrait avoir envie de spécifier que certaines entrées d'un composant ne doivent jamais être branchées sur un autre composant, mais qu'elle doivent toujours être connectées directement à un environnement physique. Cette contrainte est dynamique : il faudra s'assurer pendant l'exécution du système qu'aucune modification dynamique des branchements des composants entre eux ne violera cette propriété. Ces propriétés ne portent pas sur le comportement des composants, mais plus sur la *construction dynamique d'architectures*.

Nous n'avons pour l'instant qu'une vision très limitée de cette notion de contrat dynamique. Mais nous aimerions étudier dans un futur plus lointain quels types de propriétés on doit pouvoir exprimer et surtout *comment* on pourrait les exprimer.

13.2 Vers une plate-forme de développement pour les systèmes réactifs

Comme l'avait déjà souligné Fabien Gaucher dans sa thèse [Gau03], une perspective intéressante serait d'appliquer les manipulations de contrats à l'outil SCADE, qui a la particularité de représenter les programmes de manière graphique comme des réseaux d'opérateurs.

Nous l'avons souligné à plusieurs reprises, le prototype que nous avons développé durant cette thèse n'a pas vocation d'outil complet. Bien sûr, il nous a permis de valider les manipulations que nous avons proposées d'une manière concrète. De plus, et c'est peut être le point le plus prometteur, il a montré l'utilité d'une plate-forme de développement dédiée à LUSTRE.

Les méthodes de validation standards, qu'elles soient totalement automatiques comme le model-checking ou semi-automatiques comme les systèmes de déductions logiques, ont montré leur limite. Les premières rencontrent rapidement des problèmes de complexité sur des exemples de taille moyenne. Les secondes demandent une expérience et une connaissance des logiques sous-jacentes très importantes et sont déstinées à des utilisateurs experts.

Il est aujourd'hui important de proposer des méthodes tirant partie de ces différentes approches. Elles doivent être intégrées au sein de logiciels interactifs permettant la manipulation des spécifica-

tions (une peu dans l'esprit de notre prototype). Les extensions que nous avons proposé plus haut permettront quant à elle d'appliquer nos techniques à une variété plus grande de systèmes.

Partie VI

Annexes

Annexe A

Génération de code pour les itérateurs LUSTRE

Au paragraphe A.1, nous donnons une représentation abstraite des programme LUSTRE sous la forme d'arbres abstraits. Nous donnons ensuite (section A.2) l'algorithme général de parcours de l'arbre du noeud principal (ANPrincipal) pour générer le code C correspondant.

A.1 Représentation des programmes

Nous représentons les programmes LUSTRE sous la forme d'arbre abstrait (de type AA_Nœud). Ils sont réunis dans une liste indicée notée ListeNœuds. Parmi ces nœuds, on identifie le nœud principal, noté NPrincipal, son arbre abstrait étant noté ANPrincipal.

Chaque arbre A a pour racine le nom du nœud, noté A.nom et pour branches :

- La liste des entrées du nœud, notée A.Entrées ;
- La liste des sorties du nœud, notée A.Sorties ;
- La liste des variables locales du nœud, notée A.Vars_Locales ;
- La liste des équations du nœud, notée A.ListeEquations.

Toutes les listes manipulées sont indicées. Les listes A.Entrées, A.Sorties, A.Locales sont des listes de couples <nom,type>. Elles seront aussi regroupées sous le nom A.Paramètres_Formels, pour les distinguer des paramètres effectifs lors des itérations de nœuds. La liste A.ListeEquations est une liste de triplets <var, exprInit, exprStep>. Si Eq est une équation, on note

- Eq.var la liste de variable qu'elle définit ;
- Eq.exprInit l'expression définissant la valeur de Eq.var à l'instant initial de l'exécution ;
- Eq.exprStep l'expression définissant la valeur de Eq.var lors des instant suivants.

La séparation des expressions Eq.exprInit et Eq.exprStep est simple à réaliser sur un programme LUSTRE standard et facilite l'expression de l'algorithme de génération de code. Lorsqu'on ne s'intéressera pas à la séparation de exprInit et exprStep (notamment dans l'algorithme de traitement des itérations du chapitre 8), on pourra utiliser Eq.expr qui regroupe les deux.

Représentation des expressions – Une expression E peut être de différents types :

type 1 Une simple variable, E.nom désigne son nom .

type 2 Une opération arithmétique ou booléenne, potentiellement parenthésée. On définit alors E par E.Opérande1, E.Opérateur et E.Opérande2 s'il s'agit d'une opération binaire

```
        A.nom = "ADD"
        A.Entrées = [<A,boolˆ10>,<B,boolˆ10>]
        A.Sorties = [<S,boolˆ10>,<overflow,bool>]
        A.Variables_Locales = ∅
        A.ListeEquations = [<[overflow,S],E>]
avec
        E.type = 5
        E.typeIter = map_red
        E.nœud = FULL_ADD
        E.taille = 10
        E.init = 0
        E.Paramètres = [A,B]
```

FIG. A.1 – Arbre abstrait du nœud ADD.

et par E.Opérateur et E.Opérande s'il s'agit d'une opération unaire, où E.Opérande1, E.Opérande2 et Opérande sont à leur tour des expressions de type 1 ou de type 2.

type 3 Une liste d'expressions. On peut, en LUSTRE définir plusieurs variables avec une seule équation. Écrire a,b = (a'+1, b'+1); équivaut à écrire a = a'+1 et b = b'+1. Les éléments de la liste d'expressions peuvent être de n'importe quel type (1 à 5). On identifie par E.ListEx, la liste des expressions. E.taille est le nombre d'expressions dans la liste ;

type 4 Une expression conditionnelle. E est décrite par E.Condition, E.Alors, E.Sinon qui sont forcément des expressions (E.Condition étant une expression booléenne) ;

type 5 Une itération, qui est alors accompagnée de toutes les informations utiles (voir ci-dessous) ;

type 6 Une application de l'opérateur pre. E.var représente alors la variable à laquelle pre est appliquée.

Représentation des itérations – Soit E une expression de type 5 (itération). Quel que soit le genre d'itération, E comprend toujours : E.typeIter (le type d'itération, qui prend sa valeur parmi map, red, fill et map_red), E.nœud (le nœud itéré), E.taille (la taille de l'itération, qui est un entier). Dans les cas du map, du red et du map_red, on donne aussi les paramètres entrées "tableaux" de l'itération. Il s'agit d'une liste de noms, notée E.Paramètres. Dans les cas du fill, du red et du map_red, on trouve aussi la valeur d'initialisation de l'itération, notée E.init.

A titre d'exemple, on trouvera à la figure A.1 l'arbre abstrait du programme ADD.

A.2 Algorithme de génération de code

A.2.1 Préliminaires

Afin de simplifier la description de l'algorithme de génération de code, nous faisons les suppositions suivantes :

- Les équations des noeuds LUSTRE traités sont *ordonnées* ;
- L'opérateur pre est toujours appliqués à des noms de variables mais jamais à des expressions quelconques.

Fichiers destination – L'algorithme que nous décrivons ci-après génère du code impératif dans un fichier 'destination' appelé fichierPrincipal. Nous utilisons aussi des fichiers temporaires fichierInitialisation et fichierStep afin de faciliter la génération du code *initialisation* et *step*.

Primitives d'écriture – Nous utilisons dans les algorithmes qui suivent une primitive d'écriture notée Écrire. Elle prend une liste de paramètres aussi bien de type chaîne qu'entier. Le dernier paramètre est toujours de type Fichier.

A.2.2 Action principale

Générer_code, décrite à la figure A.2, est la fonction principale du générateur de code. Dans l'ordre, elle réalise les actions suivantes :

- Générer l'entête du programme ;
- Générer les déclarations de variables locales et de variables "mémoires" correspondant aux occurrences de l'opérateur pre dans le programme LUSTRE ;
- Générer le code correspondant aux équations du noeud principal ;
- Générer le code correspondant à la mise à jour des mémoires du programme.

```
Générer_code(ANPrincipal : AA_Noeud){
    Générer_entête(ANPrincipal);
    Ecrire("{",fichierPrincipal);
    Générer_varLocales_etMémoires(ANPrincipal,0);
    Générer_equations(ANPrincipal);
    Générer_miseAJourMémoires(ANPrincipal);
    Ecrire("}",fichierPrincipal);
}
```

FIG. A.2 – Fonction principale du générateur de code

A.2.3 Génération de l'entête de la fonction principale

L'entête de la fonction générée correspond simplement à son nom et à ses paramètres. Nous utilisons pour cela une fonction Générer_déclarations qui génère le code C correspondant à une liste de déclaration qu'on lui fournit. Le code produit pour les sorties est légèrement différent de celui produit pour les entrées, mais nous ne rentrons pas plus dans ces détails qui ne sont pas le but de ce travail. La fonction Générer_entête de la figure A.3 réalise cette partie.

```
Générer_entête(ANPrincipal : AA_Noeud){
    Écrire("void", ANPrincipal.name, "(", fichierPrincipal);
    Générer_déclarations(ANPrincipal.Entrées);
    Générer_déclarations(ANPrincipal.Sorties);
}
```

FIG. A.3 – Fonction de génération de l'entête (entrées/sorties) de la fonction principale

A.2.4 Identification et génération des variables locales et mémoires nécessaire

Une partie plus intéressante concerne la génération des déclarations dans le code C correspondant d'une part aux variables locales du noeud principal LUSTRE (et des noeuds itérés) ainsi que des mémoires nécessaires aussi bien pour le noeud principal que pour les différentes *instance* des noeuds itérés. Il faut en effet une version de la mémoire nécessaire à un noeud itéré pour chaque étage de l'itération.

La génération des variables locales du noeud principal se fait très simplement à l'aide de la fonction Générer_déclarations déjà utilisée pour le entrées/sorties du programme.

Nous découpons ensuite le travail en deux :

- d'un coté, nous allons générer déclarations de mémoires nécessaires dans le noeud principal ;
- de l'autre, nous générerons les mémoires nécessaires dans les noeuds itérés.

```
Générer_varLocales_etMémoires(ANPrincipal : AA_Noeud;
      niveau : entier){
      Générer_déclarations(ANPrincipal.VarLocales);
      Générer_mémoiresPrincipal(ANPrincipal);
      Générer_mémoiresNoeudItérés(ANPrincipal, [], niveau);
}
```

FIG. A.4 – Génération des déclarations pour variables locales et mémoires du programme

A.2.5 Génération des mémoires utiles au noeud principal

Dans le noeud principal, nous parcourant le graphe d'opérateur. Pour chaque occurrence de l'opérateur pre (qui n'est appliqué dans notre cas qu'à des variables), on génère une variable pre_x représentant la valeur à l'instant précédent de la variable x à laquelle s'applique pre. Si x est un tableau, on doit générer un tableau de mémoire de la même taille que x.

```
Générer_mémoiresPrincipal(ANPrincipal : AA_Noeud){
    Pour chaque "pre(x)" dans ANPrincipal.ListeEquations{
        Écrire(x.type," pre_", x.nom, fichierPrincipal);
        si(x.type.nbDimensions>0)
        alors Écrire("[", fichierPrincipal);
            Pour i entre 0 et x.type.nbDimensions faire{
                Écrire(x.type.dimensions[i], fichierPrincipal);
            }
            Écrire("]", fichierPrincipal);
        Écrire(";")
    }
}
```

FIG. A.5 – Génération des mémoires utiles au noeud principal

A.2.6 Génération des déclarations pour les mémoires des noeuds itérés

En ce qui concerne les noeuds itérés, le traitement est légèrement plus compliqué. Nous le décrivons à l'aide d'une action récursive Générer_mémoiresNoeudsItérés. Le premier appel de cette action parcourt les équations utilisant une itération (donc manipulant une expression de type 5) et pour chacune de ces itérations, appelle la fonction Générer_mémoires1NoeudItéré qui va se charger de générer les déclarations de variables pour *un* noeud itéré. Il se peut que des itérations soient utilisées dans le noeud itéré que l'on considère présentement. D'où l'utilité de l'appel récursif) Générer_mémoiresNoeudsItérés. Notons pour cela l'incrémentation de la valeur niveau initialisée à 0 dans Générer_code qui indique le niveau d'imbrication de l'itération en cours de traitement. Cette variable est utile, nous le verrons plus loin, pour la génération des bons nombres de dimensions des tableaux de mémoire manipulées par les itérations.

```
Générer_mémoiresNoeudsItérés(ANPrincipal : AA_Noeud;
                             tailleItération : entier[niveau];
                             niveau : entier){
    Pour chaque équation Eq dans ANPrincipal.ListeEquations
       telle que Eq.expression.type=5{
          Générer_mémoires1NoeudItéré(Eq.expression.noeud,
                                      addElement(Eq.expression.taille,
                                                 tailleItération),
                                      niveau);
          Générer_mémoiresNoeudsItérés(Eq.expression.noeud,niveau+1);
    }
}
```

FIG. A.6 – Génération des déclarations pour les mémoires des noeuds itérés

La génération des déclarations pour les mémoires utiles aux noeuds itérés se fait à l'aide de deux fonctions Générer_mémoires1Noeuditéré et Générer_mémoireItéré. La première parcours le graphe d'opérateur du noeud itéré ANItéré. Elle appelle la seconde pour chaque occurrence de l'opérateur pre qui se charge de générer effectivement le code correspondant.

```
Générer_mémoires1Noeuditéré(ANItéré : AA_Noeud;
                            tailleItération : entier[niveau];
                            niveau ; entier){
    Pourchaque "pre(x)" dans ANPrincipal.ListeEquations{
        Générer_mémoireItéré(ANItéré, x,tailleItération,niveau);
    }
}
```

FIG. A.7 – Génération des déclarations pour les mémoires d'*un* noeud itéré

Les variables générées par Générer_mémoireItéré sont forcément de type tableau. Leur nombre de dimensions dépend du niveau d'imbrication des itérations (voir les exemples de la page 83). Le nombre d'éléments par dimensions dépend de la *taille* des itérations correspondantes. Afin de pouvoir générer le code correct pour la déclaration des mémoires correspondantes, on a fait *descendre* les

tailles des itérations (le nombre d'éléments des tableaux) en utilisant le paramètre tailleItération. Pour chaque niveau i, ce tableau contient (à la case tailleItération[i]) le nombre d'éléments dans la dimension i.

Pour chaque variable x d'un noeud itéré à laquelle est appliqué l'opérateur pre nous générons un tableau de mémoire appelé pre_x. Si en plus x est une variable locale, il nous faut un tableau supplémentaire _x qui permet de sauvegarder les valeurs des différentes occurrences de x entre le calcul des sorties et la mise à jour de la mémoire pre_x.

```
Générer_mémoireItéré(ANItéré : AA_Noeud;
                     x : AA_VariableDecl;
                     tailleItération : entier[niveau];
                     niveau : entier){
    Écrire(x.type," pre_", x.nom, "[", fichierPrincipal);
    Pour i entre 0 et niveau faire{
        Écrire(tailleItération[i], fichierPrincipal);
    }
    Écrire("]",fichierPrincipal);
    si x non entrées/sorties 'tableau' de ANItéré
    alors Écrire(x.type," _", x.nom, "[", fichierPrincipal);
        Pour i entre 0 et niveau faire{
            Écrire(tailleItération[i], fichierPrincipal);
        }
    Écrire("]", fichierPrincipal);
}
```

FIG. A.8 – Génération d'une déclaration pour une occurrence de pre dans un noeud itéré

A.2.7 Génération des équations

Nous nous intéressons maintenant à la génération du code pour les équations du noeud principal. Il faut penser à générer à la fois le code correspondant à l'instant initial de l'exécution et le code valable pour le reste de l'exécution. Nous allons générer ces deux parties dans des fichiers différents (fichierInit et fichierStep). On commence par générer la condition (on teste si on est à l'instant initial (avec if(init)) et le passage aux instants suivant (avec init = 0). Ensuite, pour chaque équation du noeud principal, on génère les cas init et step dans le fichier correspondant. Pour cela on reconstruit une équation avec la partie gauche de l'équation Eq et soit l'expression initExpr soit l'expression stepExpr que l'on fournit à l'action Générer_Equation. On finit par concaténer les deux fichiers à la fin du fichier destination principal fichierPrincipal en utilisant une action Ajouter.

L'action Générer_Equation (voir figure A.10) teste le type d'expression utilisée dans la partie gauche de l'équation Eq. Si l'expression est une expression binaire, unaire (y compris une occurrence de l'opérateur pre, ou simplement un nom de variable, on utilise l'action Générer_SimpleEquation décrite à la figure A.11. S'il s'agit d'une liste d'expression (définition de plusieurs variables à l'aide d'une seule équation), on génère le code correspondant à chaque de ces définitions avec Générer_Equation. Pour une équation conditionnelle, on utilise l'action Générer_Equation_Conditionnelle donnée à la figure A.12. Enfin, pour une itération, on utilise l'action Générer_Itération de la figure A.13.

```
Générer_equations(ANPrincipal : AA_Noeud){
      Écrire("if(init){", fichierInit);
      Écrire("else{", fichierStep);
      Pour chaque Eq dans ANPrincipale.ListeEquations() faire{
            Générer_Equation(ANPrincipal.Paramètres_formels,
                             ANPrincipal.Paramètres_formels,
                             <Eq.var,Eq.initExpr>,
                             0,
                             fichierInit);
            Générer_Equation(ANPrincipal.Paramètres_formels,
                             ANPrincipal.Paramètres_formels,
                             <Eq.var,Eq.stepExpr>,
                             0,
                             fichierStep);
      }
      Écrire("init = 0", fichierInit);
      Écrire("}", fichierInit);
      Écrire("}", fichierStep);
      Ajouter(fichierInit,fichierPrincipal)
      Ajouter(fichierStep,fichierPrincipal)
}
```

FIG. A.9 – Génération du code pour l'ensemble des équations

```
Générer_Equation(For : Paramètres,
                 Eff : Paramètres,
                 Eq : Equation,
                 niveau : entier,
                 fichier : Fichier){
    selon(E.expression.type){
        si(1 ou 2 ou 6) alors
          Générer_SimpleEquation(For, Eff, Eq, niveau, fichier);
        si 3 alors
          Pour i entre 0 et Eq.expression.taille faire{
              Générer_SimpleEquation(For, Eff,
                                     <Eq.var[i],
                                     Eq.expression.ListEx[i]>,
                                     niveau, fichier);
        si 4 alors
          Générer_Equation_Conditionnelle(For, Eff, Eq, niveau, fichier);
        si 5 alors
          Générer_Itération(For, Eff, Eq, nivau+1, fichier);
    }
}
```

FIG. A.10 – Génération du code pour *une* équation

A.2.7.1 Cas d'une équation simple (type 1, 2 ou 6)

L'action Générer_SimpleEquation (figure A.11) permet de générer le code pour une équation dont la partie droite est

- une expression binaire ;
- une expression unaire ;
- ou une occurrence de l'opérateur pre.

On commence par générer la partie gauche de l'affectation correspondante, puis on utilise Générer_Expression (figure A.11) pour générer la partie droite en parcourant l'arbre syntaxique de l'expression E.expression. Ces actions sont aussi utilisées lorsqu'on génère le code pour un noeud itéré. Le paramètre niveau permet dans ce cas de savoir combien d'indice on doit génère sur chaque nom de variable (soit en partie gauche, soit pour une expression de type 1). Dans le cas d'une expression pre(x), on doit générer le nom de la variable mémoire pre_x comme indiqué plus haut.

A.2.7.2 Cas d'une équation conditionnelle (type 4)

Dans le cas d'une équation conditionnelle de la forme

V = if cond then E_Alors else E_Sinon

on va commencer par construire les équations E_Alors et E_Sinon correspondant aux deux branches de la conditionnelle. On va ensuite générer le code pour la condition, puis générer le code pour E_Alors et pour E_Sinon. L'action Générer_Equation_Conditionnelle implémente cela.

A.2.7.3 Génération des équations d'un noeud itéré (équation de type 5)

Pour une itération, on va générer tout d'abord le code pour son initialisation (seulement dans le cas d'un red, fill ou map_red. Puis, on va générer la boucle for qui parcourt les tableaux. A l'intérieur de cette boucle, on va générer le code correspondant à chaque équation du noeud itéré. L'action correspondante Générer_Itération est donnée à la figure A.13.

A.2.8 Génération du code de mise à jour des mémoires

La génération du code de mise à jour des mémoires suit exactement le principe utilisé pour la génération des variables correspondant à l'utilisation de l'opérateur pre. L'action Générer_miseAJourMémoires génère d'une part la mise à jour des mémoires du noeud principal (avec Générer_miseAJour_mémoiresPrincipal) et d'autre part la mise à jour des mémoires itérées (avec Générer_miseAJour_mémoiresNoeudItérés). Pour chaque expression de type pre(x) utilisée dans un noeud itéré, on copie les valeurs contenues dans les tableaux _x vers les tableaux pre_x correspondant. Symétriquement par rapport à la génération des déclarations de variables correspondant aux occurrences de pre vus plus haut, on définit aussi les fonctions :

- Générer_miseAJour_mémoires1Noeuditéré ;
- Générer_misAJour_mémoireItéré.

Les fonctions correspondantes sont données à la figure A.14.

```
Géner_SimpleEquation(For : Paramètres,
                     Eff : Paramètres,
                     E : Equation,
                     niveau : entier,
                     fichier : Fichier){
    variable_effective = Correspondance(E.variable, Eff);
    Ecrire(variable_effective);
    Pour k de 0 à niveau Ecrire("[i",k,"]");
    Ecrire("=");
    Générer_Expression(For, Eff, E.expression, niveau);
}

Générer_Expression(For : Paramètres,
                   Eff : Paramètres,
                   E : Expression,
                   niveau : entier,
                   fichier : Fichier){
    switch(E.type){
        case 1 :
            variable_effective = Correspondance(E.nom, Eff);
            Écrire(variable_effective, fichier);
            Pour k de 0 à niveau Écrire ("[i",k,"]");
        case 2 :
            si Est_unaire(E)
            alors Écrire(Opérateur, fichier);
                Générer_Expression(For, Eff, E.Opérande, niveau, fichier);
            sinon Générer_Expression(For, Eff, E.Opérande1, niveau, fichier);
                Écrire(Opérateur, fichier)
                Générer_Expression(For, Eff, E.Opérande2, niveau, fichier);
        case 6 :
            variable_effective = Correspondance(E.nom, Eff);
            Écrire("pre_", variable_effective, fichier);
    }
}
```

FIG. A.11 – Génération d'une expression simple

```
Générer_Equation_Conditionnelle(For : Paramètes,
                                Eff : Paramètres,
                                E : Equation,
                                niveau : entier,
                                fichier : Fichier){
    E_Alors : Equation;
    E_Sinon : Equation;
    E_Alors.variable = E.variable;
    E_Alors.expresion = E.expression.Alors;
    E_Sinon.variable = E.variable;
    E_Sinon.expresion = E.expression.Sinon;
    Ecrire("if(", fichier);
    Générer_Expression(For, Eff, E.Condition, niveau, fichier);
    Ecrire("){", fichier);
    Générer_Equation(For, Eff, E_Alors, niveau, fichier);
    Ecrire("}else{", fichier);
    Générer_Equation(For, Eff, E_Sinon, niveau, fichier);
    Ecrire("}", fichier);
}
```

FIG. A.12 – Génération de code pour une équation conditionnelle

```
Générer_Itération(For : Paramètes,
                  Eff : Paramètres,
                  E : Equation,
                  niveau : entier){
    Noeud_Itéré : AA_Noeud;
    AA_Noeud = E.noeud;
    Ecrire("{");
    si(E.expression.type=(map_red or red or fill))
    alors Générer_Equation_Initialisation(E.expression.init, niveau);
    Ecrire("for(i",niveau,=0",";i",niveau,"=",E.expression.taille,";i0++){")
    Pour chaque équation Eq dans Noeud_Itéré faire{
        Générer_Equation(Noeud_Itéré.Paramètes_Formels,
                         E.Paramètres,
                         Eq,
                         niveau);
    }
    Ecrire("}");
    Ecrire("}");
}
```

FIG. A.13 – Génération du code pour une itération

```
Générer_miseAJourMémoires(ANPrincipal : AA_Noeud){
    Générer_miseAJour_mémoiresPrincipal(ANPrincipal);
    Générer_miseAJour_mémoiresNoeudItérés(ANPrincipal, [], niveau);
}

Générer_miseAJour_mémoiresPrincipal(ANPrincipal : AA_Noeud){
    Pour chaque "pre(x)" dans ANPrincipal.ListeEquations{
        Écrire("pre_", x.nom, " = ", x.nom, ";");
    }
}
        ____________________________________________

Générer_misAJour_mémoiresNoeudItérés(ANPrincipal : AA_Noeud;
                                     tailleItération : entier[niveau];
                                     niveau : entier){
    Pour chaque équation Eq dans ANPrincipal.ListeEquations
      telle que Eq.expression.type=5{
        Écrire("for(i', niveau, "=0;i", niveau, "=",
            tailleItération[niveau-1], ";i", niveau,
            "++"){", fichierPrincipal);
        completeTailleItération = addElement(Eq.expression.taille,
                                             tailleItération);
        Générer_miseAJour_mémoires1NoeudItéré(Eq.expression.noeud,
                                              completeTailleItération,
                                              niveau);
        Générer_miseAJour_mémoiresNoeudItérés(Eq.expression.noeud,
                                              completeTailleItération,
                                              niveau+1);
        Écrire("}", fichierPrincipal);
    }
}
        ____________________________________________

Générer_miseAJour_mémoires1Noeuditéré(ANItéré : AA_Noeud;
                                      tailleItération : entier[niveau];
                                      niveau : entier){
    Pour chaque "pre(x)" dans ANPrincipal.ListeEquations{
        Générer_miseAJour_mémoireItéré(ANItéré, x,tailleItération,niveau);
    }
}
        ____________________________________________

Générer_misAJour_mémoireItéré(ANItéré : AA_Noeud;
                              x : AA_VariableDecl;
                              tailleItération : entier[niveau];
                              niveau : entier){
    Écrire("pre_", x.nom", fichierPrincipal);
    Pour k entre 0 et niveau faire{
        Écrire("[i",k,"]", fichierPrincipal);
    }
    Écrire(" = ", x.nom", fichierPrincipal);
    Pour k entre 0 et niveau faire{
        Écrire("[i",k,"]", fichierPrincipal);
    }
}
```

FIG. A.14 – Mise à jour des mémoires

Annexe B

Algorithme de manipulation d'itérations

Nous présentons maintenant l'algorithme mettant en application les règles de construction d'obligations de preuve vu au paragraphe 8.5. Pour manipuler les programmes, nous utilisons la représentation introduite au chapitre A. Nous donnons plus loin quelques des primitive de manipulation de cette représentation.

B.1 Principe général

Une propriété est un programme LUSTRE Obs avec une unique sortie booléenne ok. On veut construire un nouvel observateur (soit newObs) qui exprime le renforcement de la propriété suivant les règles données dans les paragraphes précédents. Comme nous l'avons indiqué, on va parcourir le graphe de dépendance de ok et générer les équations init et inv correspondant à chaque itération rencontrée.

Ce parcours est effectué de la manière suivante : on commence par traiter l'équation définissant ok. Ensuite, on traite toutes les équations définissant des variables utilisées directement dans l'expression définissant ok. On recommence jusqu'à ce que chaque équation du nœud ait été traitée.

A chaque équation rencontrée, l'algorithme complète le nouvel observateur en y ajoutant les équations et éventuelles variables locales ou entrées nécessaires. Ces variables sont de deux types :

- Soit elles correspondent à un tableau sur lequel est appliquée une itération (les variables elt_T4_init et elt_T4_inv de l'exemple 8.5.4.1 correspondait au tableau T4), ou à une variable accumulateur d'une itération (les variables varOut_init et varOut_inv correspondaient à l'accumulateur de l'itération fill≪T;size≫) ;
- Soit elles sont introduites dans l'algorithme afin de faciliter l'expression de la propriété construite et sa manipulation ultérieure. Par exemple, les variables propRankN et propRankNp1 pourraient ne pas être utilisées. Cependant elles rendent la lecture du nouvel observateur plus facile. De manière générale, elles sont utilisées pour identifier des sous-propriétés particulières de la propriété générée.

Les équation générées sont aussi de deux types :

- d'un coté les équations définissant les variables « *liées* » aux itérations ;
- de l'autre les équations effectuant les connections logiques entre les différentes parties de la propriété générée.

Dans les paragraphes suivants, on décrit tout d'abord quelques primitives de manipulation de programmes (ajout de variables, d'équations) puis l'algorithme de génération d'obligations de preuve

lui-même.

B.2 Manipulation des arbres abstraits

De plus, Nous définissons des primitives de manipulations des programmes qui vont nous permettre de construire le nouvel observateur.

Ajout d'une variable à un nœud – On se donne les primitives suivantes pour ajouter une variable aux ensembles d'entrées, de sortie ou de variables locales d'un nœud :

ajouterEntree(AN : AA_Nœud, V : Variable);
– état final : V a été ajoutée aux entrées du nœud AN

ajouterSortie(AN : AA_Nœud, V : Variable);
– état final : V a été ajoutée aux sorties du nœud AN

ajouterLocale(AN : AA_Nœud, V : Variable);
– état final : V a été ajoutée aux variables locales du nœud AN

Ajout d'une équation à un nœud – De la même façon, on propose la primitive ajouterEquation pour permettre de rajouter une équation à un nœud abstrait :

ajouterEquation(AN : AA_Nœud, Eq : Equation);
– état final : Eq a été ajoutée aux équations du nœud AN

Récupérer l'équation définissant une variable – On souhaitera par endroit connaître l'équation définissant une variable dont on connaît le nom. On donne pour cela la primitive suivante :

trouverEquationPourVar(AN : AA_Nœud, V : Variable)
retourne (Eq : Equation);
– ei : Il existe une équation définissant V dans AN
– ef : Eq est l'équation définissant V dans AN

Une équation a-t-elle déjà été traitée ? – Pour chaque équation, on doit pouvoir savoir si elle a déjà été traitée par l' algorithme. Pour cela, on fournit la primitive suivante :

estTraitee(Eq : Equation) retourne (traitee : booléen);
– ef : traitee vaut vrai si l'équation a déjà été traitée

B.3 Algorithme

Pré-requis – Nous donnons ci-dessous une liste de pré-requis à l'algorithme proposé. Ces pré-requis sont tous d'ordre syntaxique et peuvent être résolus facilement.

- On suppose qu'il n'y a pas d'itérations imbriquées du type map<<map<<;>>;>>. Cela ne limite pas la portée de l'algorithme, car on peut toujours réécrire des itérations imbriquées en rajoutant des nœuds supplémentaires ;
- On suppose qu'il n'y a pas non plus d'itérations d'opérateurs, du style map<<+;10>> mais que tout opérateur a été préalablement encapsulé dans un nœud. Cela pourra simplifier l'algorithme par endroit ;

- On ne prend pas en compte les appels de nœud : dans le cas où un appel de nœud est découvert pendant la transformation, on effectue la transformation en assertion vue plus haut sur l'observateur original et ensuite on applique l'algorithme qui est présenté ci-dessous sur le nouvel observateur construit ;
- L'assertion considérée ne contient pas d'appel de nœud. Dans le cas où on a rencontré un appel de nœud, on a notamment expansé les nœuds assumeN et guaranteeN, spécifiant le contrat du nœud appelé, pour faire apparaître les réductions éventuelles .
- On suppose que des itérations sont utilisées à la fois pour calculer la propriété *et* pour calculer l'assertion. Si aucune itération n'est utilisée, la transformation proposée n'a aucune utilité. Si l'assertion n'utilise pas d'itération, sa transformation est inutile : on générera une itération identique à celle d'origine .
- Il est possible que ces itérations définissent des variables locales nécessaires à la propriété ou à l'assertion et pas simplement la propriété ou l'assertion elles-mêmes. On se muni alors d'une action à résultat booléen estCalculéeparRed permettant de déterminer si l'expression calculant une variable donnée est calculée directement par un red. Nous devons différencier ces deux cas, leur traitement étant légèrement différent (voir plus loin l'utilité des actions traiterEqPrincipale_Red et traiterEqPrincipale).

Procédure principale – La procédure principale traiterObservateur implémente la transformation généralisée qui prend en compte une éventuelle assertion. À partir d'un observateur Obs, elle construit un observateur Obs en appliquant les transformations présentées au paragraphe 8.5.

traiterObservateur effectue dans l'ordre les actions suivantes :

- le traitement de l'équation définissant la sortie de l'observateur. Pour cela, on définit les actions traiterEqPrincipale_Red et traiterEqPrincipale qui permettent de traiter l'équation principale lorsqu'elle est définit respectivement à l'aide d'une réduction booléenne ou d'une expression quelconque ;
- le traitement de l'assertion de l'observateur. Pour cela, on définit les actions traiterAssertion_Red et traiterAssertion ;
- le « branchement » des différentes équations générées par la partie précédente, c'est-à-dire la génération des équations réalisant les connections logiques entre les différentes parties de la propriété générée.

Pour le traitement des équations définissant la sortie de l'observateur, notre algorithme différencie les deux cas suivants :

- la sortie de l'observateur est définie par une réduction booléenne ;
- la sortie de l'observateur est définie par une expression quelconque.

Cette différenciation est nécessaire car les traitements sont différents pour une propriété itérative ou pour une propriété quelconque :

- Dans le cas d'une propriété directement itérative, on doit générer des appels au nœud itéré ;
- Dans le cas d'une propriété quelconque, on doit générer des duplicata de l'expression utilisé.

D'autre part, le lecteur remarquera qu'on utilise l'action traiterEqPrincipal_Red pour la réduction principale, mais l'action traiteRed pour une itération utilisée ailleurs dans l'observateur à traiter. On différencie ces deux cas pour la raison suivante. Lors du traitement de l'équation principale, on doit générer les équations définissant propRankN et propRankNp1. Or ces variables sont un peu particulières : elle doivent notamment être utilisées dans la génération des branchements. Ce sont donc des variables globales de l'algorithme. Et ce sont les actions traiterEqPrincipal_Red et traiterEqPrincipal qui les définissent. Les variables définies par traiterRed n'ont pas à être connues globalement

(elle ne servent pas pour le branchement des propriétés construites). Elles sont générées dans le nœud, et on en perd la trace dans le reste de l'algorithme.

On aurait pu n'utiliser que traiterRed, mais les traitements effectués dans les deux actions traiterRed et traiterEqPrincipal_Red nous ont semblé assez différents pour justifier leur séparation dans des actions clairement identifiées.

Pour les même raisons, nous avons différencié les cas où

- l'assertion est définie par une réduction booléenne ;
- l'assertion est définie par une expression quelconque.

Nous avons aussi isolé le traitement de l'assertion pour des raisons évidentes : elle ne peut pas être prise en compte par traiterEq puisqu'il ne s'agit pas d'une équation.

```
Variable propInit;
Variable propRankN, propRankp1;
Variable assertRankN, assertRankp1;
traiterObservateur(Obs : AA_Observateur, nObs : AA_Observateur){
   if(estCalculéeparRed(Obs.Sorties[0]){
     traiterEqPrincipale_Red(equationDefining(Obs.Sorties[0]), nObs);
   }else{
     traiterEqPrincipale(equationDefining(Obs.Sorties[0]), nObs);
   }
   if(estExpRed(Obs.assertExpr)){
     traiterAssertion_Red(Obs.assertExpr, nObs);
   }else{
     traiterAssertion(Obs.assertExpr, nObs);
   }
   genererEquationsDeBranchements(nObs);
}
```

L'équation principale est un red – Le cas où l'équation principale de l'observateur est définie à l'aide d'un opérateur red est celui de l'exemple de la figure 8.29. Dans ce cas, on doit directement générer :

- une équation vérifiant la validité de la propriété itérée après un passage dans l'itération propInit = ... Cette variable est définie par un appel du nœud itéré. Pour chaque paramètre de l'itération, on génère un paramètre à l'appel de nœud. Dans le cas de l'accumulateur (premier paramètre de l'appel de nœud) on génère une copie de l'expression d'initialisation de l'itération. Dans les autres cas, on ajoute des nouvelles entrées à l'observateur nObs, correspondant à un élément des tableaux utilisés dans les paramètres de l'itération. On duplique chaque expression paramètre grâce à l'action dupliquerInit ;
- une entrée propRankN indiquant que la propriété est supposée vrai à un rang quelconque N ;
- une équation définissant la validité de la propriété au rang N+1 représentée par une variable locale propRankNp1. Comme pour propInit, on génère un appel du nœud itéré avec les paramètres adéquats. Pour l'accumulateur d'entrée, on utilise la variable propRankN générée à l'étape précédente. Ensuite, on utilise des variables représentant les éléments des tableaux sur lesquels l'itération est appliquée. On utilise pour cela l'action dupliquerRankNp1.

L'implication propRankN => propRankNp1 dépend aussi de l'hypothèse. Elle sera générée par l'action genererEquationsDeBranchements. Après avoir traité l'équation courante, il faut finalement penser à propager la transformation aux équations définissant des variables utilisées. Cette propagation en appelant sur chaque équation impliquée l'action traiterEquation.

L'ensemble de ces traitements est réalisé par l'action traiterEqPrincipale_Red de la figure B.1.

```
traiterEqPrincipale_Red(Eq : Equation, nObs : AA_Observateur){
   AA_Nœud nœudItere = Eq.expr.nœud;

   // Création de l'équation définissant propInit
   ajouterLocale(nObs, propInit);

   AA_AppelDeNœudExpr exprInit = creerAppelDeNœud(nœudItere);
   pourchaque paramètres p de l'itération Eq.expr faire{
         AA_Expr paramEff = dupliquerInit(p);
         ajouterParamètre(exprInit, paramEff);
   }
   AA_Equation eqInit = creerEquation(propInit, exprInit);
   ajouterEquation(nObs, eqInit);

   // Création de la variable d'entrée propRankN
   ajouterEntree(nObs, propRankN);

  // Création de l'équation définissant propRankNp1
   ajouterLocale(nObs, propRankNp1);

   AA_AppelDeNœudExpr exprNp1 = creerAppelDeNœud(nœudItere);
   pour chaque paramètres p de l'itération Eq.expr faire{
         AA_Expr paramEff = dupliquerRankNp1(p);
         ajouterParamètre(exprInv, paramEff);
   }
   AA_Equation eqNp1 = creerEquation(propRankNp1,exprNp1);
   ajouterEquation(nObs, eqNp1);

  // propagation aux équations définissant des variables utilisées
   pour chaque variable v utilisée dans Eq{
         traiterEquation(equationDefinissant(v),nObs);
   }
}
```

FIG. B.1 – Traitement de l'équation principale (cas red).

L'équation principale n'est pas un red – Dans le cas où l'équation n'est pas un red, on doit générer les équations définissant les 3 variables :

- propInit qui décrit toujours la validité de la propriété après un passage dans l'itération. Cette fois-ci, l'expression principal n'est pas une itération. On doit donc la dupliquer intégralement en remplaçant chaque occurrence de variable par une instance de la variable init correspondante. Cette étape est réalisée par l'action dupliquerInit ;
- propRankN qui décrit la validité de la propriété à un étage N de l'itération. On duplique l'expression courante de la même façon que pour le cas init, cette fois-ci en utilisant l'action dupliquerRankN ;

- propRankNp1 qui décrit la validité de la propriété à l'étage N+1 de l'itération. On duplique l'expression courante de la même façon que pour les cas init et N, cette fois-ci en utilisant l'action dupliquerRankNp1.

L'ensemble de ces traitements est réalisé par l'action traiterEqPrincipale_Quelconque donnée à la figure B.2.

```
traiterEqPrincipale_Quelconque(Eq : Equation, nObs : AA_Observateur){
   AA_Expr expression = Eq.expr;

   // Création de l'équation définissant propInit
   Variable propInit;
   ajouterLocale(nObs, propInit);
   AA_Expr exprInit = dupliquerInit(expression);
   Equation eqInit = creerEquation(propInit, exprInit);
   ajouterEquation(nObs, eqInit);

   // Création de l'équation définissant propRankN
   ajouterLocale(nObs, propRankN);
   AA_Expr exprRankN = dupliquerRankN(expression);
   Equation eqRankN = creerEquation(propRankN, exprRankN);
   ajouterEquation(nObs, eqRankN);

   // Création de l'équation définissant propRankNp1
   ajouterLocale(nObs, propRankNp1);
   AA_Expr exprRankNp1 = dupliquerRankNp1(expression);
   Equation eqRankNp1 = creerEquation(propRankNp1, exprRankNp1);
   ajouterEquation(nObs, eqRankNp1);

   // propagation aux équations définissant des variables utilisées
   pour chaque variable v utilisée dans Eq{
         traiterEquation(equationDefinissant(v),nObs);
   }
}
```

FIG. B.2 – Traitement de l'équation (non itérative) principale.

Traitement de l'assertion – Pour l'assertion, on procède exactement de la même façon. On va générer les nouvelles variables suivantes, symétriquement par rapport au traitement de l'itération :

- assertInit représente la validité de l'assertion après un passage dans les itérations utilisées ;
- assertRankN représente l'hypothèse « l'assertion est vraie au rang N de l'itération ;
- assertRankNp1 représente la validité de l'assertion au rang suivant, N+1.

Le reste du traitement est strictement identique. On distingue également le cas où l'assertion est elle-même une itération du cas où il s'agit d'une expression quelconque dépendant d'itérations. Comme pour l'équation principale, on n'oubliera pas de propager le traitement aux équations définissant les variables locales utilisées dans l'assertion. Les deux actions traiterAssertion_Red et traiterAssertion sont données aux figure B.3 et B.4.

Les actions suivantes permettent de traiter les équations définissant les variables locales de l'observateur à traiter. L'action traiterEquation décrite à la figure B.5 décide quelle action doit être accom-

plie, selon le type de l'équation rencontrée. Les actions traiterRed, traiterMapRed, traiterMap et traiterFill prennent en charge la propagation à chaque itération rencontrée de la transformation en cas init et inv.

Traitement d'une équation – Dans le cas d'une équation non-itérative, on se contente de dupliquer l'expression rencontrée. On utilise pour cela l'action dupliquer. Cette action réalisera les duplicata nécessaires. Deux cas doivent être distingués par l'action dupliquer :

- Le premier est celui où la variable définie est utilisée pour calculer une propriété non « directement » itérative. Dans ce cas, il faudra dupliquer l'équation en trois exemplaires : un pour le cas init, un pour le cas rankN, un dernier pour le cas rankNp1. Ce traitement suit exactement la méthode mise en place dans traiterEqPrincipale ;
- Le second est celui où la variable définie est utilisée pour calculer l'initialisation d'une réduction. Dans ce cas, on doit simplement dupliquer l'équation rencontrée.

Dans le cas d'une itération, on suit le même algorithme que pour le traitement de l'équation principale. Cette fois, on ne génère pas des équations définissant les variables propRankN et propRankNp1, mais des variables locales représentant les valeurs calculées par le nœud :

- après le premier passage dans l'itération ;
- après un passage de rang quelconque.

Dans l'exemple du paragraphe 8.5.4.1, on a notamment généré les équations :

```
res3_init,elt_T3_init = R(1, elt_tab_in_init);
res3_inv,elt_T3_inv = R(accIn_R, elt_tab_in_inv);
```

pour l'équation :

```
res3, T3 = map_red≪R;size≫(1, tab_in);
```

On donne à la figure B.6 l'action réalisant le traitement d'une équation map_red. Les actions correspondant au map, red et fill sont similaires. On commence par générer un appel *initial* du nœud itéré. Les paramètres de cet appel sont construits en appliquant l'action dupliquerInit aux paramètres de l'itération, et l'appel permet de calculer :

- l'accumulateur calculée par l'appel initial du nœud itéré : accRes_Init ;
- les variables init correspondant aux tableaux calculés par l'itération.

On ajoute ensuite l'entrée accRankN correspondant à l'accumulateur calculé au niveau N. Puis on calcule la variable accRankNp1 en appliquant le nœud itéré à :

- accRankN ;
- les variables RankN correspondant aux tableaux paramètres de l'itération.

Cette itération calcule aussi les éléments de niveau N+1 des tableaux définis par l'itération.

```
traiterAssertion_Red(Assertion : Expression, nObs : AA_Observateur){
   AA_Nœud nœudItere = Assertion.nœud;

  // Création de l'équation définissant assertInit
   AA_Variable assertInit;
   ajouterLocale(nObs, assertInit);

   AA_AppelDeNœudExpr exprInit = creerAppelDeNœud(nœudItere);
   pourchaque paramètres p de l'itération Eq.expr faire{
         AA_Expr paramEff = dupliquerInit(p);
         ajouterParamètre(exprInit, paramEff);
   }
   AA_Equation eqInit = creerEquation(assertInit, exprInit);
   ajouterEquation(nObs, eqInit);

   // Création de la variable d'entrée propRankN
   ajouterEntree(nObs, assertRankN);

   // Création de l'équation définissant propRankNp1
   ajouterLocale(nObs, assertRankNp1);

   AA_AppelDeNœudExpr exprNp1 = creerAppelDeNœud(nœudItere);
   pour chaque paramètres p de l'itération Eq.expr faire{
         AA_Expr paramEff = dupliquerRankNp1(p);
         ajouterParamètre(exprInv, paramEff);
   }
   AA_Equation eqNp1 = creerEquation(assertRankNp1,exprNp1);
   ajouterEquation(nObs, eqNp1);

   // propagation aux équations définissant des variables utilisées
   pour chaque variable v utilisée dans Eq{
   traiterEquation(equationDefinissant(v),nObs);
   }
}
```

FIG. B.3 – Traitement de l'assertion (cas red).

```
traiterAssertion(Assertion : Expression, nObs : AA_Observateur){

   // Création de l'équation définissant propInit
   Variable propInit;
   ajouterLocale(nObs, assertInit);
   AA_Expr exprInit = dupliquerInit(Assertion);
   Equation eqInit = creerEquation(assertInit, exprInit);
   ajouterEquation(nObs, eqInit);

   // Création de l'équation définissant propRankN
   ajouterLocale(nObs, assertRankN);
   AA_Expr exprRankN = dupliquerRankN(Assertion);
   Equation eqRankN = creerEquation(assertRankN, exprRankN);
   ajouterEquation(nObs, eqRankN);

   // Création de l'équation définissant propRankNp1
   ajouterLocale(nObs, propRankNp1);
   AA_Expr exprRankNp1 = dupliquerRankNp1(Assertion);
   Equation eqRankNp1 = creerEquation(propRankNp1, exprRankNp1);
   ajouterEquation(nObs, eqRankNp1);

   // propagation aux équations définissant des variables utilisées
   pour chaque variable v utilisée dans Eq{
         traiterEquation(equationDefinissant(v),nObs);
   }
}
```

FIG. B.4 – Traitement de l'assertion (non itérative).

```
traiterEquation(eq : Equation; nObs : AA_Observateur) {
   if(not(eq.isTreated)){
    markAsTreated(eq);
    switch(eq.expression.type){
            case(1 or 2 or 3 or 4) :
                  dupliquer(eq, nObs);
            case 5:
                  if(eq.expression.type=red){
                   traiterRed(eq,nObs);
                  }else if(eq.expression.type=map_red){
                   traiterMapRed(eq,nObs);
                  }else if(eq.expression.type=map){
                   traiterMap(eq,nObs);
                  }else if(eq.expression.type=fill){
                   traiterFill(eq,nObs);
                  }
   }
   pourchaque variable v utilisée dans Eq{
        traiterEquation(equationDefinissant(v),nObs);
   }
}
```

FIG. B.5 – Traitement d'une équation.

```
traiterMapRed(Eq : Equation, nObs : AA_Observateur){
   AA_Nœud nœudItere = Eq.expr.nœud;

   // Création de l'appel initial
   AA_Variable accRes_Init;
   ajouterLocale(nObs, accRes_Init);
   AA_AppelDeNœudExpr exprInit = creerAppelDeNœud(nœudItere);
   pour chaque paramètres p de l'itération Eq.expr faire{
         AA_Expr paramEff = dupliquerInit(p);
         ajouterParamètre(exprInit, paramEff);
   }
   AA_Equation eqInit = creerEquation(accRes_Init, exprInit);
   // On rajoute ensuite les parties gauches « tableaux » manquantes
   pour chaque partie gauche 'tableau' p de l'itération Eq faire {
         AA_Expr paramEff = dupliquerInit(p);
         ajouterPartieGauche(EqN,paramEff);
   }
   ajouterEquation(nObs, eqInit);

   // Création de la variable d'entrée accRankN
   ajouterEntree(nObs, accRankN);

   // Création de l'équation définissant accRankNp1
   ajouterLocale(nObs, accRankNp1);
   AA_AppelDeNœudExpr exprNp1 = creerAppelDeNœud(nœudItere);
   // Le premier paramètre est la variable accRank
   ajougerParamètre(exprInv, accRankN)
   // Ensuite, on duplique les paramètes « tableaux »
   pour chaque paramètres 'tableau' p de l'itération Eq.expr faire{
         AA_Expr paramEff = dupliquerRankNp1(p);
         ajouterParamètre(exprInv, paramEff);
   }
   AA_Equation eqNp1 = creerEquation(accRankNp1,exprNp1);
   // On rajoute ensuite les parties gauches « tableaux » manquantes
   pour chaque partie gauche 'tableau' p de l'itération Eq faire {
         AA_Expr paramEff = dupliquerRankNp1(p);
         ajouterPartieGauche(EqNp1,paramEff);
   }
   ajouterEquation(nObs, eqNp1);

   // propagation aux équations définissant des variables utilisées
   pour chaque variable v utilisée dans Eq{
         traiterEquation(equationDefinissant(v),nObs);
   }
}
```

FIG. B.6 – Traitement d'une itération map_red.

Bibliographie

[AdAG+01] R. Alur, L. de Alfaro, R. Grosu, T.A. Henzinger, M. Kang, C.M. Kirsch, R. Majumdar, F. Mang, and B.Y. Wang. jMOCHA: A model checking tool that exploits design structure. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 835–836, Los Alamitos, California, May 2001. IEEE Computer Society.

[AH96] R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, July 1996.

[AHM+98] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Computer Aided Verification Conference*, pages 521–525, 1998.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoreritcal Compuuture Science*, 82(2):253–284, 1991.

[AL93] M. Abadi and L. Lamport. Composing specification. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a non-procedural language with iteration. *Communications of ACM*, 20(7):519–526, July 1977.

[Bac78] J. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 8:613–641, 1978.

[BBS95] D. Brand, R.A. Bergamaschi, and L. Sotk. Don't cares in synthesis: Theoretical pitfalls and practical solutions. Technical report, IBM - Research Division, 1995.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. LeGuernic, and R. de Simone. Synchronous languages, 12 years later. Proceedings of the IEEE, January 2003.

[BFMW04] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass — Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):15, January 2004.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[Bir88] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, pages 151–216. Springer-Verlag, 1988. NATO ASI Series, F55.

[BLK92] R. A. Bergamaschi, D. Lobo, and A. Kuehlmann. Control optimization in high-level synthesis using behavioral don't cares. In *Proceedings of the 29th ACM/IEEE Conference on Design automation*, pages 657–661. IEEE Computer Society Press, 1992.

[BM75] S. K. Basu and J. Misra. Proving loop programs. *IEEE Transactions on Software Engineering*, 1(1):76–86, March 1975.

[Bou89] F. Boussinot. A reactive extension of C. Technical Report RR-1027, Inria, Institut National de Recherche en Informatique et en Automatique, 1989.

[Bou01] F. Boussinot. Java fair threads. Technical Report RR-4139, Inria, Institut National de Recherche en Informatique et en Automatique, 2001.

[Bro98] M. Broy. A functional rephrasing of the assumption/commitment specification style. *Formal Methods in System Design: An International Journal*, 13(1):87–119, May 1998.

[BS97] F. Boussinot and J.-F. Susini. The sugarcubes tool box. Technical Report RR-3247, Inria, Institut National de Recherche en Informatique et en Automatique, 1997.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CEJS98] E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model checking. In *Proceedings of the 10th International Computer Aided Verification Conference*, pages 145–458, 1998.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):77–104, August 1996.

[CP96] Paul Caspi and Marc Pouzet. Synchronous kahn networks. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 226–238. ACM Press, 1996.

[CP00] Jean-Louis Colaco and Marc Pouzet. Prototypages. Rapport final du projet GENIE II, Verilog SA, Janvier 2000.

[dAH01a] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, SOFTWARE ENGINEERING NOTES, pages 109–120. ACM Press, September 2001.

[dAH01b] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. *Lecture Notes in Computer Science*, 2211:148–165, 2001.

[DC00] C. Dumas-Canovas. *Méthodes Déductives Pour la Preuve de Programmes* LUSTRE. PhD thesis, Université Joseph Fourier, 2000.

[DKR00] D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.

[DM93] M. Damiani and G. De Micheli. Don't care set specifications in combinational and synchronous logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(3):365–388, March 1993.

[DQ94] C. Dezan and P. Quinton. Verification of regular architectures using ALPHA: a case study. Technical report, INRIA, June 1994.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proceedings of the 5th International Computer Aided Verification Conference*, pages 463–478, 1993.

[FF00] R.B. Findler and M. Felleisen. Behavioral interface contracts for Java. Technical report, Department of Computer Science, Rice University, Houston, TX, August 2000.

[FF02] R.B. Findler and M. Felleisen. Contracts for higher-order functions. *ACM SIGPLAN Notices*, 37(9):48–59, Setpember 2002.

[FMF01] R.B. Findler, M.Latendresse, and M. Felleisen. Object-oriented programming languages need well-founded contracts. Technical report, Department of Computer Science, Rice University, Houston, TX, 2001.

[Fuc92] N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, September 1992.

[Gau03] F. Gaucher. *Etude du Débogage de Systèmes Réactifs et Application au Langage Synchrone* LUSTRE. PhD thesis, Institut National Polytechnique de Grenoble, November 2003.

[GB87] P. Le Guernic and A. Benveniste. The synchronous language SIGNAL. In M. R. Barbacci, editor, *Proceedings from the Second Workshop on Large-Grained Parallelism*, pages 56–57, Pittsburgh, PA, November 1987. Carnegie-Mellon University Software Engineering Institute.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison Wesley, 1995.

[GL94] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[GLP93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. Technical report, University of Glasgow, October 1993.

[Hal93a] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[Hal93b] N. Halbwachs. A tutorial of lustre, 1993.

[Hay89] C. B. Jones I. J. Hayes. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, October 2002. LNCS 2491, Springer Verlag.

[HBL+04] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding symmetry reduction to uppaal, 2004.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HJJJ85] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Towards reachability trees for high-level petri nets. *Lecture Notes in Computer Science: Advances in Petri Nets 1984*, 188:215–233, 1985. NewsletterInfo: 27.

[HLR93] 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.

[HMM+03] N. Halbwachs, F. Maraninchi, D. Merchat, C. Parent, and R.K. Shyamasundar. A case-study with the luste programming environment: a fault-tolerant data acquisition system. Technical report, Verimag Grenoble, France and Tata Institute of Fundamental Research Mubai, India, 2003.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.

[Hol00] L. Holenderski. Compositional verification of synchronous networks. In *FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS*. LNCS, Springer-Verlag, 2000.

[HP85] D. Harel and A. Pnueli. *On the Development of Reactive Systems*, pages 477–498. Springer-Verlag New York, Inc., 1985.

[HQR98] T. A. Henzinger, S. Q., and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification*, pages 440–451, 1998.

[ID93a] C. N. Ip and D. L. Dill. Better verification through symmetry. In David Agnew, Luc Claesen, and Raul Camposano, editors, *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–112, Amsterdam, The Netherlands, April 1993. North-Holland.

[ID93b] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In Edna Straub, editor, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, Cambridge, MA, October 1993. IEEE Computer Society Press.

[ID96] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murφ. *Lecture Notes in Computer Science*, 1102:147–158, 1996.

[Jea00] B. Jeannet. *Partitionnement Dynamique Dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones*. PhD thesis, Institut National Polytechnique de Grenoble, 2000.

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *IEEE International Conference on Computer Languages (ICCL), Toulouse, France)*, 1994.

[JLS00] H. Jensen, K. G. Larsen, and A. Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS*. LNCS, Springer-Verlag, 2000.

[JTM99] J-M. Jezequel, M. Train, and M. Mingins. *Design Patterns and Contracts*. Addison-Wesley, 1999.

[JV00] P. Johann and E. Visser. Warm fusion in Stratego: A case study in generation of program transformation systems. Technical Report 2000-43, Institute of Information and Computing Sciences, University of Utrecht, August 2000.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[KHB99] M. Karaorman, U. Hølzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. *Lecture Notes in Computer Science*, 1616:175–197, 1999.

[KL93] R. P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In *Proceedings of the 5th International Computer Aided Verification Conference*, pages 166–179, 1993.

[Kra98] R. Kramer. iContract—the Java Designs by Contract tool. In *Proceedings of Technology of Object-Oriented Languages and Systems, TOOLS 26*. IEEE CS Press, Los Alamitos, 1998.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*, chapter 4, 6.3 and 7.11. Holt, Rinehart and Winston, New York, 1976.

[LBR99] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[LL95] F. Laroussinie and K. G. Larsen. Compositional model checking of real time systems. *Lecture Notes in Computer Science*, 962:27–47, 1995.

[LS90] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *ALGRTHMICA: Algorithmica*, 6, 1990.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 314–323, La Jolla, California, June 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[MA04] K. Morin-Allory. *Vérification Formelle dans le Modèle Polyédrique*. PhD thesis, Université de Rennes 1, 2004.

[Mau89] C. Mauras. *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, Rennes, France, December 1989.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In *Proc. 9th International Computer Aided Verification Conference*, pages 24–35, 1997.

[McM99] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 219–234, 1999.

[McM00] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, May 2000.

[McM01] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. *Lecture Notes in Computer Science*, 2144:179–195, 2001.

[Mey92] B. Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, Octobre 1992.

[MG00] F. Maraninchi and F. Gaucher. Step-wise + algorithmic debugging for reactive programs: Ludic, a debugger for LUSTRE. In *AADEBUG'2000 – Fourth International Workshop on Automated Debugging*, Munich, August 2000.

[MHB98] G. S. Manku, R. Hojati, and R. Brayton. Structural symmetry and model checking. *Lecture Notes in Computer Science*, 1427:159–172, 1998.

[Mik02] J. Mikac. Un raffinement pour le langage LUSTRE. Master's thesis, Paris 6, 2002.

[MM04a] F. Maraninchi and L. Morel. Arrays and contracts for the specification and analysis of regular systems. In *Fourth International Conference on Application of Concurrency to System Design (ACSD)*, Hamilton, Ontario, Canada, June 2004.

[MM04b] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004.

[Mor02] L. Morel. Efficient compilation of array iterators for lustre. In Florence Maraninchi, Alain Girault, and ?ric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[MP] L. Mandel and M. Pouzet. Reactiveml, a reactive extension to ml. Submitted to publication.

[MR01] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):61–92, October 2001.

[MR03] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 1381(46):219–254, 2003.

[Muc00] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[Nik91] R. S. Nikhil. *ID Reference Manual*. MIT Laboratory for Computer Science, 90.1 edition, July 1991.

[OCL97] Rational Software Corp. et al. *Object Constraint Language Specification, version 1.1*, September 1997.

[OM92] J-P Sansonnet O. Michel, D. De Vito. 8_1/2 : Data-parallelism and data-flow. Technical report, LRI-CNRS, Université Paris-Sud, 1992.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In Krzystof R. Apt, editor, *Logics and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, October 1984.

[Rat92] C. Ratel. *Définition et Réalisation d'un Outil de Vérification Formelle de Programmes* LUSTRE *: le Système* LESAR. PhD thesis, Université Joseph Fourier - Grenoble I, 1992.

[Ray91] P. Raymond. *Compilation Efficace d'un Langage Déclaratif Synchrone : le Générateur de Code* LUSTRE-V3. PhD thesis, Institut National Polytechnique de Grenoble, 1991.

[RM01] R.Jhala and K.L. McMillan. Microarchitecture verification by compositional model checking. *Lecture Notes in Computer Science*, 2102:396–410, 2001.

[Roc92] F. Rocheteau. *Extension du Langage* LUSTRE *et Application à la Conception de Circuits: Le Langage* LUSTRE*-V4 et le Système* POLLUX. PhD thesis, Institut National Polytechnique de Grenoble, 1992.

[Rou04] Y. Roux. *Déscription et Simulation de Systèmes Réactifs Non-Déterministes*. PhD thesis, Institut National Polytechnique de Grenoble, 2004.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[Ser04] M. Serrano. *Bigloo, A "Practical Scheme Compiler". User Manual for Version 2.6d*, 2004.

[Sha98] N. Shankar. Lazy compositional verification. *Lecture Notes in Computer Science*, 1536:541–564, 1998.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science : proceedings of the 5th conference*, volume 206 of *Lecture notes in computer science*, pages 369–391, New Dehli, December 1985. Springer-Verlag.

[Stø96] K. Stølen. Assumption/commitment rules for dataflow networks—with an emphasis on completeness. In *6th European Symposium on Programming—ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 356–372. Springer, April 1996.

[SYS] MAN MACHINE SYSTEMS. Jmsassert.

[Var02] M. Vareille. Extension du concept de programmation par contrats aux système synchrones réactifse à travers le langage synchrone lustre. Master's thesis, UFRIMA-UJF Grenoble, 2002.

[WA85a] W. Wadge and E. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press Inc., 1985.

[WA85b] W. W. Wadge and E. A. Ashcroft. *Lucid, the Data Flow Programming Language*. Academic Press, London, 1985.

[Wad84] P. L. Wadler. Listlessness is better than laziness. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52. ACM, ACM, August 1984.

[Wad85] P. L. Wadler. Listlessness is better than laziness II: Composing listless functions. In *Lecture Notes in Computer Science 217*. Springer-Verlag, New York, NY, 1985.

[Wad90] P. L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 90.

[Wat91] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.

[WB93] H.-Y. Wang and R.K. Brayton. Input Don't Care Sequences in FSM Networks. In *IEEE /ACM International Conference on CAD*, pages 321–329, Santa Clara, California, November 1993. IEEE Computer Society Press.

[WB94] H.-Y. Wang and R.K. Brayton. Permissible observability relations in FSM networks. In Michael Lorenzetti, editor, *Proceedings of the 31st Conference on Design Automation*, pages 677–683, New York, NY, USA, June 1994. ACM Press.

[ZS01] M. Zulkernine and R.E. Seviora. Assume/Guarantee supervisor for concurrent systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, pages 151–155, Los Alamitos, CA, April 2001. IEEE Computer Society.

Résumé

Le travail décrit dans cette thèse s'inscrit dans le contexte du développement et de la validation des systèmes réactifs synchrones. Il vise à tirer parti de certaines formes de structuration des programmes durant le processus de développement et de validation. Nous étudions premièrement l'utilisation d'opérateurs réguliers de types "itérateurs" qui permettent d'exprimer assez facilement des programmes réguliers manipulant des tableaux. Nous montrons aussi comment, au moment de la validation, on peut tirer partie de ces structures régulières pour rendre la preuve d'une propriété plus simple. Nous nous intéressons ensuite à la spécification dite "par contrat" où un couple *(assume, guarantee)* est associé à chaque composant pour spécifier les hypothèses sur l'environnement et les propriétés satisfaites par le composant sous ces hypothèses. Nous montrons l'intérêt de tels contrats à la fois en terme de spécification et de vérification pour le cas particulier des systèmes synchrones. Nous proposons une série d'algorithmes de transformations de programmes (aussi bien autour de l'utilisation des itérateurs que des contrats) utilisable comme pre-processeur d'objectifs de preuve pour les outils de validation. Nos propositions, notamment sur l'aspect langage des itérateurs, ont répondu à des besoins rencontrés dans les applications industrielles, particulièrement autour du langage Lustre, auquel nous appliquons nos résultat. Ces propositions seront bientôt incluses dans la version industrielle du langage.

Mots-clés : programmation synchrone, Lustre, contrats "assume-guarantee", itérateurs de tableaux, validation

Abstract

The work presented in this thesis takes place in the domain of programming and validation of synchronous reactive systems. It aims at taking certain forms of structure of programs into account during the development and validation process. We first study the use of regular operators called "iterators" which allow to express in a quit natural way regular programs that manipulate arrays. On a validation point of view, we show how to take these regular structures into account in order to facilitate the proof of properties. In parallel to those iterators, we investigate a form of specification called *assume-guarantee contracts* where a couple of properties are used to specify some hypothesis of a component upon its environment and some property that the component satisfies whenever the hypothesis on the environment is satisfied. We show the interest of such contracts in terms of specification and verification for the particular case of the synchronous systems. We finally propose a a series of program transformation algorithms, both for the iteration and the contract aspects that can be used as a pre-processor of proof objectives for validation tools. Our proposals, in particular on the language aspect of the iterators, met needs of several industrial applications, notably around the language Lustre, to which we apply our result. These propositions will soon be added to the industrial version of the language.

Keywords: Synchronous programming, "assume-guarantee" contrats, array iterators, validation