HAL
open science

Axiomatic Semantics of Data-Parallel Languages; Automatization of Programs Verification

V. Mounier, Gil Utard

To cite this version:

V. Mounier, Gil Utard. Axiomatic Semantics of Data-Parallel Languages; Automatization of Programs Verification. [Research Report] LIP RR-1993-08, Laboratoire de l'informatique du parallélisme. 1993, 2+48p. hal-02101782

HAL Id: hal-02101782

https://hal-lara.archives-ouvertes.fr/hal-02101782

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Institut IMAG
Unité de recherche associée au CNRS n°1398

Sémantique axiomatique des langages à parallélisme de données ; automatisation de la preuve de programmes

Laurent Mounier
Gil Utard

Mars 1993

Research Report N° 93-08

Ecole Normale Supérieure de Lyon
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

Sémantique axiomatique des langages à parallélisme de données ; automatisation de la preuve de programmes

Laurent Mounier
Gil Utard

Mars 1993

Abstract

We give a Hoare-like proof system for the data-parallel language $\mathcal{L}$, and we present an automatic tool to aid program correctness proof. After recalling $\mathcal{L}$'s *operational* semantics, we define an *axiomatic* semantics. We illustrate proof of $\mathcal{L}$ programs with two examples. Then we extend our semantics to weakest precondition, and we deduce techniques for mechanizing program verification from Gordon's *verification condition* method. We prove its correctness, and we present an implementation of this method in the CENTAUR system.

Keywords: data-parallel languages, axiomatic semantics, weakest precondition, program verification, program verification tools, CENTAUR

Résumé

Nous présentons un système de preuve à la Hoare pour le langage à parallélisme de donnée $\mathcal{L}$, ainsi qu'une automatisation possible de la preuve de programmes. Dans un premier temps nous définissons une sémantique *axiomatique* de $\mathcal{L}$, après un rappel de sa sémantique *opérationnelle*. Nous illustrons son application à la preuve de programmes sur deux exemples, dont un non trivial. Ensuite, après avoir étendu notre sémantique aux pré-conditions les plus faibles, nous déduisons une mécanisation de la preuve de programmes inspirée des *verification conditions* de Gordon. Nous prouvons alors sa correction. Nous présentons enfin son implémentation sous l'atelier sémantique CENTAUR.

Mots-clés: langages à parallélisme de données, semantique axiomatique, pré-conditions les plus faibles, preuve de programmes, outils d'aide à la vérification de programme, CENTAUR

Sémantique axiomatique des langages à parallélisme de données ; automatisation de la preuve de programmes*

Laurent Mounier† et Gil Utard

LIP-IMAG, URA CNRS 1398, ENS Lyon
46 Allée d'Italie, F-69364 Lyon Cedex 07, France
Email : `Gil.Utard@lip.ens-lyon.fr`

Mars 1992

Table des matières

1 Le langage $\mathcal{L}$ **2**
1.1 Description informelle du langage . . . 3
1.2 Sémantique opérationnelle . . . 3
1.3 Restrictions sur les programmes considérés . . . 6

2 Sémantique axiomatique **7**
2.1 Langage d'assertion . . . 8
2.2 Substitutions . . . 13
2.3 Quelques équivalences utiles . . . 14
2.4 Le système de preuve . . . 16

3 La sémantique axiomatique comme outil de programmation **18**
3.1 Une preuve complète de programme . . . 18
3.2 Conception d'un programme dirigée par sa preuve . . . 22
3.2.1 Approche abstraite . . . 22
3.2.2 Implémentation . . . 25
3.3 Discussion . . . 27

4 Automatiser la preuve de programmes $\mathcal{L}$ **29**
4.1 Une stratégie automatisable pour générer des preuves . . . 29
4.2 Calcul des plus faibles préconditions . . . 31
4.3 Calcul des *verification conditions* . . . 35

5 Un environnement d'aide à la preuve automatique **39**
5.1 Architecture générale . . . 40
5.2 Mise en œuvre sous CENTAUR . . . 42
5.3 Mise en œuvre sous HOL . . . 43
5.4 Discussion . . . 45

6 Conclusions **45**

*Ce travail a été soutenu par le Programme de Recherches Coordonnées du CNRS C^3 et le contrat DRET 91/1180.

†Nouvelle adresse : LGI-IMAG, BP 53 X, 38041 Grenoble Cedex France. Email : `mounier@imag.fr`.

Introduction

Dans le domaine du calcul intensif, le grand défi de cette fin de siècle est d'atteindre et d'*exploiter* des puissances de calculs de l'ordre du *Teraflops* (10^{12} opérations flottantes par seconde). Les constructeurs de calculateurs nous promettent de fournir de telles capacités par le biais du parallélisme massif. Mais de la diversité et de la complexité des machines qui nous sont, et seront, proposées naît un autre problème : comment exploiter rapidement et à moindre coût toutes ces nouvelles architectures ? La solution consiste à fournir un modèle de programmation unique à toutes ces machines, à l'aide de langages de programmation adéquats.

Le parallélisme de données semble être un bon candidat, il rallie la plupart des suffrages, comme le prouve la définition de la norme HPF [18]. Bougé [7] souligne que ce modèle n'est pas une vision restreinte des architectures parallèles, mais doit être considéré comme une discipline de programmation, comme pour l'introduction de la structure de contrôle `while` dans les langages séquentiels.

Jusqu'à présent, les langages à parallélisme de données ont été peu étudiés d'un point de vue formel. Notre travail tente de combler ce vide à partir de la définition d'un langage représentatif minimal : le langage $\mathcal{L}$ [6]. La définition d'une sémantique *opérationnelle* précise a permis d'étudier [4] :

- l'*expressivité* de $\mathcal{L}$ par rapport aux langages réels tels que C^*, MPL ou POMPC, c'est-à-dire que toutes structures de contrôle de ces derniers sont exprimables dans notre langage cible ;
- la validation des processus de compilation de ces dernières.

Nous allons maintenant étudier une autre sémantique de $\mathcal{L}$ visant la preuve de programmes : la sémantique *axiomatique* [23], dite encore «logique de Hoare». Plus orientée vers l'utilisateur de part sa finalité, nous allons nous intéresser plus particulièrement à son emploi comme outil de génie logiciel pour le développement et la preuve de programmes.

Dans un premier temps nous rappellerons la définition du langage $\mathcal{L}$, ainsi que sa sémantique axiomatique associée. Nous illustrerons sur deux exemples sa mise en œuvre pour prouver, et surtout pour aider la dérivation de programmes. Après avoir introduit la notion de *pré-condition la plus faible*, nous définirons une méthode pour automatiser partiellement la preuve de programmes. Nous terminerons en présentant une implantation de cette méthode dans l'atelier logiciel CENTAUR.

1 Le langage $\mathcal{L}$

Le langage $\mathcal{L}$ a été proposé par Bougé [6] pour offrir une plate-forme d'étude des langages data-parallèles. L'objectif est de disposer d'un langage qui soit à la fois suffisamment simple pour pouvoir en décrire les aspects sémantiques de façon exhaustive, mais également suffisamment expressif pour que les résultats obtenus puissent être étendus aux structures de contrôle présentes dans les langages data-parallèles réels, comme C^*, MPL ou encore POMPC. On ne décrit pas ici le modèle d'exécution sous-jacent qui est associé à $\mathcal{L}$, et qui est directement inspiré des architectures de types SIMD : une unité de contrôle centralisée diffuse de manière synchrone les instructions du programme à un ensemble de processeurs qui les exécutent ou non selon leur activité (le *contexte*), chaque processeur traitant des données propres. Une description plus détaillée de ce modèle peut être trouvée dans [6]

On donne la définition du langage $\mathcal{L}$ qui a été adoptée dans la suite en présentant tout d'abord la syntaxe de ses instructions, ainsi que leur signification informelle, puis en en donnant une sémantique opérationnelle précise. On termine alors cette section en proposant une «forme normale» qui a été choisie pour représenter les programmes que l'on considère du point de vue de la vérification.

1.1 Description informelle du langage

Pour des raisons de simplicité, on supposera que les variables des programmes appartiennent à un seul type de données, les *vecteurs d'entiers de dimension 1*, et on les notera à l'aide d'identificateurs écrits en majuscule. Par suite, la valeur locale au processeur u d'une variable parallèle X, qui représente la u^{e} composante de ce vecteur, sera notée $\sigma(X)|_u$. Enfin, des *expressions parallèles* peuvent être construites sur ces variables à l'aide des opérateurs arithmétiques et logiques usuels. Ces expressions sont *locales*, c.-à-d. qu'il n'est pas possible de faire référence à des composantes différentes dans une expression. En reprenant le modèle de machine abstrait SIMD précédemment cité, cela signifie que l'évaluation d'une expression pour un processeur ne fait aucune référence aux valeurs détenues par un autre processeur. Le seul moyen d'accéder aux valeurs des autres processeurs est d'effectuer une instruction de communication *explicite* auparavant.

On donne les différentes instructions du langage $\mathcal{L}$ que l'on va considérer dans la suite, en précisant leur signification informelle :

Affectation : X `:=` E

L'expression E est évaluée *localement* par chacun des processeurs actifs, sa valeur à l'adresse u ne dépend que des valeurs à l'adresse u des variables qui la composent, la valeur obtenue est alors affectée à leur composante de la variable parallèle X.

Communication globale : `get` Y `from` A `into` X

Chaque processeur actif u affecte à sa composante locale du vecteur X la valeur de la composante de Y située sur le processeur d'adresse $A\,|_u$, ou A est une expression donnée. On suppose que les composantes de A qui correspondent à des processeurs actifs contiennent toutes des adresses correctes.

Composition séquentielle : S ; T

Les instructions S et T sont exécutées en séquence, de manière synchrone, par l'ensemble des processeurs.

Itération : `while` B `do` S `end`

L'ensemble des processeurs actifs exécutent le programme S tant que l'un d'entre eux au moins évalue l'expression booléenne vectorielle B à vrai. L'instruction se termine donc quand tous les processeurs actifs évaluent B à faux.

Conditionnement : `where` B `do` S `end`

L'ensemble des processeurs actifs qui évaluent l'expression booléenne vectorielle B à faux sont rendus inactifs pendant toute l'exécution du programme S. L'activité des autres processeurs est inchangée.

1.2 Sémantique opérationnelle

Nous présentons brièvement une sémantique opérationnelle «à la Plotkin» pour le langage $\mathcal{L}$, qui est inspirée de [4].

L'objectif d'une telle sémantique est de décrire formellement, à partir de la syntaxe d'un programme, les transitions effectuées par une machine abstraite qui exécuterait ce programme. Confor-

mément au modèle d'exécution sous-jacent que l'on considère pour $\mathcal{L}$, les états de cette machine abstraite sont des triplets de la forme $\langle S, \sigma, s\rangle$.

- S représente la partie du programme restant à exécuter (la *continuation*). La continuation vide est notée $\bullet$.

- σ représente la fonction *environnement*, qui associe une valeur à chacune des variables du programme :
$$\sigma : VAR \rightarrow VAL$$
où VAR et VAL désignent respectivement les domaines des noms et des valeurs des variables des programmes $\mathcal{L}$. Notons que, les variables parallèles étant des vecteurs d'entiers de dimension 1, l'ensemble choisi pour VAL est l'ensemble des fonctions des entiers dans les entiers.

  On désignera par env l'ensemble des environnements.

- s représente l'activité de chacun des processeurs (ou le *contexte*), qui est modifié lors de l'exécution d'une instruction *where*. Du fait de l'imbrication possible de ces instructions, s est représenté par une pile de vecteurs booléens, telle que $Top(s)|_u = vrai$ si et seulement si le processeur u est actif dans l'état considéré.

  On désignera par $ctxte$ l'ensemble des contextes.

Dans la suite, on utilisera également les notations suivantes.

- La pile de contexte vide est notée ε. Elle représente l'activité des processeurs au début du programme, et l'on suppose $Top(\varepsilon) = vrai$.

- Pour tout processeur u, le prédicat $actif\,(u)$ représente l'*activité* du processeur u :
$$actif\,(u) \equiv (Top(s)|_u = vrai)$$

- La fonction σ est implicitement étendue aux *expressions parallèles* : pour toute expression parallèle E, qui contient ou non des variables libres, on note $\sigma(E)$ la valeur (parallèle) de E, et $\sigma(E)|_u$ la valeur de E locale au processeur u.

- Parmi les expressions parallèles, on distingue la constante parallèle $This$ où pour chaque composante u, on a $\sigma(This)|_u = u$.

- Pour tout environnement σ, on note $\sigma[X \leftarrow V]$, l'environnement obtenu à partir de σ en remplaçant la valeur $\sigma(X)$ par la nouvelle valeur V :
$$\begin{array}{rcl} \forall Y\ .\ X \neq Y \ \Rightarrow\ \sigma[X \leftarrow V](Y) & = & \sigma(Y) \\ \sigma[X \leftarrow V](X) & = & V \end{array}$$

- Pour tout ensemble d'états $\mathcal{E}C \subseteq env \times ctxte$, on note pr_e (respectivement pr_c) la projection de $\mathcal{E}C$ dans env (respectivement $ctxte$).

Il reste alors à donner pour chaque instruction $\mathcal{L}$, la transition correspondante effectuée par la machine abstraite :

Affectation :

$$\langle X := E,\ \sigma,\ s\rangle \longrightarrow \langle \bullet,\ \sigma',\ s\rangle$$

avec :

- $\sigma'(X)|_u = \sigma(E)|_u$ si $actif(u)$
- $\sigma'(X)|_u = \sigma(X)|_u$ si $\neg actif(u)$
- $\sigma'(Y)|_u = \sigma(Y)|_u$ si $Y \neq X$

Communication générale :

$$\langle \texttt{get}\ Y\ \texttt{from}\ A\ \texttt{into}\ X, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma', s\rangle$$

avec :

- $\sigma'(X)|_u = \sigma(Y)|_{\sigma(A)|_u}$ si $actif(u)$;
- $\sigma'(X)|_u = \sigma(X)|_u$ si $\neg actif(u)$;
- $\sigma'(Y)|_u = \sigma(Y)|_u$ si $Y \neq X$.

Composition séquentielle :

$$\frac{\langle S, \sigma, s\rangle \longrightarrow \langle S', \sigma', s'\rangle}{\langle S;T, \sigma, s\rangle \longrightarrow \langle S';T, \sigma', s'\rangle}$$

$$\frac{\langle S, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma', s'\rangle}{\langle S;T, \sigma, s\rangle \longrightarrow \langle T, \sigma', s'\rangle}$$

Itération :

$$\frac{\forall u\ .\ (actif(u)\ \Rightarrow\ \neg\,(\sigma(B)|_u))}{\langle \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma, s\rangle}$$

$$\frac{\exists u\ .\ (actif(u)\ \wedge\ \sigma(B)|_u)}{\langle \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}, \sigma, s\rangle \longrightarrow \langle S\ ;\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}, \sigma, s\rangle}$$

Ces deux règles indiquent bien que tant qu'il existe un processeur *actif* qui vérifie la condition de boucle, le corps de celle-ci est exécuté. La pile de contexte n'est, quant à elle, pas modifiée.

Conditionnement :

La nouvelle activité, qui correspond au vecteur booléen $Top(s) \wedge \sigma(B)$, doit être empilée sur la pile des contextes s. Pour conserver une trace des blocs conditionnels, on utilise également la nouvelle construction syntaxique `begin` S `end` :

$$\langle \texttt{where}\ B\ \texttt{do}\ S\ \texttt{end}, \sigma, s\rangle \longrightarrow \langle \texttt{begin}\ S\ \texttt{end}, \sigma, s'\rangle$$

avec

$$s' = Push\,(Top(s) \wedge \sigma(B), s).$$

Les règles suivantes indiquent alors que le programme P est exécuté jusqu'à sa terminaison éventuelle, et que l'ancien contexte est ensuite restauré à la sortie du bloc en dépilant la pile s :

$$\frac{\langle S, \sigma, s\rangle \longrightarrow \langle S', \sigma', s'\rangle}{\langle \texttt{begin}\ S\ \texttt{end}, \sigma, s\rangle \longrightarrow \langle \texttt{begin}\ S'\ \texttt{end}, \sigma', s'\rangle}$$

$$\langle \texttt{begin}\ \bullet\ \texttt{end}, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma, Pop(s)\rangle$$

Pour compléter cette description formelle du langage, on définit également la fonction $[\![S]\!]$, qui, pour tout programme S et pour tout environnement et contexte initial (σ, s), retourne le *résultat* de l'exécution de S, lorsqu'il existe (c.-à-d. lorsque S termine).

Définition 1-1

Pour tout programme S, et, pour tout environnement et contexte (σ, s) :

$$[\![S]\!](\sigma, s) = \{(\sigma', s') \mid \langle S, \sigma, s\rangle \longrightarrow^* \langle \bullet, \sigma', s'\rangle\}$$

■

Par abus de notations, nous étendons la fonction $[\![S]\!]$ aux ensembles d'états, c.-à-d. si :

$$\mathcal{E}C \subseteq env \times ctxte$$

on a :

$$[\![S]\!](\mathcal{E}C) = \{(\sigma', s') \mid \exists(\sigma, s) \in \mathcal{E}C.\langle S, \sigma, s\rangle \longrightarrow^* \langle \bullet, \sigma', s'\rangle\}$$

On a alors les résultats suivants, qui seront utilisés par la suite :

Proposition 1-1

Pour tout environnement $\sigma \in env$ et contexte $s \in ctxte$ on a:

$$\begin{array}{rcl}
[\![X := E]\!](\sigma, s) & = & \{(\sigma[X \leftarrow V], s)\} \\
& & \text{où } V \text{ est définie comme suit :} \\
& & \quad V|_u = \sigma(E)|_u \text{ si } actif(u) \\
& & \quad V|_u = \sigma(X)|_u \text{ si } \neg actif(u) \\
[\![\texttt{get } Y \texttt{ from } A \texttt{ into } X]\!](\sigma, s) & = & \{(\sigma[X \leftarrow V], s)\} \\
& & \text{où } V \text{ est définie comme suit :} \\
& & \quad V|_u = \sigma(Y)|_{\sigma(A)|_u} \text{ si } actif(u) \\
& & \quad V|_u = \sigma(X)|_u \text{ si } \neg actif(u) \\
[\![S_1; S_2]\!](\sigma, s) & = & [\![S_2]\!]([\![S_1]\!](\sigma, s)) \\
[\![\texttt{where } B \texttt{ do } S \texttt{ end}]\!](\sigma, s) & = & \{(\sigma', s) \mid \sigma' \in pr_e([\![S]\!](\sigma, Push(Top(s) \wedge \sigma(B), s)))\} \\
[\![\texttt{while } B \texttt{ do } S \texttt{ end}]\!](\sigma, s) & = & (\sigma, s) \\
& & \quad \text{si : } \forall u.actif(u) \Rightarrow \neg\sigma(B)|_u \\
[\![\texttt{while } B \texttt{ do } S \texttt{ end}]\!](\sigma, s) & = & [\![S; \texttt{while } B \texttt{ do } S \texttt{ end}]\!](\sigma, s) \\
& & \quad \text{si : } \exists u.actif(u) \wedge \sigma(B)|_u
\end{array}$$

■

Notons que, le langage $\mathcal{L}$ étant *déterministe*, le cardinal de ces ensembles est au plus de un.

1.3 Restrictions sur les programmes considérés

La méthode de preuve qui est proposée dans ce rapport a été appliquée à un sous-ensemble du langage $\mathcal{L}$ par rapport à celui présenté dans la section 1.1. On décrit ici les hypothèses qui sont effectuées sur les programmes que l'on considère dans la suite, puis on montre comment ces restrictions peuvent être levées.

On introduit tout d'abord les notations suivantes :

- Pour tout programme S, on désigne par $Change(S)$ l'ensemble des variables qui peuvent être *modifiées* lors de l'exécution de S. Il s'agit donc des variables qui apparaissent en partie gauche d'une affectation ou dans une instruction de communication.

– Pour toute expression E, on désigne par $Var(E)$ l'ensemble des variables qui *apparaissent* dans E.

On désigne alors par Pgm un sous-ensemble des programmes en $\mathcal{L}$ qui satisfont la définition suivante.

Définition 1-2
Un programme T de $\mathcal{L}$ est dit à *contexte fixe* si pour toute instruction «`where` B `do` S `end`» de T, on a :

$$Change(S) \cap Var(B) = \emptyset$$

■

En d'autres termes, cette contrainte signifie que les variables de T qui apparaissent dans une expression de conditionnement ne sont pas modifiées dans le corps du sous-programme S qui est conditionné. Notons qu'il est possible de déterminer *statiquement* si un programme $\mathcal{L}$ donné satisfait ou non cette contrainte.

Le motif de cette restriction est qu'elle permet de simplifier de façon notable le système de preuve qui décrit la sémantique axiomatique de de $\mathcal{L}$, en supprimant notamment la notion d'*équivalence de contexte* [23]. Par suite, les preuves mises en œuvre pour établir la correction d'un programme sont plus simples que dans le cas général, et leur automatisation s'en trouve également facilitée. En outre, il est possible de montrer qu'un programme P qui n'est pas à contexte fixe peut toujours être transformé en un programme P' qui la satisfait. L'idée intuitive de cette transformation est de remplacer dans P chaque instruction

`where` B `do` S `end`

telle que $Change(S) \cap Var(B) \neq \emptyset$, par la séquence d'instruction :

B' `:=` B ; `where` B' `do` S `end`

où B' est une nouvelle variable.

Cette transformation peut être justifiée formellement à l'aide d'une notion d'équivalence de programme [4] définie à partir de la sémantique opérationnelle.

Enfin, Le Guyadec et Virot [16] ont récemment proposé une modification du système de preuve dans laquelle cette transformation est codée de manière simple à l'aide d'une règle d'inférence. L'automatisation que nous décrivons dans la suite pourrait être adaptée à ce système de preuve, et elle deviendrait donc ainsi applicable à un programme $\mathcal{L}$ quelconque.

2 Sémantique axiomatique

Nous rappelons dans cette section la sémantique axiomatique proposée dans [23] pour le langage $\mathcal{L}$. L'objectif d'une telle sémantique est de fournir un système de preuve, similaire à la «Logique de Hoare» des langages séquentiels, qui permet d'inférer statiquement des *assertions* sur le comportement d'un programme au cours de son exécution, et à terme de prouver qu'un programme est correct. Plus précisément, la «Logique de Hoare» se présente de la façon suivante : soient un programme S et deux prédicats P et Q qui décrivent les valeurs d'entrée et de sortie des variables manipulées par S (autrement dit sa *spécification*). Le but est de prouver que, si celles-ci vérifient P avant l'exécution de S, et si S termine, alors elles vérifient Q après l'exécution. On note alors cette propriété $\{P\}\ S\ \{Q\}$, où P est appelé *précondition*, et Q *postcondition*.

Toutefois, dans le cas d'un programme data-parallèle, la difficulté principale vient du fait que le comportement du programme se déduit non seulement des valeurs successives prises par ses variables, mais également des changements d'activités des processeurs de la machine sous-jacente. Pour résoudre ce problème, la solution proposée dans [23] consiste à définir les assertions comme étant des couples constitués de deux composantes distinctes :

- une partie *état*, représentée par une proposition logique sur les valeurs des variables du programme ;
- une partie *contexte*, représentée par expression booléenne parallèle qui permet de déduire l'activité des processeurs.

Cette forme d'assertions en deux parties est aussi employée dans les travaux de Gabarró et Gavaldà [11], mais la partie contexte est alors un ensemble d'entiers dénotant explicitement les processeurs actifs.

Par suite, les propriétés définies par notre système de preuve seront de la forme :

$$\{P, C\}\ S\ \{Q, D\}$$

où S est un fragment de programme et $\{P, C\}$ et $\{Q, D\}$ des assertions (P et Q décrivant des états, C et D des contextes).

Nous présentons plus en détails les deux langages qui définissent ces assertions, puis nous donnons le système de preuve associé au sous-ensemble du langage $\mathcal{L}$ que nous considérons dans la suite.

2.1 Langage d'assertion

On note $Etat$ et $Contexte$ les langages dédiés respectivement aux parties *état* et *contexte* des assertions. On décrit successivement la syntaxe et la sémantique de ces deux langages.

Syntaxe

On définit tout d'abord quatre types d'expressions sur lesquels seront construits les langages $Etat$ et $Contexte$, et qui sont respectivement :

- les expressions arithmétiques et logiques scalaires, notées $Sarith$ et $Sbool$;
- les expressions arithmétiques et logiques vectorielles, notées $Varith$ et $Vbool$.

Concrètement, le langage $Etat$ sera défini comme un ensemble de formules quantifiées construites sur $Sbool$, et le langage $Contexte$ sera constitué des expressions $Vbool$.

Les expressions $Sarith$, $Sbool$, $Varith$ et $Vbool$ sont décrites par la grammaire suivante :

$$\begin{array}{lcl} Sarith & ::= & c \\ & & x \\ & & Varith|_{Sarith} \\ & & Sarith\ Binop\ Sarith \\ & & \text{if } Sbool \text{ then } Sarith \text{ else } Sarith \end{array}$$

$$
\begin{array}{rcl}
Sbool & ::= & \text{true} \\
 & & \text{false} \\
 & & Vbool|_{Sarith} \\
 & & \neg\, Sbool \\
 & & Sarith\ Rel\ Sarith \\
 & & Sbool\ Connect\ Sbool \\
\\
Varith & ::= & C \\
 & & X \\
 & & Varith|_{Varith} \\
 & & Varith\ Binop\ Varith \\
 & & \text{IF } Vbool \text{ THEN } Varith \text{ ELSE } Varith \\
\\
Vbool & ::= & \text{TRUE} \\
 & & \text{FALSE} \\
 & & \neg\, Vbool \\
 & & Varith\ Rel\ Varith \\
 & & Vbool\ Connect\ Vbool
\end{array}
$$

dans lesquelles :

- c (*resp.* C) dénote une constante entière scalaire (*resp.* parallèle dont $This$) ;
- x (*resp.* X) dénote une variable entière scalaire (*resp.* parallèle) ;
- $Varith|_{Sarith}$ (*resp.* $Vbool|_{Sarith}$) représente la composante d'indice $Sarith$ de l'expression parallèle $Varith$ (*resp.* $Vbool$) ;
- $Binop$ est un ensemble d'opérateurs arithmétiques :

$$Binop ::= + \mid - \mid * \mid div \mid mod \mid \ \ldots$$

- Rel est un ensemble d'opérateurs de comparaison :

$$Rel ::= = \mid \neq \mid < \mid > \mid \leq \mid \ \ldots$$

- $Connect$ est un ensemble de connecteurs logiques :

$$Connect ::= \ \wedge \ \mid \ \vee \ \mid \Rightarrow \mid \Leftrightarrow \mid \ \ldots$$

On définit alors un ensemble $Sform$ de formules quantifiées sur $Sbool$:

$$
\begin{array}{rcl}
Sform & ::= & Sbool \\
 & & Sform\ Connect\ Sform \\
 & & \neg\, Sform \\
 & & \forall x\ .\ Sform \\
 & & \exists x\ .\ Sform
\end{array}
$$

dans lequel x désigne une variable entière *scalaire*.

Finalement, la syntaxe des assertions est la suivante :

$$\begin{array}{rcl} Assertion & ::= & \{Etat, Contexte\} \\ Etat & ::= & Sform \\ Contexte & ::= & VBool \end{array}$$

Sémantique

Le domaine choisi pour interpréter les assertions et celui des entiers de l'arithmétique de Peano, ce qui correspond en fait à l'interprétation usuelle des symboles arithmétiques.

Pour décrire formellement la sémantique de notre langage d'assertions, il convient tout d'abord d'étendre aux variables scalaires la notion d'environnement définie dans la section 1.2. Plus précisément, on considère dans la suite qu'un environnement σ associe une valeur scalaire (c.-à-d. un entier) à chaque variable scalaire de type var, et une valeur parallèle (c.-à-d. une fonction des entiers dans les entiers) à chaque variable parallèle de type VAR.

On a donc :

$$\sigma : \begin{cases} var \rightarrow entier \\ VAR \rightarrow (entier \rightarrow entier) \end{cases}$$

On donne tout d'abord la sémantique des expressions, en introduisant une fonction $[\![E]\!](\sigma)$, qui, étant donné une expression E et un environnement σ, retourne la *valeur* de E dans σ. Le type de son résultat dépend donc du type de l'expression évaluée. Formellement :

$$[\![\,]\!] \;:\; \begin{bmatrix} & Sarith \\ \cup & Sbool \\ \cup & Varith \\ \cup & Vbool \end{bmatrix} \longrightarrow \left(env \longrightarrow \begin{bmatrix} & entier \\ \cup & \{vrai, faux\} \\ \cup & entier \longrightarrow entier \\ \cup & entier \longrightarrow \{vrai, faux\} \end{bmatrix} \right)$$

La fonction $[\![\,]\!]$ est définie récursivement sur la structure des expressions, en distinguant une interprétation entière (lorsque le résultat est un entier, ou une fonction des entiers dans les entiers), et une interprétation booléenne (lorsque le résultat est un booléen, ou une fonction des entiers dans les booléens). Dans cette définition, on utilise les conventions suivantes :

- c désigne une constante scalaire, et C désigne une constante parallèle ;
- x désigne une variable scalaire, et X désigne une variable parallèle ;
- s, s_1 et s_2 représentent des éléments de $Sarith$;
- V, V_1 et V_2 représentent des éléments de $Varith$;
- b, b_1 et b_2 représentent des éléments de $Sbool$;
- B, B_1 et B_2 représentent des éléments de $Vbool$.

Interprétation entière

Pour une expression de type $Sarith$,

$$\begin{array}{rcl}
[\![c]\!](\sigma) & = & c \\
[\![x]\!](\sigma) & = & \sigma(x) \\
[\![V|_s]\!](\sigma) & = & ([\![V]\!](\sigma))|_{([\![s]\!](\sigma))} \\
[\![s_1 \ Binop \ s_2]\!](\sigma) & = & [\![s_1]\!](\sigma) \ Binop \ [\![s_2]\!](\sigma) \\
[\![\text{if } b \text{ then } s_1 \text{ else } s_2]\!](\sigma) & = & \left\{ \begin{array}{l} [\![s_1]\!](\sigma) \text{ si } [\![b]\!](\sigma) = vrai \\ [\![s_2]\!](\sigma) \text{ si } [\![b]\!](\sigma) = faux \end{array} \right.
\end{array}$$

Pour une expression de type $Varith$,

$$\begin{array}{rcl}
[\![C]\!](\sigma) & = & C \\
[\![This]\!](\sigma) & = & This, \text{ où } This|_u = u \\
[\![X]\!](\sigma) & = & \sigma(X) \\
[\![V_1|_{V_2}]\!](\sigma) & = & V, \text{ où } V|_u = ([\![V_1]\!](\sigma))|_{([\![V_2]\!](\sigma))|_u} \\
[\![V_1 \ Binop \ V_2]\!](\sigma) & = & V, \text{ où } V|_u = ([\![V_1]\!](\sigma))|_u \ Binop \ ([\![V_2]\!](\sigma))|_u \\
[\![\text{IF } B \text{ THEN } V_1 \text{ ELSE } V_2 \text{ FI}]\!](\sigma) & = & V, \text{ où } V|_u = \left\{ \begin{array}{l} ([\![V_1]\!](\sigma))|_u \text{ si } ([\![B]\!](\sigma))|_u = vrai \\ ([\![V_2]\!](\sigma))|_u \text{ si } ([\![B]\!](\sigma))|_u = faux \end{array} \right.
\end{array}$$

Interprétation booléenne

Pour une expression de type $Sbool$,

$$\begin{array}{rcl}
[\![true]\!](\sigma) & = & vrai \\
[\![false]\!](\sigma) & = & faux \\
[\![B|_s]\!](\sigma) & = & ([\![B]\!](\sigma))|_{([\![s]\!](\sigma))} \\
[\![\neg \, b]\!](\sigma) & = & \neg \, [\![b]\!](\sigma) \\
[\![s_1 \ Rel \ s_2]\!](\sigma) & = & [\![s_1]\!](\sigma) \ Rel \ [\![s_2]\!](\sigma) \\
[\![b_1 \ Connect \ b_2]\!](\sigma) & = & [\![b_1]\!](\sigma) \ Connect \ [\![b_2]\!](\sigma)
\end{array}$$

Pour une expression de type $VBool$,

$$\begin{array}{rcl}
[\![TRUE]\!](\sigma) & = & B, \text{ où } B|_u = vrai \\
[\![FALSE]\!](\sigma) & = & B, \text{ où } B|_u = faux \\
[\![\neg \, B]\!](\sigma) & = & \neg \, ([\![B]\!](\sigma)) \\
[\![V_1 \ Rel \ V_2]\!](\sigma) & = & B, \text{ où } B|_u = ([\![V_1]\!](\sigma))|_u \ Rel \ ([\![V_2]\!](\sigma))|_u \\
[\![B_1 \ Connect \ B_2]\!](\sigma) & = & B, \text{ où } B|_u = ([\![B_1]\!](\sigma))|_u \ Connect \ ([\![B_2]\!](\sigma))|_u
\end{array}$$

Il reste alors à préciser la sémantique des formules booléennes $Sform$.

Interprétation des formules

Etant donné un environnement σ, interpréter une formule F dans σ revient à déterminer si cette formule est vraie ou non dans cet environnement.

Appliquée aux formules, la fonction $[\![]\!]$ a donc le profil :

$$[\![]\!] \ : \ Sform \rightarrow (env \rightarrow \{vrai, faux\})$$

et elle est définie par :

$$\begin{array}{rcl} [\![\neg F]\!](\sigma) & = & \neg [\![F]\!](\sigma) \\ [\![F_1 \ Connect \ F_2]\!](\sigma) & = & [\![F_1]\!](\sigma) \ Connect \ [\![F_2]\!](\sigma) \\ [\![\exists x \ . \ P]\!](\sigma) & = & vrai \text{ ssi } \exists c \in \mathbb{N} \text{ tel que } [\![P[c/x]]\!](\sigma) = vrai \end{array}$$

La sémantique de la formule «$\forall x \ . \ P$» s'obtient en utilisant l'équivalence suivante :

$$\forall x \ . \ p \equiv \neg (\exists x \ . \ (\neg p))$$

Dans la suite, on indiquera à l'aide de l'opérateur $\models$ le fait qu'un environnement σ *valide* une formule F :

$$\sigma \models F \text{ si et seulement si } [\![F]\!](\sigma) = vrai$$

Enfin, on termine en donnant la sémantique des assertions proprement dites.

Sémantique d'une assertion

En premier lieu, il est facile de voir que la sémantique que l'on a associée jusqu'ici aux langages d'état et de contexte permet bien d'une part d'exprimer des propositions sur les valeurs des variables d'un programme et, d'autre part, de déduire le contexte d'activité de ses processeurs, à un instant donné de son exécution.

Plus précisément, on dira qu'une assertion $\{P, C\}$ est *valide* dans un état du programme où l'environnement et le contexte valent respectivement σ et s si et seulement si :

- σ valide la formule P :

$$\sigma \models P$$

- la valeur de l'expression booléenne parallèle est identique au vecteur exprimant l'activité courante des processeurs :

$$\forall u \ . \ ([\![C]\!](\sigma))|_u = vrai \text{ ssi } actif(u)$$

  ce qui s'écrit également,

$$[\![C]\!](\sigma) = Top(s)$$

Formellement, on définira donc une fois encore une fonction $[\![]\!]$, qui, étant donnée une assertion, retourne l'ensemble des couples (état, contexte) pour lesquels elle est valide.

Définition 2-1

La fonction $[\![]\!]$ est définie sur l'ensemble des assertions de la manière suivante :

$$[\![]\!] \ : \ Assertion \rightarrow env \times ctxte$$

avec, pour toute assertion $\{P, C\}$,

$$[\![\{P, C\}]\!] = \{(\sigma, s) \mid \sigma \models P \ \wedge \ \sigma(C) = Top(s)\}$$

■

Là encore, l'opérateur $\models$ pourra être utilisé pour indiquer qu'un couple (σ, s) *valide* une assertion donnée :

$$(\sigma, s) \models \{P, C\} \text{ si et seulement si } (\sigma, s) \in [\![\{P, C\}]\!]$$

On définit une nouvelle relation entre les assertions.

Définition 2-2

Soit deux assertions $\{P, C\}$ et $\{Q, D\}$, on dit que $\{P, C\}$ implique $\{Q, D\}$, ce que l'on note $\{P, C\} \overset{C}{\Rightarrow} \{Q, D\}$, ssi on a :

$$(P \;\Rightarrow\; Q) \;\wedge\; (P \;\Rightarrow\; \forall u.(C|_u \;\Leftrightarrow\; D|_u))$$

■

Il est facile de voir que cette définition reste cohérente lorsque l'on compare des assertions en fonction de leur sémantique :

$$\{P, C\} \overset{C}{\Rightarrow} \{Q, D\} \quad \Rightarrow \quad [\![\{P, C\}]\!] \subseteq [\![\{Q, D\}]\!]$$

2.2 Substitutions

Afin de pouvoir déduire statiquement de nouvelles assertions sur le comportement d'un programme, il est nécessaire de disposer d'un mécanisme qui permette de modifier de manière symbolique l'expression représentée par une variable dans une assertion. Nous introduisons donc une fonction de *substitution* sur les variables parallèles, qui permet de remplacer, dans une expression E, une variable Y par une expression T, du même type que Y. La nouvelle expression obtenue est notée $E\,[T/Y]$.

Notons que, compte tenu de la sémantique que nous avons donnée aux assertions, la substitution définit en fait une interaction entre les langages d'état et de contexte, puisque les propositions sur les variables du programme sont toujours conditionnées par l'expression de contexte. Stewart [21] propose un mécanisme similaire pour l'instruction d'affectation *data-parallel* de FORTRAN 90, mais comme pour Gabarró et Gavaldà le contexte est toujours dénoté *explicitement* par un ensemble d'entiers, alors que dans notre méthode il est dénoté *implicitement*.

En supposant que Y est une variable parallèle et que T représente une expression de type $Varith$, et en conservant les conventions établies dans la section 2.1 pour les types des autres identificateurs, ce mécanisme de substitution peut alors être défini par induction comme suit.

Expressions scalaires :

$$\begin{array}{rcl}
c\,[T/Y] & = & c \\
true\,[T/Y] & = & true \\
false\,[T/Y] & = & false \\
x\,[T/Y] & = & x \\
V|_s\,[T/Y] & = & (V[T/Y])|_{(s[T/Y])} \\
B|_s\,[T/Y] & = & (B[T/Y])|_{(s[T/Y])} \\
(s_1\ Binop\ s_2)\,[T/Y] & = & s_1[T/Y]\ Binop\ s_2[T/Y] \\
(s_1\ Rel\ s_2)\,[T/Y] & = & s_1[T/Y]\ Rel\ s_2[T/Y]
\end{array}$$

$$
\begin{array}{rcl}
(b_1 \; Connect \; b_2)\,[T/Y] & = & b_1[T/Y] \; Connect \; b_2[T/Y] \\
(\neg\, b)\,[T/Y] & = & \neg\,(b[T/Y]) \\
(\text{if } b \text{ then } s_1 \text{ else } s_2)\,[T/Y] & = & \text{if } b[T/Y] \text{ then } s_1[T/Y] \text{ else } s_2[T/Y] \\
(\exists x \,.\, F)\,[T/Y] & = & \exists x \,.\, (F[T/Y]) \\
(\forall x \,.\, F)\,[T/Y] & = & \forall x \,.\, (F[T/Y])
\end{array}
$$

Expressions vectorielles :

$$
\begin{array}{rcl}
C\,[T/Y] & = & C \\
TRUE\,[T/Y] & = & TRUE \\
FALSE\,[T/Y] & = & FALSE \\
X\,[T/Y] & = & \begin{cases} T \text{ si } X = Y \\ X \text{ si } X \neq Y \end{cases} \\
(V_1 \; Binop \; V_2)\,[T/Y] & = & V_1[T/Y] \; Binop \; V_2[T/Y] \\
(V_1 \; Rel \; V_2)\,[T/Y] & = & V_1[T/Y] \; Rel \; V_2[T/Y] \\
(B_1 \; Connect \; B_2)\,[T/Y] & = & B_1[T/Y] \; Connect \; B_2[T/Y] \\
(\neg\, B)\,[T/Y] & = & \neg\,(B[T/Y]) \\
(\text{IF } B \text{ THEN } V_1 \text{ ELSE } V_2)\,[T/Y] & = & \text{IF } B[T/Y] \text{ THEN } V_1[T/Y] \text{ ELSE } V_2[T/Y]
\end{array}
$$

2.3 Quelques équivalences utiles

Après avoir défini la notion d'équivalence sur les composantes de notre langage d'assertions, nous présenterons quelques résultat utiles pour la manipulation de ce dernier, puis nous énoncerons une propriété fondamentale du mécanisme de substitution.

Equivalence de formules ou d'expressions

Nous définissons une notion d'équivalence entre des formules ou des expressions de la manière suivante :

Définition 2-3

Deux expressions E_1 et E_2 sont dites équivalentes, ce que l'on note $E_1 \equiv E_2$ si l'on a :

$$\forall \sigma.\; [\![E_1]\!]\,(\sigma) = [\![E_2]\!]\,(\sigma)$$

■

Définition 2-4

Deux formules F_1 et F_2 sont dites équivalentes, ce que l'on note $F_1 \equiv F_2$ si l'on a :

$$\forall \sigma.\; [\![F_1]\!]\,(\sigma) = [\![F_2]\!]\,(\sigma)$$

■

Equivalences pratiques

On termine cette section en établissant un certain nombre de propositions relatives aux substitutions qui seront utilisées dans la suite.

Proposition 2-1

$$
\begin{array}{rcl}
(V_1 \; Binop \; V_2)|_s & \equiv & V_1|_s \; Binop \; V_2|_s \\
(V_1 \; Rel \; V_2)|_s & \equiv & V_1|_s \; Rel \; V_2|_s \\
(B_1 \; Connect \; B_2)|_s & \equiv & B_1|_s \; Connect \; B_2|_s \\
(\neg B)|_s & \equiv & \neg (B|_s) \\
(\text{IF } B \text{ THEN } V_1 \text{ ELSE } V_2)|_s & \equiv & \text{if } B|_s \text{ then } V_1|_s \text{ else } V_2|_s \\
 & & \\
(\text{if } b \text{ then } s_1 \text{ else } s_2) \; Binop \; s_3 & \equiv & \text{if } b \text{ then } (s_1 \; Binop \; s_3) \text{ else } (s_2 \; Binop \; s_3) \\
(\text{if } b \text{ then } s_1 \text{ else } s_2) \; Rel \; s_3 & \equiv & \text{if } b \text{ then } (s_1 \; Rel \; s_3) \text{ else } (s_2 \; Rel \; s_3) \\
(\text{IF } B \text{ THEN } V_1 \text{ ELSE } V_2) \; Binop \; V_3 & \equiv & \text{IF } B \text{ THEN } (V_1 \; Binop \; V_3) \text{ ELSE } (V_2 \; Binop \; V_3) \\
(\text{IF } B \text{ THEN } V_1 \text{ ELSE } V_2) \; Rel \; V_3 & \equiv & \text{IF } B \text{ THEN } (V_1 \; Rel \; V_3) \text{ ELSE } (V_2 \; Rel \; V_3)
\end{array}
$$

■

Preuve :

La preuve consiste à vérifier que la fonction sémantique $[\![]\!]$ retourne bien des résultats identiques pour les membres gauches et droits de chacune des équivalences proposées. Il suffit ensuite d'utiliser l'implication suivante :

$$\forall \sigma \; . \; [\![E_1]\!](\sigma) = [\![E_2]\!](\sigma) \;\Rightarrow\; E_1 \equiv E_2$$

□

Propriétés de la substitution

On termine en donnant alors une proposition qui montre que, du point de vue de l'évaluation, substituer une variable parallèle dans une expression ou dans l'environnement dans lequel cette expression est évaluée, conduit à un résultat identique. Cette proposition étant assez intuitive, et ne dépendant en fait pas des caractéristiques particulières des langages que l'on considère ici pour décrire les expressions, sa preuve ne sera pas détaillée. Notons que l'on trouve une proposition similaire dans [1].

Proposition 2-2

$$[\![P]\!](\sigma[X \leftarrow [\![E]\!](\sigma)]) \;=\; [\![P[E/X]]\!](\sigma)$$

■

Preuve : Par induction sur la structure de P. □

En choisissant alors pour E l'expression «IF C THEN E_1 ELSE E_2», il est possible d'en déduire le corollaire suivant :

Corollaire 2-1
Pour tout environnement σ et formule P on a :

$$\sigma[X \leftarrow [\![\text{IF } C \text{ THEN } E_1 \text{ ELSE } E_2]\!](\sigma)] \models P \quad \text{ssi} \quad \sigma \models P[\text{IF } C \text{ THEN } E_1 \text{ ELSE } E_2/X]$$

■

Ce corollaire est à la base de la validité des axiomes du système de preuve que nous présentons dans la suite.

2.4 Le système de preuve

On rappelle le système de preuve qui décrit la sémantique axiomatique du langage $\mathcal{L}$. Compte-tenu des restrictions que nous avons imposées sur la structure des programmes que l'on considère (c.-à-d. à *contexte fixe* comme défini en section 1.3), on donne en fait ici une version simplifiée de ce système par rapport à la version originale proposée dans [23].

Ce système de preuve est constitué de deux axiomes, qui correspondent respectivement aux instructions d'affectation et de communications, et de trois règles d'inférences, associées aux instructions de composition séquentielle, de conditionnement, et d'itération. Rappelons que les objets manipulés sont des *triplets* de la forme $\{P, C\}\ S\ \{Q, D\}$, où $\{P, C\}$ et $\{Q, D\}$ sont des assertions et S est un fragment de programme $\mathcal{L}$ à *contexte fixe* telle que $Change(S) \cap (Var(C) \cup Var(D)) = \emptyset$, ceci afin de simplifier la gestion des variables de contextes.

Affectation :

$$\{P[\text{IF } C \text{ THEN } E \text{ ELSE } X/X], C\}\ X := E\ \{P, C\}$$

Communication :

$$\{P[\text{IF } C \text{ THEN } Y|_A \text{ ELSE } X/X], C\}\ \texttt{get}\ Y\ \texttt{from}\ A\ \texttt{into}\ X\ \{P, C\}$$

Composition séquentielle :

$$\frac{\{P, C\}\ S_1\ \{P', C\},\ \{P', C\}\ S_2\ \{Q, C\}}{\{P, C\}\ S_1\ ;\ S_2\ \{Q, C\}}$$

Conditionnement :

$$\frac{\{P, C\ \wedge\ B\}\ S\ \{Q, C\ \wedge\ B\}}{\{P, C\}\ \texttt{where}\ B\ \texttt{do}\ S\ \texttt{end}\ \{Q, C\}}$$

Itération :

$$\frac{\{I \wedge \exists u\,.\,(E|_u\ \wedge\ B|_u), E\}\ S\ \{I, E\}}{\{I, E\}\ \texttt{while}\ B\ \texttt{do}\ S\ end\ \{I\ \wedge\ \forall u\,.\,(E|_u\ \Rightarrow\ \neg B|_u), E\}}$$

Dans cette règle, l'assertion $\{I, E\}$ n'est autre que l'*invariant* de la boucle.

Enfin, on ajoute également à ce système une dernière règle d'inférence, la *règle de conséquence*, qui permettra de renforcer une précondition ou encore d'affaiblir une postcondition :

$$\frac{\{P,C\} \overset{C}{\Rightarrow} \{P',C'\},\ \{P', C'\}\ S\ \{Q', D'\},\ \{Q',D'\} \overset{C}{\Rightarrow} \{Q,D\}}{\{P, C\}\ S\ \{Q, D\}}$$

On notera classiquement par $\vdash \{P, C\}\ S\ \{Q, D\}$ le fait que le triplet $\{P, C\}\ S\ \{Q, D\}$ puisse être *déduit* du système de preuve.

Pour justifier le fait que ce système de preuve décrit bien la sémantique attendue pour le langage $\mathcal{L}$, il reste à établir le lien avec la sémantique opérationnelle qui était proposée dans la section 1.2. On donne pour ce faire une interprétation des triplets en termes de la machine abstraite qui nous avait permis de dériver cette sémantique opérationnelle.

Plus précisément, on dira qu'un triplet $\{P, C\}\ S\ \{Q, D\}$ est *valide* ce que l'on note :

$$\models \{P, C\}\ S\ \{Q, D\}$$

si et seulement si toute exécution terminée du programme S à partir d'un état de la machine qui vérifie l'assertion $\{P, C\}$, mène à un état qui satisfait l'assertion $\{Q, D\}$. Formellement, cette proposition s'écrit donc de la manière suivante :

$$\models \{P, C\}\ S\ \{Q, D\} \text{ ssi } [\![S]\!]\,([\![\{P, C\}]\!]\,) \subseteq [\![\{Q, D\}]\!]$$

Il resterait alors à montrer que le système de preuve est *correct*, c'est à dire que tout triplet qui peut être déduit est un triplet valide :

$$(\vdash \{P, C\}\ S\ \{Q, D\}) \Rightarrow (\models \{P, C\}\ S\ \{Q, D\})$$

Nous ne donnons pas cette preuve de correction qui sera détaillée par ailleurs.

Nous donnons ici deux lemmes, qui seront nécessaire par la suite, concernant l'*invariances des expressions de contexte* lorsque les fragments de programmes ne modifient pas ces dernières.

Lemme 2-1

Soit le triplet $\{P, C\}\ S\ \{Q, D\}$ tel que $Var(C) \cap Change(S) = \emptyset$. Si

$$\models \{P, C\}\ S\ \{Q, D\}$$

alors

$$\models \{P, C\}\ S\ \{Q, C\}$$

■

Preuve :

$$\models \{P, C\}\ S\ \{Q, D\}$$

signifie que l'on a :

$$\forall (\sigma, s) \in [\![\{P, C\}]\!]\,.[\![S]\!]\,(\sigma, s) = \{(\sigma', s')\} \subseteq [\![\{Q, D\}]\!]$$

or d'après la sémantique opérationnel on a $s' = s$, donc :

$$\forall (\sigma, s) \in [\![\{P, C\}]\!]\,.[\![S]\!]\,(\sigma, s) = \{(\sigma', s)\} \subseteq [\![\{Q, D\}]\!]$$

comme $Var(C) \cap Change(S) = \emptyset$, on a par induction sur la structure syntaxique de S :

$$\sigma(C) = \sigma'(C)$$

c.-à-d. :

$$\forall u.\sigma(C)|_u = \sigma'(C)|_u = Top(s)|_u$$

donc $(\sigma', s) \models \{Q, C\}$, on obtient finalement :

$$\models \{P, C\}\ S\ \{Q, C\}$$

□

On a le lemme symétrique suivant :

Lemme 2-2

Soit le triplet $\{P, C\}\ S\ \{Q, D\}$ tel que $Var(D) \cap Change(S) = \emptyset$. Si

$$\models \{P, C\}\ S\ \{Q, D\}$$

alors

$$\models \{P, D\}\ S\ \{Q, D\}$$

■

Preuve : La preuve est similaire à la précédente. □

Ces deux lemmes justifient aussi la restriction des programmes considérés, c.-à-d. qu'ils soient à *contexte fixe* et que pour tous triplets $\{P, C\}\ S\ \{Q, D\}$ on doit avoir :

$$(Var(C) \cup Var(D)) \cap Change(S) = \emptyset$$

3 La sémantique axiomatique comme outil de programmation

Nous allons nous intéresser ici à l'emploi de notre système de preuve comme outil de programmation. Dans un premier temps nous exhiberons la preuve complète d'un programme donné. Dans un deuxième temps, nous verrons comment la conception d'un programme peut être dirigée par sa preuve.

3.1 Une preuve complète de programme

Nous allons voir ici comment notre système de preuve peut nous aider à trouver et prouver ce que calcule un programme donné.

Considérons le programme M de la figure 1, le problème est de savoir ce qu'il calcule.

Après une première lecture syntaxique, on peut déjà présumer du rôle de chaque variable du programme :

- T est apparemment la variable d'entrée du programme, puisqu'elle est la seule non initialisée ;
- j est une variable compteur scalaire[1] ;
- S est sûrement la variable de sortie du programme parce qu'elle cumule plusieurs valeurs ;
- A est apparemment une variable auxiliaire car réinitialisée à chaque itération.

Maintenant que nous avons une information sur le type des variables, il faut s'intéresser aux valeurs calculées par le programme, c.-à-d. connaissant T, que vaut S à la fin de l'exécution.

Pour cela le premier réflexe est d'exécuter le programme avec un jeu d'essai. La figure 2 en est un exemple avec huit processeurs. En entrée T vaut 1 partout, et on constate qu'en sortie chaque élément de S est égal au numéro de processeur plus un. A priori le programme calcul pour chaque processeur la somme des valeurs de ces prédécesseurs. Ce qui équivaut à la fonction `scan`, ou somme préfixée, de la Connection Machine 2.

[1] par scalaire nous entendons une variable parallèle dont les composantes ont toutes la même valeur à chaque pas d'exécution du programme.

```
S := T;
j := 1;
while j < N do
      A := S;
      where This > j do
            Get A from This - j;
            S = S + A
            ◀
      end;
      j = j * 2
end
```

FIG. 1 - Programme mystère M

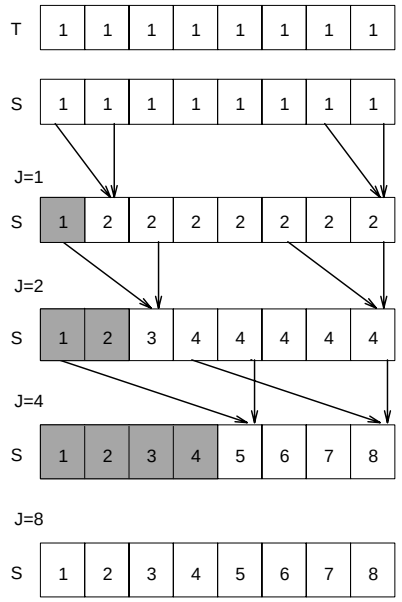

FIG. 2 - Essai du programme P

Mais cela n'est encore qu'une supposition, et n'est vrai que pour l'exemple très particulier du test.

Pour valider notre hypothèse dans un cadre plus général, nous allons appliquer notre système de preuve axiomatique. Nous allons chercher à prouver l'assertion suivante :

$$\{\forall u.1 \leq u \leq N \Rightarrow T|_u \in \mathbb{N}, true\}\ M\ \{\forall u.1 \leq u \leq N \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i, true\}$$

En examinant notre système et le programme P, on voit qu'il est nécessaire de dégager un invariant pour la boucle `while`.

Trouver un invariant de boucle est souvent une tâche délicate, car non seulement doit-il être correct, c.-à-d. rester stable après exécution du corps de la boucle, mais il doit être aussi suffisant pour prouver la condition qui suit immédiatement la boucle dans le programme (en l'occurrence la post-condition finale de la spécification de notre programme), mais pas trop pour que la condition précédente à la boucle le satisfasse.

La trace du programme où l'on aurait placé un point d'arrêt à la position «◀» de la figure 1, est aussi représentée sur la figure 2. Les valeurs ombrées sont celle des processeurs inactifs.

En s'attardant sur une étape particulière du calcul, par exemple pour $j = 4$, on peut remarquer que les valeurs de S en entrées de l'itération, c.-à-d. celle en sorties de l'itération précédente, satisfont les propriétés suivantes.

- Tous les processeurs inactifs à une itération, le restent pour les itérations suivantes, c.-à-d. que leur valeur pour S est la valeur finale. De plus celle-ci est la somme des valeurs de T de tous les prédécesseurs.
- Pour tous les processeurs actifs, c'est la somme des j prédécesseurs.

Ainsi, on peut formuler l'invariant de la boucle de la manière suivante :

$$I \equiv \forall u.(1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i) \wedge (j < u \leq N \Rightarrow S|_u = \sum_{i=u-j+1}^{i=u} T|_i)$$

On vérifie immédiatement si celui-ci est suffisant pour prouver la post-condition finale de notre programme. En effet la règle de la boucle `while` indique que l'on a comme post-condition :

$$\{\forall u.(1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i) \wedge (j < u \leq N \Rightarrow S|_u = \sum_{i=u-j+1}^{i=u} T|_i) \wedge (j \geq N), true\}$$

celle-ci implique effectivement la post-condition finale de notre spécification :

$$\{\forall u.1 \leq u \leq N \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i, true\}$$

Pour montrer qu'il est correct, regardons le programme M annoté par sa preuve de la figure 3, où toutes les assertions intermédiaires ont été générées en appliquant les axiomes d'affectation et de communication. On peut remarquer d'ailleurs que les assertions du `where` sont simples car l'invariant à été exprimé en tenant compte du contexte.

Le seul cas délicat dans cette preuve est de vérifier que (1)⇒(2), avec :

$$\begin{aligned}(1) \equiv \{&\forall u.1 \leq u \leq j \Rightarrow S|_u = \textstyle\sum_{i=1}^{i=u} T|_i \\ &\wedge j < u \leq N \Rightarrow S|_u = \textstyle\sum_{i=u-j+1}^{i=u} T|_i, true\}\end{aligned}$$

$\{\forall u.1 \leq u \leq N \Rightarrow T|_u \in \mathbb{N}, true\}$
$\{\forall u.1 \leq u \leq N \Rightarrow T|_u = \sum_{i=u-1+1}^{i=u} T|_i, true\}$

S `:=` T;

$\{\forall u.1 \leq u \leq 1 \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge 1 < u \leq N \Rightarrow S|_u = \sum_{i=u-1+1}^{i=u} T|_i, true\}$

j `:=` 1;

$\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq N \Rightarrow S|_u = \sum_{i=u-j+1}^{i=u} T|_i, true\}$

`while` $j < N$ `do`

(1) $\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq N \Rightarrow S|_u = \sum_{i=u-j+1}^{i=u} T|_i, true\}$

(2) $\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq j*2 \Rightarrow S|_u + S|_{u-j} = \sum_{i=1}^{i=u} T|_i$
$\wedge j*2 < u \leq N \Rightarrow S|_u + S|_{u-j} = \sum_{i=u-j*2+1}^{i=u} T|_i, true\}$

A `:=` S;

$\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq j*2 \Rightarrow S|_u + A|_{u-j} = \sum_{i=1}^{i=u} T|_i$
$\wedge j*2 < u \leq N \Rightarrow S|_u + A|_{u-j} = \sum_{i=u-j*2+1}^{i=u} T|_i, true\}$

`where This` $> j$ `do`

$\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq j*2 \Rightarrow S|_u + A|_{u-j} = \sum_{i=1}^{i=u} T|_i$
$\wedge j*2 < u \leq N \Rightarrow S|_u + A|_{u-j} = \sum_{i=u-j*2+1}^{i=u} T|_i, \texttt{This} > j\}$

`Get` A `from This -` j;

$\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq j*2 \Rightarrow S|_u + A|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j*2 < u \leq N \Rightarrow S|_u + A|_u = \sum_{i=u-j*2+1}^{i=u} T|_i, \texttt{This} > j\}$

S `=` S `+` A

$\{\forall u.1 \leq u \leq j*2 \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j*2 < u \leq N \Rightarrow S|_u = \sum_{i=u-j*2+1}^{i=u} T|_i, \texttt{This} > j\}$

`end;`

$\{\forall u.1 \leq u \leq j*2 \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j*2 < u \leq N \Rightarrow S|_u = \sum_{i=u-j*2+1}^{i=u} T|_i, true\}$

j `=` j `*` 2

$\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq N \Rightarrow S|_u = \sum_{i=u-j+1}^{i=u} T|_i, true\}$

`end`

$\{\forall u.1 \leq u \leq j \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i$
$\wedge j < u \leq N \Rightarrow S|_u = \sum_{i=u-j+1}^{i=u} T|_i$
$\wedge j \geq N, true\}$
$\{\forall u.1 \leq u \leq N \Rightarrow S|_u = \sum_{i=1}^{i=u} T|_i, true\}$

FIG. 3 - Programme M annoté par sa preuve

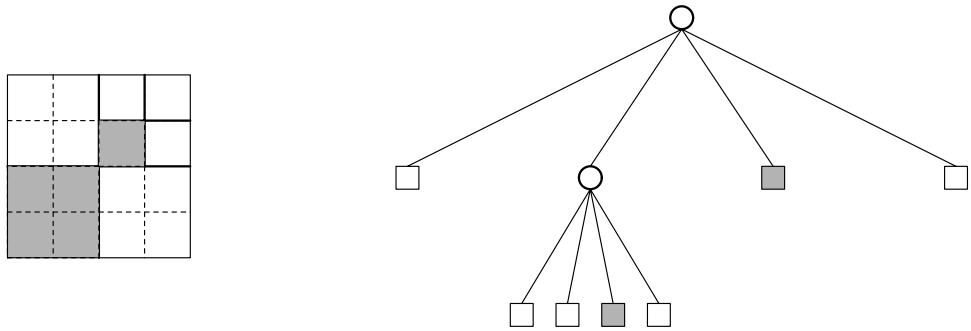

FIG. 4 - Quadtree et récursivité du maillage carré

$$
\begin{aligned}
(2) \equiv \{ & \forall u . 1 \leq u \leq j \Rightarrow S|_u = \textstyle\sum_{i=1}^{i=u} T|_i \\
& \wedge j < u \leq j * 2 \Rightarrow S|_u + S|_{u-j} = \textstyle\sum_{i=1}^{i=u} T|_i \\
& \wedge j * 2 < u \leq N \Rightarrow S|_u + S|_{u-j} = \textstyle\sum_{i=u-j*2+1}^{i=u} T|_i, true\}
\end{aligned}
$$

Dans la condition (2), le cas où $1 \leq u \leq j$ se déduit directement de la condition (1). Pour le cas où $j < u \leq j * 2$, il suffit de remarquer que $u - j \leq j$ pour voir que celui-ci se déduit aussi de (1). De même pour le cas où $j * 2 < u \leq N$, remarquer que $u - j > j$ suffit à prouver l'implication.

Enfin on voit que l'invariant «remonte» bien vers la pré-condition initiale de notre spécification.

Nous avons donc employé notre sémantique afin de découvrir et prouver ce que calcule un programme fixé. Pour cela nous avons étudié une trace de son exécution afin de dégager une propriété constante associée à un point d'arrêt. Cette méthode est celle qui a été préconisée par Peter Naur [20], et qui a été justement à la genèse de la sémantique axiomatique.

3.2 Conception d'un programme dirigée par sa preuve

Nous allons maintenant illustrer l'utilité de notre système dans la conception conjointe d'un programme et de sa preuve. L'exemple choisi est la génération de *quadtrees* à partir d'images binaires. Le quadtree est une structure de donnée largement utilisée dans le traitement d'image. Comme exemple d'application on peut citer les travaux de P. Bonnin sur la segmentation coopérative contour/région [5], dont le programme présenté ici est issu d'une mise en œuvre en POMPC [3].

3.2.1 Approche abstraite

La notion de *quadtree* est liée à la propriété de récursivité du maillage carré. On peut représenter une image discrète par un arbre quaternaire. Chaque nœud de l'arbre correspond à un pavé unique de l'image, de taille dépendant de sa profondeur dans l'arbre. Le niveau le plus bas étant celui du pixel du capteur. La figure 4 est un exemple de quadtree, où l'arbre est construit en observant l'image de gauche à droite et de haut en bas.

Un quadtree est une bonne représentation d'une image discrète si chaque feuille de l'arbre représente une zone homogène de celle-ci, auquel cas on étiquette chaque feuille avec la «couleur» de la zone. On dit alors qu'il est «correct» relativement à l'image.

Parmi ces quadtrees «corrects», il en existe un qui a un nombre minimal de nœuds : c'est celui où toutes les feuilles ont un frère qui ne décrit pas une zone homogène de l'image, ou alors d'une autre couleur, tel que celui présenté en figure 4.

C'est un algorithme à parallélisme de données de construction d'un quadtree minimal à partir d'une image binaire de dimension $2^N \times 2^N$ que nous nous proposons de construire ici, en nous aidant de la sémantique axiomatique.

Dans un premier temps, nous considérons que nous pouvons traiter directement les quadtrees sans nous préoccuper de leur implémentation. La structure de données parallèle (*collection* au sens de POMPC) est l'arbre quaternaire complet. Les variables sont :

$Feuille$: un booléen qui indique pour chaque nœud s'il est une feuille d'un quadtree correct ;

Cl : indique la couleur de la zone homogène représentée par la feuille ;

$Niveau$: une constante entière indiquant à quel niveau ce trouve le nœud dans l'arbre quaternaire.

Si l'on part d'une image quelconque, le quadtree complet où la variable $Feuille$ n'est vraie que pour les nœuds de niveau le plus bas ($Niveau = 0$), et où Cl est alors fixée à celle du pixel, est un quadtree correct par rapport à l'image. La spécification initiale du programme cherché est donc :

$$\{\forall n.Niveau|_n = 0 \Leftrightarrow Feuille|_n, true\}$$

Si l'on est capable d'énumérer tous les quadtrees corrects, il est alors aisé de trouver le minimal. On peut représenter tous les quadtrees corrects sur notre structure si tous les nœuds représentant une zone homogène de l'image sont étiquetés par $Feuille = tt$. Nous considérons que cette représentation intermédiaire est la spécification finale de notre programme, c.-à-d. plus formellement :

$$\{\forall n.(Feuille|_n \Rightarrow homogène(n)) \wedge (\neg Feuille|_n \Rightarrow \neg homogène(n)), true\}$$

avec le prédicat $homogène(n)$ est défini comme suit, et où $fils(n)$ représente l'ensemble des quatre fils du nœud n :

- $\forall n.(Niveau|_n = 0 \Rightarrow homogène(n) = true)$
- $\forall n.Niveau|_n > 0 \Rightarrow homogène(n) = (\forall n' \in fils(n).homogène(n')) \wedge (\forall (n', n'') \in fils(n)^2.Cl|_{n'} = Cl|_{n''})$

La spécification de notre programme est résumée par le schéma de la figure 5.

De la définition d'homogénéité, on voit que l'état initial du programme satisfait la condition suivante :

$$\{\forall n.Niveau|_n = 0 \Rightarrow (Feuille|_n \Rightarrow homogène(n) \wedge \neg Feuille|_n \Rightarrow \neg homogène(n)), true\}$$

D'où l'idée de construire le programme avec une boucle d'indice j ayant comme invariant :

$$I \equiv \forall n.(Niveau|_n \leq j \Rightarrow (Feuille|_n \Rightarrow homogène(n) \wedge \neg Feuille|_n \Rightarrow \neg homogène(n)))$$

Celui-ci est donc vérifié par l'état initial ($j = 0$), et pour $j = N - 1$ l'invariant satisfait la condition finale.

On a donc le synopsis de programme de la figure 6, où nous avons introduit une nouvelle variable H qui représente le résultat du test d'homogénéité. Nous avons juste décrit les propriétés

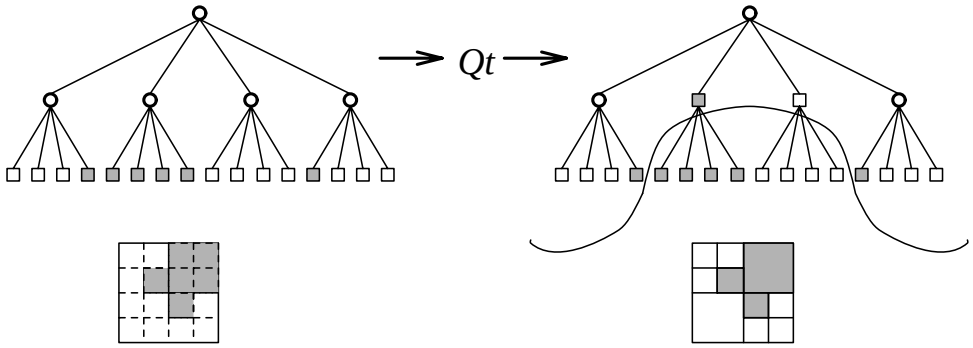

FIG. 5 - Spécification du programme de quadtree Qt

$\{\forall n.(Niveau|_n = 0 \Leftrightarrow Feuille|_n), true\}$
$\{I[0/j], true\}$
j := 0;
$\{I, true\}$
while $j < N - 1$ do
$\{I \wedge j < N - 1, true\}$
S;
$\{I \wedge \forall n.(Niveau|_n = j + 1 \Rightarrow (H|_n \Leftrightarrow homogène(n))), true\}$
$\{I[IF\ (Niveau = j + 1 \wedge H)\ THEN\ true\ ELSE\ Feuille/Feuille,$
$j + 1/j], true\}$
where $Niveau = (j + 1)$ do
where H do
$Feuille$:= $true$;
Get Cl from fils into Cl
end
end;
$\{I[j + 1/j], true\}$
j := j + 1
$\{I, true\}$
end
$\{I \wedge j = N - 1, true\}$
$\{\forall n.((Feuille|_n \Rightarrow homogène(n)) \wedge (\neg Feuille|_n \Rightarrow \neg homogène(n))), true\}$

FIG. 6 - Synopsis du programme de quadtree Qt

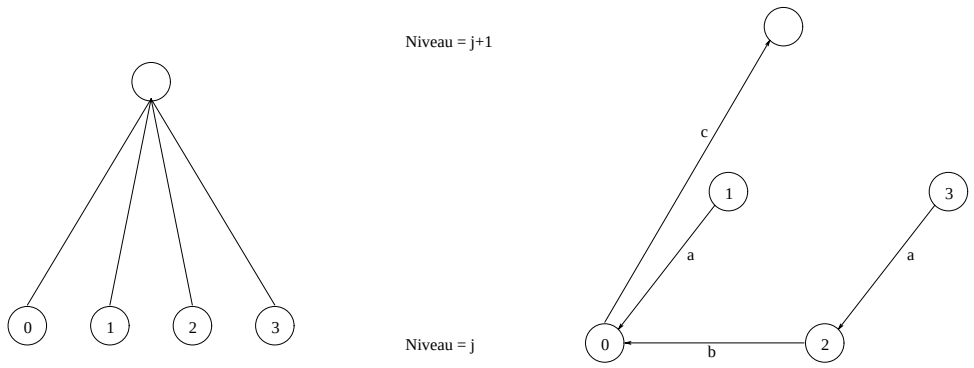

FIG. 7 - *Rang* des nœuds et évaluation du test d'homogénéité

logiques qu'elle doit satisfaire pour conserver l'invariant. Il nous faut donc trouver un morceau de programme S qui satisfasse la spécification suivante :

$$\{I \wedge j < N-1, true\}\ S\ \{I \wedge \forall n.(Niveau|_n = j+1 \Rightarrow (H|_n \Leftrightarrow homogène(n))), true\}$$

Pour calculer H il est nécessaire que les nœuds communiquent. Sans nous préoccuper du réseau de communication, considérons la nouvelle variable *Rang* qui dénote le rang des nœuds dans leur filiation par rapport à leur père.

Comme représenté sur la figure 7, on voit que l'on peut calculer le test d'homogénéité par une réduction au niveau des fils (communication a et b) pour déterminer si tous les fils représentent une zone homogène (c.-à-d. qu'ils soient tous des feuilles de couleur identique), puis par une lecture du résultat par le père sur un fils privilégié (communication c). Ceci par définition de l'homogénéité et grâce à l'invariant qui nous assure que :

$$\forall n.(Niveau|_n \leq j \Rightarrow (Feuille|_n \Leftrightarrow homogène(n)))$$

ce qui est le cas pour les fils. Le code réalisant ce calcul est présenté dans la figure 8.

3.2.2 Implémentation

Nous avons dégagé le squelette du programme en raisonnant sur une structure de données abstraite. Maintenant nous allons nous intéresser à l'implémentation de celui-ci sur une structure de données manipulable par $\mathcal{L}$, c.-à-d. les *tableaux*.

Un quadtree est entièrement spécifié par la liste de ses feuilles. De plus le nombre de feuilles maximales est égal au nombre de pixels de l'image initiale. D'où l'idée de plonger le quadtree dans un tableau à deux dimensions comme le montre la figure 9.

Chaque nœud de l'arbre est projeté dans le pixel supérieur gauche du pavé qu'il décrit. Ainsi le pixel de coordonnées nulles décrit N nœuds, qui correspondent à ceux rencontrés lors d'un parcours des branches de l'arbre les plus à gauche. Il suffit juste alors de savoir pour chaque pixel quel est le niveau maximum des nœuds décrits qui sont encore des feuilles du quadtree. Nous introduisons

```
        {I ∧ j < N − 1, true}
where Niveau = j do
      H := Feuille;
      where Rang pair do
            Get H from frère de Rang + 1 into H';
            Get Cl from frère de Rang + 1 into Cl'
      end;
      H := (H ∧ H') ∧ (Cl = Cl');
      where Rang = 0 do
            Get H from frère de Rang + 2 into H';
            Get Cl from frère de Rang + 2 into Cl'
      end;
      H := (H ∧ H') ∧ (Cl = Cl')
end;
where Niveau = j + 1 do
      Get H from fils de Rang 0 into H
end
        {I ∧ ∀n.(Niveau|_n = j + 1 ⇒ (H|_n ⇔ homogène(n))), true}
```

FIG. 8 - Extrait du programme S pour le calcul de H

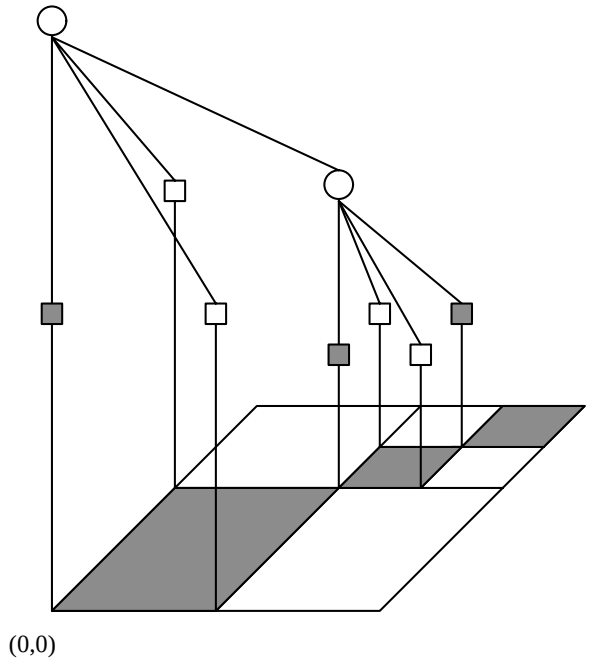

FIG. 9 - Plongement d'un quadtree dans un tableau 2D

une variable tableau à deux dimensions[2] Niv qui satisfait la propriété suivante :

$$I_r \equiv \forall(x,y).(Niv|_{(x,y)} = k) \Leftrightarrow (\forall n \in noeud(x,y). \quad Niveau|_n \leq k \Rightarrow Feuille|_n \\ \wedge \quad Niveau|_n > k \Rightarrow \neg Feuille|_n)$$

$noeud(x,y)$ étant l'ensemble des nœuds décrit par le pixel de position (x,y).

Remarquons que les pixels qui décrivent les nœuds de $Niveau = j$ sont ceux dont les coordonnées sont des multiples de 2^j :

$$\forall(x,y).(\exists n \in noeud(x,y).Niveau|_n = j) \Leftrightarrow (\exists k,l.x = k * 2^j \wedge y = l * 2^l)$$

On peut de même définir la fonction $Rang$ pour un niveau j donné par :

$$Rang(j,(x,y)) = \begin{cases} 0 & si \quad \exists k. \quad x = k * 2^{j+1} \quad et \quad \exists l. \quad y = l * 2^{j+1} \\ 1 & si \quad \neg\exists k. \quad x = k * 2^{j+1} \quad et \quad \exists l. \quad y = l * 2^{j+1} \\ 2 & si \quad \exists k. \quad x = k * 2^{j+1} \quad et \quad \neg\exists l. \quad y = l * 2^{j+1} \\ 3 & si \quad \neg\exists k. \quad x = k * 2^{j+1} \quad et \quad \neg\exists l. \quad y = l * 2^{j+1} \end{cases}$$

Remarquons qu'un père et son fils de $Rang = 0$ se situent sur le même processeur, d'où l'économie d'une communication sur le programme final de la figure 10. Dans cette version les variables Cl, H et H' sont des variables tableaux, x et y sont des constantes indiquant les coordonnés de chaque pixel.

Le programme final résultant est non trivial. Résumons la démarche suivie pour y aboutir.

Tout d'abord nous avons *spécifié* le problème sur une structure de donnée *abstraite* : l'arbre quaternaire. Après avoir noté les propriétés de celui-ci (*récursivité* liée au maillage carré et à la définition d'homogénéité), nous avons dégagé un *invariant* de boucle I. Nous avons introduit une nouvelle variable H, qui spécifiait l'ensemble des nœuds qui devaient être mis à jour pour conserver l'invariant. Nous avons ensuite, à partir de sa spécification, exhibé un fragment de programme calculant correctement celle-ci. Nous avons donc utilisé la propriété de *compositionnalité* de la sémantique axiomatique pour construire le corps de la boucle.

Pour la phase d'*implémentation*, il a suffit d'établir une équivalence entre la représentation abstraite du quadtree et la variable tableaux Niv par l'introduction de l'invariant I_r, pour exhiber le programme final qui conserve la même structure que la version intermédiaire.

3.3 Discussion

De ces deux applications de notre sémantique axiomatique, on constate finalement que les méthodes employées sont très proches de celle du cas séquentiel. Cela est réconfortant pour l'utilisateur final, car une fois qu'il aura assimilé les notions spécifique au parallélisme de données tel que celles de *localité* et de *contexte*, il pourra employer les mêmes méthodes d'approche pour les problèmes qu'il aura à résoudre. Ces méthodes sont le fruit de plus de 20 années de recherche autour de la logique de Hoare et de la construction de programme en général, qui sont couramment employées et enseignées par la communauté informatique d'aujourd'hui.

L'application la plus importante est bien sûr l'aide à la conception de programme. Généralement quand un programme est écrit, il devient difficile de le prouver. Il faut donc systématiquement développer tout programme avec sa preuve, sous forme d'assertions intermédiaires. Il est clair que

[2] Jusqu'à présent nous avons considéré que $\mathcal{L}$ manipulait des vecteurs, c.-à-d. des tableaux de dimension un, ici nous considérons que $\mathcal{L}$ peut manipuler des matrices. On peut se ramener bien sûr à des vecteurs en y plongeant ces dernières, mais cela alourdirait inutilement les notations.

```
        {∀n.Niveau|_n = 0 ⇔ Feuille|_n, true}
        {I[0/j] ∧ I_r[0/Niv], true}
j := 0;
Niv := 0;
        {I ∧ I_r, true}
while j < N − 1 do
      where x%2^j = 0 ∧ y%2^j = 0 do
            H := Niv = j;
            where x%2^(j+1) = 0 do
                  Get H from (x + 2^j, y) into H';
                  Get Cl from (x + 2^j, y) into Cl';
                  H := (H ∧ H') ∧ (Cl = Cl')
            end
      end;
      where x%2^(j+1) = 0 ∧ y%2^(j+1) = 0 do
            Get H from (x, y + 2^j) into H';
            Get Cl from (x, y + 2^j) into Cl';
            H := (H ∧ H') ∧ (Cl = Cl');
            where H do
                  Niv := j + 1
            end
      end;
      j := j + 1
end
        {I ∧ I_r ∧ j = N − 1, true}
        {I_r ∧ ∀n.Feuille|_n ⇒ homogène(n) ∧ ¬Feuille|_n ⇒ ¬homogène(n), true}
```

FIG. 10 - Programme de quadtree final

pour ne pas freiner le processus de développement, le degré de formalisme des assertions doit rester faible, c.-à-d. se limiter aux assertions fortes tel que les spécifications initiales/finales et les invariants. En fait, on ramène la conception d'un programme à la recherche des invariants. Il existe maintenant des heuristiques pour trouver ceux-ci. Celle que nous avons employée pour le quadtree est connue sous le nom de *relaxation des constantes*, d'autres peuvent être trouvées dans [15].

Il est clair qu'un relâchement du degré de formalisme dans la conception d'un programme avec sa preuve réintroduit un risque majeur d'erreur. Si on veut une preuve correcte du programme, il faut redescendre à un formalisme strict. C'est une tâche fastidieuse, qui heureusement, peut être automatisée comme nous le verrons dans la partie suivante.

4 Automatiser la preuve de programmes $\mathcal{L}$

Nous avons donc illustré «à la main» comment la sémantique axiomatique est utilisée pour *prouver* les programmes. Elle consiste à spécifier le problème sous la forme d'un triplet $\{P, C\}\ S\ \{Q, D\}$ où, d'après la sémantique des assertions, on a :

- la précondition $\{P, C\}$ détermine l'ensemble des état possibles, dans lequel est supposé se trouver le programme à son initialisation ;
- la postcondition $\{Q, D\}$ représente l'ensemble des états autorisés, dans lequel doit nécessairement se trouver le programme après l'une de ses exécutions terminées.

Par conséquent, pour vérifier que S satisfait ces spécifications, une approche envisageable consiste à montrer que le triplet $\{P, C\}\ S\ \{Q, D\}$ peut être *déduit* du système de preuve, la correction de ce dernier impliquant la *validité* du triplet.

On présente tout d'abord globalement l'approche qui a été retenue pour automatiser cette méthode de preuve, ainsi que l'architecture générale d'un outil d'aide à la preuve qui en découle. On décrit alors de façon plus précise comment cette solution peut être spécifiée dans le cas particulier du langage $\mathcal{L}$, et on la justifie formellement.

4.1 Une stratégie automatisable pour générer des preuves

Comme il a été rappelé au paragraphe précédent, une approche possible pour prouver un programme $\mathcal{L}$ consiste à décider si un énoncé $\{P, C\}\ S\ \{Q, D\}$ peut ou non être déduit du système de preuve qui définit la sémantique axiomatique de ce langage. Toutefois, en raison de l'expressivité du langage que l'on a choisi pour décrire les assertions (l'arithmétique de Peano), ce problème n'est pas décidable, en particulier dans la recherche des invariants [9].

Par contre, si l'on dispose d'une «preuve» supposée correcte du triplet $\{P, C\}\ S\ \{Q, D\}$, c.-à-d. d'une séquence

$$\{P_0, C_0\}\ S_0\ \{Q_0, D_0\},\ \{P_1, C_1\}\ S_1\ \{Q_1, D_1\},\ \ldots,\ \{P_n, C_n\}\ S_n\ \{Q_n, D_n\} = \{P, C\}\ S\ \{Q, D\}$$

il est alors possible de décider de manière automatique si une telle preuve est effectivement valide dans notre système, c.-à-d. si chaque triplet $\{P_i, C_i\}\ S_i\ \{Q_i, D_i\}$ est soit un axiome, soit obtenu à partir des triplets précédents à l'aide d'une règle d'inférence.

Par conséquent, la solution que l'on envisage ici pour automatiser cette méthode de preuve de programme est une approche *semi-automatique*, qui consiste à implémenter respectivement :

- un *proof-generator*, c.-à-d. un outil d'aide à la génération de preuves au sein du système de preuve ;

– un *proof-checker*, c.-à-d. un outil qui permet de vérifier si une preuve donnée est ou non valide dans ce système.

On détaille cette solution plus précisément dans la suite, en nous intéressant tout d'abord au problème de la génération de la preuve.

Tel qu'il a été proposé, le système de preuve décrit une méthode formelle pour prouver des énoncés, mais il ne fournit pas un algorithme qui permette d'en générer automatiquement une preuve. Il est donc nécessaire en pratique de lui adjoindre une stratégie qui détermine en particulier dans quel ordre les règles doivent être appliquées à chaque étape de la preuve. La stratégie qui a été choisie ici est directement inspirée des résultas obtenus dans le cas des langages séquentiels, c.-à-d. un calcul des *plus faibles préconditions* ([10]). Intuitivement, étant donné un programme S et une postcondition $\{Q, D\}$ pour S, il est possible de construire l'assertion $\{P_0, C_0\}$ qui correspond à sa précondition la plus large (au sens de l'implication). Par suite, décider si un énoncé $\{P, C\}\ S\ \{Q, D\}$ est valide revient à vérifier que $\{P_0, C_0\}$ contient bien $\{P, C\}$.

L'origine du choix d'un calcul de précondition, qui revient à effectuer les preuves «en arrière», vient du fait que, comme dans le cas séquentiel, les axiomes de notre système de preuve ne peuvent être appliqués de manière constructive que si on les suppose orientés de droite à gauche. Plus précisément, l'axiome

$$\{P[\text{IF } C \text{ THEN } E \text{ ELSE } X/X], C\}\ X := E\ \{P, C\}$$

permet de calculer la précondition associée à une postcondition $\{P, C\}$ donnée, mais la réciproque n'est pas vraie. Enfin, notons également que la règle d'inférence associée à l'instruction d'itération n'est constructive que si les invariants sont connus. Ces assertions n'étant pas calculable en pratique, la solution la plus raisonnable consiste à supposer qu'ils sont fournis avec le programme à vérifier. Toutefois, il reste nécessaire de s'assurer que ces invariants sont corrects, c.-à-d. qu'il représentent bien des propositions invariantes lors de l'exécution des itérations du programme.

Par conséquent, une vérification semi-automatique de la validité d'un énoncé du type :

$$\{P, C\}\ S\ \{Q, D\}$$

peut être mise en œuvre selon l'algorithme informel suivant :

1. *annoter* à la main le programme S en lui ajoutant ses invariants ;
2. *calculer* la plus faible précondition $\{P_0, C_0\}$ associée à $\{Q, D\}$ et au programme S annoté ;
3. *montrer* que $\{P_0, C_0\}$ contient $\{P, C\}$ et que les invariants qui ont été ajoutés sont corrects.

Les étapes 2 et 3 sont spécifiées plus en détail dans les sections 4.2 et 4.3. On montre en particulier que les conditions qui doivent être vérifiées dans l'étape 3 peuvent être exprimées par un ensemble de formules de logique du premier ordre, usuellement désignées sous le terme de *verification conditions* ([14]). Prouver que ces propositions sont correctes suffit donc pour montrer que $\{P, C\}\ S\ \{Q, D\}$ est valide.

On termine en décrivant brièvement sur la figure 11 l'architecture d'un outil d'aide à la preuve, et en indiquant les différentes ressources nécessaires à chacune des étapes.

Notons que la phase de génération des *verification conditions* peut être assimilée à un processus de compilation : elle transforme un problème décrit dans un langage «utilisateur» (celui des triplets), en une «implémentation» exprimée dans un langage «exécutable» par un *theorem prover* classique (la logique du premier ordre).

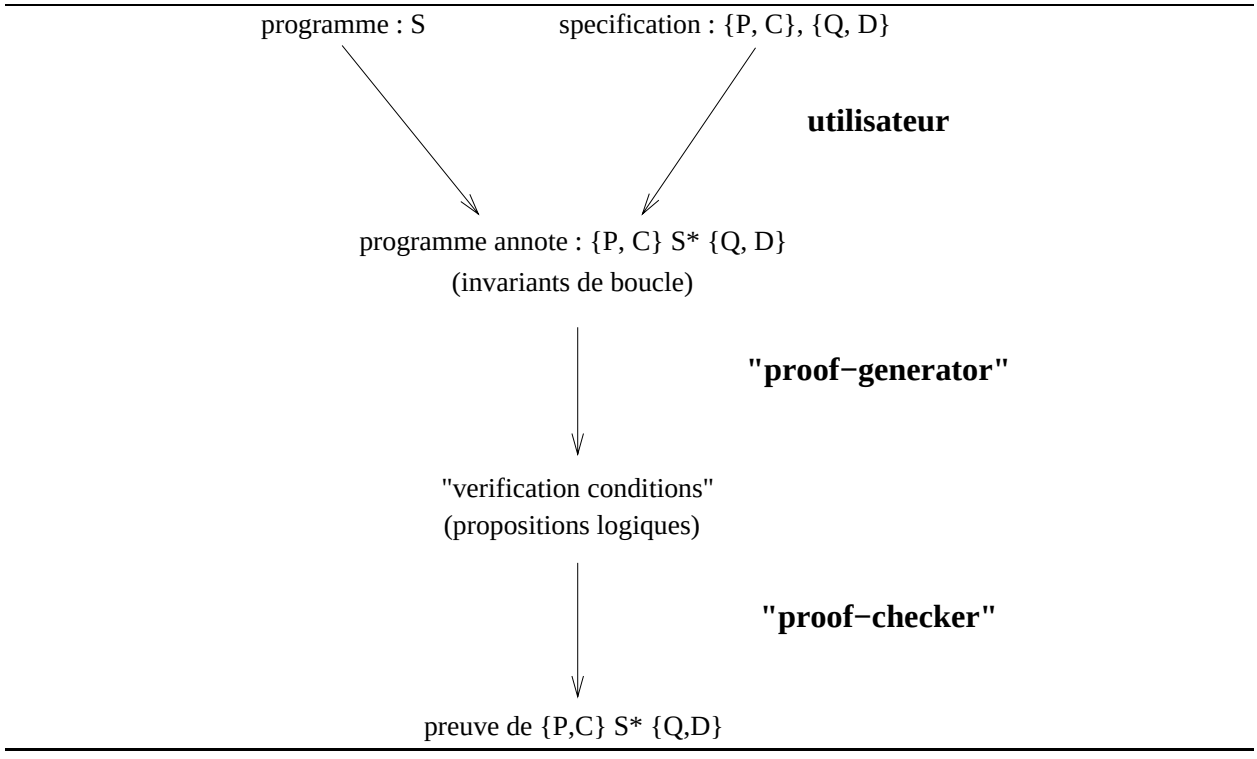

FIG. 11 - Architecture d'un outil d'aide à la preuve

4.2 Calcul des plus faibles préconditions

La fonction wp permet de calculer la plus faible précondition associée à un fragment de programme S et une postcondition $\{Q, D\}$ donnés, c.-à-d. l'ensemble des états pour lesquels une exécution terminée de S mène à un état qui satisfait $\{Q, D\}$.

Formellement, cette fonction peut donc être définie de la manière suivante : Rappelons que l'ensemble Pgm des programmes considérés, sont les programmes à *contexte fixe*.

Définition 4-1

Soit wp la fonction définie par :

$$wp : Pgm \times 2^{env \times ctxte} \rightarrow 2^{env \times ctxte}$$

telle que :

$$wp(«S», \mathcal{E}C) = \{(\sigma, s) \mid [\![S]\!](\sigma, s) \subseteq \mathcal{E}C\}$$

■

Par abus de notation, lorsque l'ensemble des états du second paramètre de la fonction wp sera dénoté par une assertion on notera :

$$wp(«S», \{Q, D\})$$

au lieu de

$$wp(«S», [\![\{Q, D\}]\!])$$

Il est facile de s'assurer que cette définition de wp vérifie bien les propriétés «de plus faible précondition» attendues. Plus précisément, si $[\![\{P, C\}]\!] = wp(S, \{Q, D\})$ alors les deux propositions suivantes sont vérifiées :

(i) $\{P, C\}$ est une précondition : $\models \{P, C\}\ S\ \{Q, D\}$

(ii) Tout autre précondition est plus forte :

$$\forall \{P', C'\} \,.\, (\models \{P', C'\}\ S\ \{Q, D\}) \ \Rightarrow\ (\{P', C'\} \overset{C}{\Rightarrow} \{P, C\})$$

Notons que dans les deux propositions précédentes, nous avons fait l'hypothèse qu'il existe une assertion qui dénote entièrement l'ensemble des états d'une précondition la plus faible. Cette hypothèse sous-entend que notre langage d'assertion soit «assez *expressif*» pour décrire de tels ensembles dans tous les cas. C'est un point très délicat à vérifier qui n'est pas lié au système d'inférence, mais au langage de spécification choisi. Il est par contre nécessaire pour prouver la *complétude* du système de preuve. Mais ce n'est pas notre propos ici, nous ne ferons donc pas appel à cette notion d'expressivité, et c'est pour cela que la fonction *wp* dénote des ensembles d'états plutôt que des assertions comme c'est souvent le cas dans la littérature. Pour plus d'informations sur ces problèmes le lecteur peut se référer à [8] et [1].

A partir de la définition 4-1, la proposition 4-1 établit alors, pour chaque instruction du langage $\mathcal{L}$, la plus faible précondition associée à une postcondition donnée. En fait, on ne donne pas l'expression exacte de *wp* mais uniquement une valeur approchée, obtenue sous certaines hypothèses. Cette approximation sera toutefois suffisante dans la suite.

Proposition 4-1

Soit $\{Q, D\}$ une assertion. Pour chaque instruction S de $\mathcal{L}$ tel que $Var(D) \cap Change(S) = \emptyset$, la fonction $wp\,(S, \{Q, D\})$ vérifie les conditions suivantes :

affectation :

$$wp\,(\text{«}X := E\text{»}, \{Q, D\}) = [\![\{Q\,[\text{IF } D \text{ THEN } E \text{ ELSE } X/X],\, D\}]\!]$$

communication :

$$wp\,(\text{«}\texttt{get}\ Y\ \texttt{from}\ A\ \texttt{into}\ X\text{»}, \{Q, D\}) = [\![\{Q\,[\text{IF } D \text{ THEN } Y|_A \text{ ELSE } X/X],\, D\}]\!]$$

composition séquentielle :

$$wp\,(\text{«}S_1\,;\,S_2\text{»}, \{Q, D\}) = wp\,(\text{«}S_1\text{»}, wp\,(\text{«}S_2\text{»}, \{Q, D\}))$$

conditionnement :

$$\begin{array}{ll} & \{(\sigma, s) \mid (\sigma, Push(Top(s) \wedge \sigma(B), s)) \in wp(\text{«}S\text{»}, \{Q, D \wedge B\}) \wedge Top(s) = \sigma(D)\} \\ \subseteq & wp\,(\text{«}\texttt{where}\ B\ \texttt{do}\ S\ \texttt{end}\text{»}, \{Q, D\}) \end{array}$$

itération : on a les deux propositions suivantes :

1. $$wp(\text{«}\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\text{»}, \{Q, D\}) \cap [\![\{\forall u.(D|_u \Rightarrow \neg B|_u), D\}]\!] \subseteq [\![\{Q, D\}]\!]$$

2. $$\begin{array}{ll} & wp(\text{«}\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\text{»}, \{Q, D\}) \cap [\![\{\exists u.(D|_u \wedge B|_u), D\}]\!] \\ \subseteq & wp(\text{«}S\text{»}, wp(\text{«}\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\text{»}, \{Q, D\})) \end{array}$$

■

Preuve : on montre que, pour chaque instruction, l'expression proposée se déduit bien de la définition générale de la fonction *wp* (définition 4-1).

Affectation :

$$wp\,(\text{«}X := E\text{»}, \{Q, D\}) = \{(\sigma, s) \mid [\![X := E]\!](\sigma, s) = \{(\sigma', s)\} \subseteq [\![\{Q, D\}]\!]\,\}$$

avec :

$$\sigma' = \sigma[X \leftarrow V]$$

où V est défini comme suit :

$$\begin{array}{rcll} V|_u & = & \sigma(E)|_u & \text{si} \quad actif(u) \\ V|_u & = & \sigma(X)|_u & \text{si} \quad \neg actif(u) \end{array}$$

Or $actif(u) = \sigma(D)|_u$, donc par définition :

$$V \quad = \quad [\![\text{IF } D \text{ THEN } E \text{ ELSE } X]\!](\sigma)$$

Or :

$$\sigma[X \leftarrow [\![\text{IF } D \text{ THEN } E \text{ ELSE } X]\!](\sigma)] \models Q$$

Donc d'après le corollaire 2-1, on a :

$$\sigma \models Q[\text{IF } D \text{ THEN } E \text{ ELSE } X]$$

Par conséquent :

$$[\![\{Q[\text{IF } D \text{ THEN } E \text{ ELSE } X], D\}]\!] = wp(«X := E», \{Q, D\})$$

Communication : la preuve est similaire à celle de l'affectation.

Composition séquentielle :

$$\begin{array}{rcl} wp\,(«S_1\,;\,S_2», \{Q, D\}) & = & \{(\sigma, s) \mid [\![S_1\,;\,S_2]\!](\sigma, s) \subseteq [\![\{Q, D\}]\!]\,\} \\ & & \quad \text{par définition de la fonction } wp \\ & = & \{(\sigma, s) \mid [\![S_2]\!]\,([\![S_1]\!](\sigma, s)) \subseteq [\![\{Q, D\}]\!]\,\}\} \\ & & \quad \text{d'après la définition de } [\![S_1\,;\,S_2]\!] \end{array}$$

Or, $\{(\sigma, s) \mid [\![S_2]\!]\,([\![S_1]\!](\sigma, s)) \subseteq [\![\{Q, D\}]\!]\,\} =$
$\{(\sigma, s) \mid [\![S_1]\!](\sigma, s) \subseteq \{(\sigma', s') \mid [\![S_2]\!](\sigma', s') \subseteq [\![\{Q, D\}]\!]\,\}\}$

d'où, toujours d'après la définition de la fonction wp,

$$\begin{array}{rcl} wp\,(«S_1\,;\,S_2», \{Q, D\}) & = & \{(\sigma, s) \mid [\![S_1]\!](\sigma, s) \in \{(\sigma', s') \mid [\![S_2]\!](\sigma', s') \subseteq [\![\{Q, D\}]\!]\,\}\} \\ & = & \{(\sigma, s) \mid [\![S_1]\!](\sigma, s) \subseteq wp\,(«S_2», \{Q, D\})\} \\ & = & wp\,(«S_1», wp\,(«S_2», \{Q, D\})) \end{array}$$

Conditionnement : Considérons l'état (σ, s) tel que :

$$\begin{array}{ll} & (\sigma, Push(Top(s) \wedge \sigma(B), s)) \in wp(«S», \{Q, D \wedge B\}) \\ \text{et} & Top(s) = \sigma(D) \end{array}$$

Cela signifie que l'on a d'une part :

$$\forall (\sigma', s') \in [\![S]\!]\,(\sigma, Push(Top(s) \wedge \sigma(B), s))$$

$$\sigma' \models Q$$

$$\sigma'(D \ \wedge \ B) = Top(s) \ \wedge \ \sigma(B)$$
$$\sigma(D) = Top(s)$$

et d'autre part comme $Var(D) \cap Change(S) = \emptyset$ on a $\sigma(D) = \sigma'(D)$, donc :

$$(\sigma', s) \in [\![\{Q, D\}]\!]$$

Il faut de plus vérifier si :

$$(\sigma', s) \in [\![\texttt{where} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma, s)$$

Ce qui est vrai, car :

$$[\![\texttt{where} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma, s) = \{(\sigma', s) \mid \sigma' \in pr_e[\![S]\!](\sigma, Push(Top(s) \wedge \sigma(B), s))\}$$

et que σ' a été pris tel qu'il existe une pile s' où :

$$(\sigma', s') \in [\![S]\!](\sigma, Push(Top(s) \wedge \sigma(B), s))$$

Itération :

1.

$$= \begin{array}{l} wp(\text{«}\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}\text{»}, \{Q, D\}) \cap [\![\{\forall u.(D|_u \Rightarrow \neg B|_u), D\}]\!] \\ \{(\sigma, s) \ \mid \ (\sigma, s) \in [\![\{\forall u.(D|_u \Rightarrow \neg B|_u), D\}]\!] \\ \qquad \wedge \ [\![\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma, s) \subseteq [\![\{Q, D\}]\!]\} \end{array}$$

on a :

$$[\![\{\forall u.(D|_u \Rightarrow \neg B|_u), D\}]\!] \equiv (\forall u.(actif(u) \Rightarrow \neg B|_u))$$

d'où par définition de la sémantique opérationnel du `while` on a :

$$(\sigma, s) \in [\![\{\forall u.(D|_u \Rightarrow \neg B|_u), D\}]\!] \Rightarrow [\![\text{«}\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}\text{»}]\!](\sigma, s) = (\sigma, s)$$

par conséquent :

$$(\sigma, s) \in [\![\{Q, D\}]\!]$$

2.

$$= \begin{array}{l} wp(\text{«}\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}\text{»}, \{Q, D\}) \cap [\![\{\exists u.(D|_u \wedge B|_u), D\}]\!] \\ \{(\sigma, s) \ \mid \ (\sigma, s) \in [\![\{\exists u.(D|_u \wedge B|_u), D\}]\!] \\ \qquad \wedge \ [\![\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma, s) \subseteq [\![\{Q, D\}]\!]\} \end{array}$$

on a :

$$[\![\{\exists u.(D|_u \wedge B|_u), D\}]\!] \equiv (\exists u.actif(u) \Rightarrow B|_u))$$

d'où par définition de la sémantique opérationnel du `while` on a :

$$\forall(\sigma, s) \in [\![\{\exists u.(D|_u \wedge B|_u), D\}]\!]$$

$$\begin{array}{rcl} [\![\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma, s) & = & [\![S \ \texttt{;while} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma, s) \\ & = & [\![\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}]\!]([\![S]\!](\sigma, s)) \end{array}$$

or :

$$\begin{array}{rl} & \{(\sigma, s) \mid [\![\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}]\!]([\![S]\!](\sigma, s)) \subseteq [\![\{Q, D\}]\!]\} \\ = & \{(\sigma, s) \mid [\![S]\!](\sigma, s) \subseteq \{(\sigma', s') \mid [\![\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}]\!](\sigma', s') \subseteq [\![\{Q, D\}]\!]\}\} \\ = & wp(\text{«}S\text{»}, wp(\text{«}\texttt{while} \ B \ \texttt{do} \ S \ \texttt{end}\text{»}, \{Q, D\})) \end{array}$$

□

4.3 Calcul des *verification conditions*

Les résultats obtenus dans la section précédente permettent de calculer effectivement la fonction wp sur un programme $\mathcal{L}$ uniquement lorsque les invariants de boucle sont fournis. En pratique, il est donc nécessaire de modifier la grammaire du langage pour permettre à l'utilisateur d'indiquer explicitement, pour chaque itération, quel est l'invariant pressenti qui lui est associé. On introduit donc la nouvelle instruction `while` B `do` inv $\{I, E\}$ S `end` dans laquelle $\{I, E\}$ dénote une assertion.

Dans la suite, on désignera par $Annoted_Pgm$ l'ensemble des programmes de Pgm dans lesquels les instructions d'itérations sont ainsi annotées, et le résultat de la transformation d'un programme S en un programme annoté sera noté S^*. Notons que la sémantique de ces programmes ne diffère en rien de celle des programmes originaux, et qu'en particulier pour tout programme S on a $[\![S]\!] = [\![S^*]\!]$.

On est désormais en mesure de donner une approximation de la fonction wp qui soit calculable pour un programme annoté. Plus précisément, on définit en fait simultanément deux fonctions pour chaque instruction du langage :

- Une fonction $VCgen$, qui retourne, pour tout programme S^* et pour toute assertion $\{Q, D\}$, un ensemble de propositions de logique du premier ordre, construites sur le langage d'assertions :

  $$VCgen \ : \ Annoted_Pgm \times Assertion \rightarrow 2^{Sform}$$

  Intuitivement, ces propositions seront valides si et seulement si les invariants de S^* sont *corrects* (ce sont bien des invariants), et s'ils sont suffisants (ils permettent bien de déduire la postcondition $\{Q, D\}$).

- Une fonction pre, qui, pour un programme S^* et une assertion $\{Q, D\}$ donnés, permet d'approcher la fonction $wp(S, \{Q, D\})$:

  $$pre \ : \ Annoted_Pgm \times Assertion \rightarrow Assertion$$

  Si les propositions $VCgen(S^*, \{Q, D\})$ sont valides, alors $pre(S^*, \{Q, D\})$ est une précondition pour $(S^*, \{Q, D\})$. On notera pre^e et pre^c les projections respectives de cette fonction sur la partie état ou expression de contexte.

Ces deux fonctions sont explicitées dans la définition suivante.

Définition 4-2

Affectation :

$$\begin{aligned} pre(«X := E», \{Q, D\}) &= (Q\,[\text{IF } D \text{ THEN } E \text{ ELSE } X/X], D) \\ VCgen(«X := E», \{Q, D\}) &= \emptyset \end{aligned}$$

Communication :

$$\begin{aligned} pre(«\texttt{get } Y \texttt{ from } A \texttt{ into } X», \{Q, D\}) &= (Q\,[\text{IF } D \text{ THEN } Y|_A \text{ ELSE } X/X], D) \\ VCgen(«\texttt{get } Y \texttt{ from } A \texttt{ into } X», \{Q, D\}) &= \emptyset \end{aligned}$$

Composition séquentielle :

$$\begin{aligned} pre(«S_1^* \,;\, S_2^*», \{Q, D\}) &= pre(S_1^*, pre(S_2^*, \{Q, D\})) \\ VCgen(«S_1^* \,;\, S_2^*», \{Q, D\}) &= VCgen(S_2^*, \{Q, D\}) \cup VCgen(S_1^*, pre(S_2^*, \{Q, D\})) \end{aligned}$$

Conditionnement :

$$\begin{array}{rcl} pre(\text{«}\texttt{where}\ B\ \texttt{do}\ S^*\ \texttt{end}\text{»}, \{Q, D\}) & = & \{pre^e(S^*, \{Q, D \wedge B\}), D\} \\ VCgen(\text{«}\texttt{where}\ B\ \texttt{do}\ S^*\ \texttt{end}\text{»}, \{Q, D\}) & = & VCgen(S^*, \{Q, D \wedge B\}) \end{array}$$

Itération :

$$\begin{array}{rcl} pre(\text{«}\texttt{while}\ B\ \texttt{do}\ inv\{I, E\}\ S^*\ \texttt{end}\text{»}, \{Q, D\}) & = & \{I, E\} \\ VCgen(\text{«}\texttt{while}\ B\ \texttt{do}\ inv\{I, E\}\ S^*\ \texttt{end}\text{»}, \{Q, D\}) & = & VCgen(S^*, \{I, E\}) \cup Inv \end{array}$$

où Inv représente la conjonction suivante :

$$(\{I \wedge \forall u\,.\,(E|_u \Rightarrow \neg B|_u), E\} \overset{C}{\Rightarrow} \{Q, D\}) \wedge (\{I \wedge \exists u\,.\,(E|_u \wedge B|_u), E\} \overset{C}{\Rightarrow} pre(S^*, \{I, E\}))$$

■

Il reste alors à montrer que ces définitions de pre et $VCgen$ vérifient bien les propriétés énoncées précédemment, ce qui est formalisé dans la proposition 4-2. La preuve de cette proposition reposera sur le lemme suivant qui établit une propriété de monotonie de la fonction wp pour la relation $\overset{C}{\Rightarrow}$:

Lemme 4-1

Soient S un fragment de programme $\mathcal{L}$ et $\{P, C\}$ et $\{Q, D\}$ des assertions. On a alors :

$$(\{P, C\} \overset{C}{\Rightarrow} \{Q, D\}) \Rightarrow wp(\text{«}S\text{»}, \{P, C\}) \subseteq wp(\text{«}S\text{»}, \{Q, D\})$$

■

Preuve : L'hypothèse $\{P, C\} \overset{C}{\Rightarrow} \{Q, D\}$ implique $[\![\{P, C\}]\!] \subseteq [\![\{Q, D\}]\!]$.
Par suite :

$$\begin{array}{rcl} wp(\text{«}S\text{»}, \{P, C\}) & = & \{(\sigma, s) \mid [\![S]\!](\sigma, s) \subseteq [\![\{P, C\}]\!]\} \\ & \subseteq & \{(\sigma, s) \mid [\![S]\!](\sigma, s) \subseteq [\![\{Q, D\}]\!]\} \\ & \subseteq & wp(\text{«}S\text{»}, \{Q, D\}) \end{array}$$

□

Proposition 4-2

Pour tout programme annoté S^*, pour toute assertion $\{Q, D\}$ telle que

$$Change(S^*) \cap Var(D) = \emptyset$$

on a :

$$VCgen(S^*, \{Q, D\}) \Rightarrow [\![pre(S^*, \{Q, D\})]\!] \subseteq wp(\text{«}S\text{»}, \{Q, D\})$$

■

En langage clair, cette proposition signifie que si les invariants sont corrects et suffisants, alors la précondition obtenue par la fonction pre est correcte par rapport à la postcondition $\{Q, D\}$ et au programme S^*.

Preuve : Elle se fait par induction sur la structure de S^*, en examinant chaque cas :

Affectation, communication :

Elle est immédiate puisque les définitions des fonctions pre et wp coïncident.

Composition séquentielle :

Par définition de $VCgen$, on a :

$$VCgen\,(«S_1^*\,;\,S_2^*»,\{Q,D\}) \;\Rightarrow\; (VCgen\,(S_2^*,\{Q,D\}) \;\wedge\; VCgen\,(S_1^*,pre\,(S_2^*,\{Q,D\})))$$

Par hypothèse d'induction sur S_1^* et S_2^*, on en déduit :

$$[\![pre\,(S_2^*,\{Q,D\})]\!] \subseteq wp\,(«S_2»,\{Q,D\})$$

et

$$[\![pre\,(S_1^*,pre\,(S_2^*,\{Q,D\}))]\!] \subseteq wp\,(«S_1»,pre\,(S_2,\{Q,D\}))$$

En appliquant alors le lemme 4-1, on obtient bien :

$$[\![pre\,(S_1^*,pre\,(S_2^*,\{Q,D\}))]\!] \subseteq wp\,(«S_1»,wp\,(«S_2»,\{Q,D\}))$$

Conditionnement :

Par définition de $VCgen$, on a :

$$VCgen\,(«\texttt{where}\;\; B\;\; \texttt{do}\;\; S^*\;\; \texttt{end}»,\{Q,D\}) \;\Rightarrow\; VCgen\,(S^*,\{Q,D\;\wedge\;B\})$$

Par hypothèse d'induction, on en déduit :

$$[\![pre\,(S^*,\{Q,D\;\wedge\;B\})]\!] \subseteq wp\,(«S»,\{Q,\,D\;\wedge\;B\})$$

on a :

$$Var(D\;\wedge\;B)\cap Change(S)=\emptyset$$

donc $pre\,(S^*,\{Q,D\;\wedge\;B\})$ est de la forme $\{P,D\;\wedge\;B\}$ et on a :

$$[\![\{P,D\;\wedge\;B\}]\!] \subseteq wp(«S»,\{Q,\,D\;\wedge\;B\})$$

donc comme :

$$\begin{aligned}&\{(\sigma,s)\mid\;\; (\sigma,Push(Top(s)\;\wedge\;\sigma(B),s))\in wp(«S»,\{Q,D\;\wedge\;B\})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge Top(s)=\sigma(D)\}\\ \subseteq\;\; & wp(«\texttt{where}\;\; B\;\; \texttt{do}\;\; S\;\; \texttt{end}»,\{Q,D\})\end{aligned}$$

on a :

$$\begin{aligned}&\{(\sigma,s)\mid(\sigma,Push(Top(s)\;\wedge\;\sigma(B),s))\in[\![\{P,D\;\wedge\;B\}]\!]\wedge Top(s)=\sigma(D)\}\\ \subseteq\;\; & wp(«\texttt{where}\;\; B\;\; \texttt{do}\;\; S\;\; \texttt{end}»,\{Q,D\})\end{aligned}$$

Par définition on a :

$$\begin{aligned}&\{(\sigma,s)\mid(\sigma,Push(Top(s)\;\wedge\;\sigma(B),s))\in[\![\{P,D\;\wedge\;B\}]\!]\wedge Top(s)=\sigma(D)\}\\ =\;\; & \{(\sigma,s)\mid(\sigma\models P)\wedge(\sigma(D\;\wedge\;B)=Top(s)\;\wedge\;\sigma(B))\wedge(\sigma(D)=Top(s))\}\end{aligned}$$

comme $\sigma(D\;\wedge\;B)=\sigma(D)\;\wedge\;\sigma(B)$ on a donc l'égalité suivante :

$$\begin{aligned}&\{(\sigma,s)\mid(\sigma,Push(Top(s)\;\wedge\;\sigma(B),s))\in[\![\{P,D\;\wedge\;B\}]\!]\wedge Top(s)=\sigma(D)\}\\ =\;\; & \{(\sigma,s)\mid(\sigma\models P)\wedge(\sigma(D)=Top(s))\}\end{aligned}$$

or on a :

$$\{(\sigma,s)\mid(\sigma\models P)\wedge(\sigma(D)=Top(s))\}=[\![\{P,D\}]\!]=[\![\{pre^e(«S^*»,\{Q,D\;\wedge\;B\}),D\}]\!]$$

donc on obtient finalement :

$$[\![\{pre^e(«S^*»,\{Q,D\;\wedge\;B\}),D\}]\!]\subseteq wp(«\texttt{where}\;\; B\;\; \texttt{do}\;\; S\;\; \texttt{end}»,\{Q,D\})$$

Itération :

Par définition de $VCgen$, l'hypothèse

$$VCgen\,(\text{«}\texttt{while}\ B\ \texttt{do}\ inv\,\{I,E\}\ S^*\ \texttt{end}\text{»}, \{Q,D\})$$

implique les trois propositions suivantes :

(i) $VCgen\,(S^*, \{I,E\})$

(ii) $\{I \wedge \forall u\,.\,(E|_u \Rightarrow \neg B|_u),\, E\} \overset{C}{\Rightarrow} \{Q,\, D\}$

(iii) $\{I \wedge \exists u\,.\,(E|_u \wedge B|_u),\, E\} \overset{C}{\Rightarrow} pre\,(S^*, \{I,E\})$

De *(i)*, on déduit par induction sur S^*

$$[\![pre\,(S^*, \{I,E\})]\!] \subseteq wp\,(\text{«}S\text{»}, \{I,E\})$$

donc, par transitivité, *(iii)* devient :

$$[\![\{I \wedge \exists u\,.\,(E|_u \wedge B|_u),\, E\}]\!] \subseteq wp\,(\text{«}S\text{»}, \{I,E\})$$

D'autre part :

$$[\![\{I,E\}]\!] \subseteq wp(\text{«}\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\text{»}, \{Q,D\})$$

revient à montrer que :

$$\forall(\sigma,s) \in [\![\{I,E\}]\!] \quad \text{soit} \quad [\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}]\!]\,(\sigma,s) \subseteq [\![\{Q,D\}]\!]$$

Par l'absurde, supposons :

$$\exists(\sigma,s) \in [\![\{I,E\}]\!] \quad \text{tel que} \quad [\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}]\!]\,(\sigma,s) \not\subseteq [\![\{Q,D\}]\!]$$

En particulier, comme nous nous intéressons seulement à la correction partielle des programmes, c.-à-d. en ne considérant que les propriétés obtenues à partir d'état qui ne provoquent pas la divergence des programmes, (σ,s) sera tel que :

$$[\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}]\!]\,(\sigma,s) \neq \emptyset$$

Soit (σ',s') tel que :

$$(\sigma',s') \in [\![\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}]\!]\,(\sigma,s)$$

d'après la sémantique opérationnelle du `while`, et de par la conséquence des propriétés *(i)* et *(iii)* énoncées auparavant on a $(\sigma',s') \in [\![\{I,E\}]\!]$.

D'autre part, toujours d'après la sémantique opérationnelle du `while` on a :

$$\forall u.actif(u) \Rightarrow \neg\sigma'(B)|_u$$

c.-à-d. :

$$(\sigma',s') \in [\![\{I \wedge \forall u\,.\,(E|_u \Rightarrow \neg B|_u),\, E\}]\!]$$

Donc affirmer que $(\sigma',s') \notin [\![\{Q,D\}]\!]$ est en contradiction avec la proposition *(ii)*.

□

On déduit de cette proposition le corollaire suivant, sur lequel repose en fait l'ensemble de la méthode de vérification que l'on a proposée. Plus précisément, ce corollaire fournit une condition suffisante, qui est décidable, et qui permet de déterminer si un triplet $\{P, C\}\ S\ \{Q, D\}$ donné peut être déduit du système de preuve. Par suite, l'évaluation des fonctions pre et $VCgen$ sur le couple $(S^*, \{Q, D\})$ suffit en pratique pour confirmer que le programme S satisfait bien ses spécifications.

Corollaire 4-1

Pour tout programme annoté S^*, et pour toutes assertions $\{P, C\}$ et $\{Q, D\}$ tels que :

$$Change(S^*) \cap Var(D) = \emptyset$$

on a :

$$(VCgen(S^*, \{Q, D\}) \wedge (\{P, C\} \overset{C}{\Rightarrow} pre(S^*, \{Q, D\}))) \Rightarrow \models \{P, C\}\ S\ \{Q, D\}$$

■

Preuve : D'après la proposition 4-2, on a :

$$VCgen(S^*, \{Q, D\}) \Rightarrow ([\![pre(S^*, \{Q, D\})]\!] \subseteq wp(«S», \{Q, D\}))$$

Par définition, $\{P, C\} \overset{C}{\Rightarrow} pre(S^*, \{Q, D\})$ implique :

$$[\![\{P, C\}]\!] \subseteq [\![pre(S^*, \{Q, D\})]\!]$$

donc par transitivité :

$$[\![\{P, C\}]\!] \subseteq wp(«S», \{Q, D\})$$

par définition même de la fonction wp, on obtient bien :

$$\models \{P, C\}\ S\ \{Q, D\}$$

□

Ce dernier corollaire exprime la *correction* de la méthode de *verification condition* que nous avons définie pour le langage $\mathcal{L}$. C'est à dire que si nous validons un triplet $\{P, C\}\ S\ \{Q, D\}$ par cette méthode, alors toutes exécutions terminées de S à partir d'un état vérifiant $\{P, C\}$, termineront dans un état vérifiant $\{Q, D\}$. Nous allons voir maintenant une mécanisation possible de cette méthode dans l'atelier logiciel CENTAUR.

5 Un environnement d'aide à la preuve automatique

Outre la définition d'une méthode de preuve de programmes qui puisse être automatisable, l'un des objectifs de notre travail était également de pouvoir la valider d'un point de vue pratique en proposant une implémentation au sein d'un outil. Comme on l'a vu dans la section 4.1, l'approche que nous avons retenue est une approche semi-automatique, qui nécessite la réunion de plusieurs composants logiciels avec lesquels l'utilisateur doit interagir. Par conséquent, le prototype qui a été développé doit être considéré plus comme un *environnement d'aide à la preuve de programmes* $\mathcal{L}$, que comme un outil de vérification automatique, capable de décider sans aide extérieure si un programme est correct ou non.

On présente tout d'abord l'architecture générale de cet environnement, en décrivant les interactions entre ses différents composants. On précise alors comment ces composants peuvent être mis en œuvre à l'aide des outils logiciels CENTAUR et HOL. Enfin, on termine en indiquant brièvement les perspectives d'évolution de notre environnement, et notamment les principales fonctionnalités qui resteraient à implémenter.

5.1 Architecture générale

On cherche ici à proposer une architecture pour un environnement logiciel qui puisse implémenter la méthode de preuve décrite dans la section 4.1. Un tel environnement doit donc inclure un certain nombre de fonctionnalités :

- faciliter l'écriture et la modification d'un programme et de ses spécifications, par exemple à l'aide d'un éditeur syntaxique ;
- apporter une aide pour la recherche et l'écriture des invariants du programme ;
- implémenter le calcul des plus faibles préconditions et des *verification conditions* associées à un programme annoté (c.-à-d. les fonctions pre et $VCgen$ décrites dans la section 4) ;
- permettre la preuve des *verification conditions*, en donnant notamment accès à des procédures de décision dans la logique du premier ordre ;
- enfin, offrir des primitives de gestion de la preuve du programme, en autorisant par exemple une preuve modulaire, ou encore la réutilisation au cours d'une session d'énoncés déjà prouvés.

Etant donné qu'il n'était pas souhaitable de développer intégralement un environnement de ce type, nous avons choisi d'articuler notre prototype autour d'un «générateur d'environnement», le logiciel CENTAUR. Le principe de ce générateur est de proposer un certain nombre de *meta-langages* qui permettent de spécifier des outils relatifs à la syntaxe et à la sémantique d'un langage de programmation, ainsi que des primitives de haut niveau qui permettent de définir une interface graphique pour ces outils. Néanmoins, les meta-langages existant sous CENTAUR ne permettant pas de construire facilement un outil de preuve dans la logique du premier ordre, il a également été nécessaire de faire appel à un *theorem prover* externe, le logiciel HOL, pour la preuve des *verification conditions*.

La figure 5.1 décrit de façon plus précise comment CENTAUR et HOL ont été utilisés, en indiquant quelles sont les fonctionnalités qu'ils implémentent.

La partie CENTAUR de cette application a été construite à partir d'un environnement de programmation pour le langage $\mathcal{L}$ développé par Levaire [19]. Outre un éditeur syntaxique, cet environnement comprend un *interpréteur* de programme, auquel sont associés des outils graphiques d'aide à la mise au point, qui permettent entre autre d'examiner le contenu des variables, ou encore l'activité des processeurs à tout instant de l'exécution. Deux principales fonctionnalités ont donc été ajoutées à cet environnement initial :

- l'éditeur et l'interpréteur ont été étendus pour pouvoir prendre en compte des assertions intermédiaires incluses dans un programme, fournissant ainsi une aide à la recherche des invariants ;
- un générateur de *verification conditions*.

Notons que l'interface graphique offerte par CENTAUR permet également à l'utilisateur de modifier une assertion ou un fragment de programme à tout moment de la preuve, si les résultats obtenus le nécessitent. Enfin, les communications entre CENTAUR et HOL sont envisagées à l'aide d'une interface entre ces deux outils développée à l'INRIA [22]. L'origine de cette connexion entre les deux outils était d'utiliser CENTAUR comme interface graphique pour HOL, ce qui la rend tout à fait adaptée à l'utilisation qui en est attendue ici.

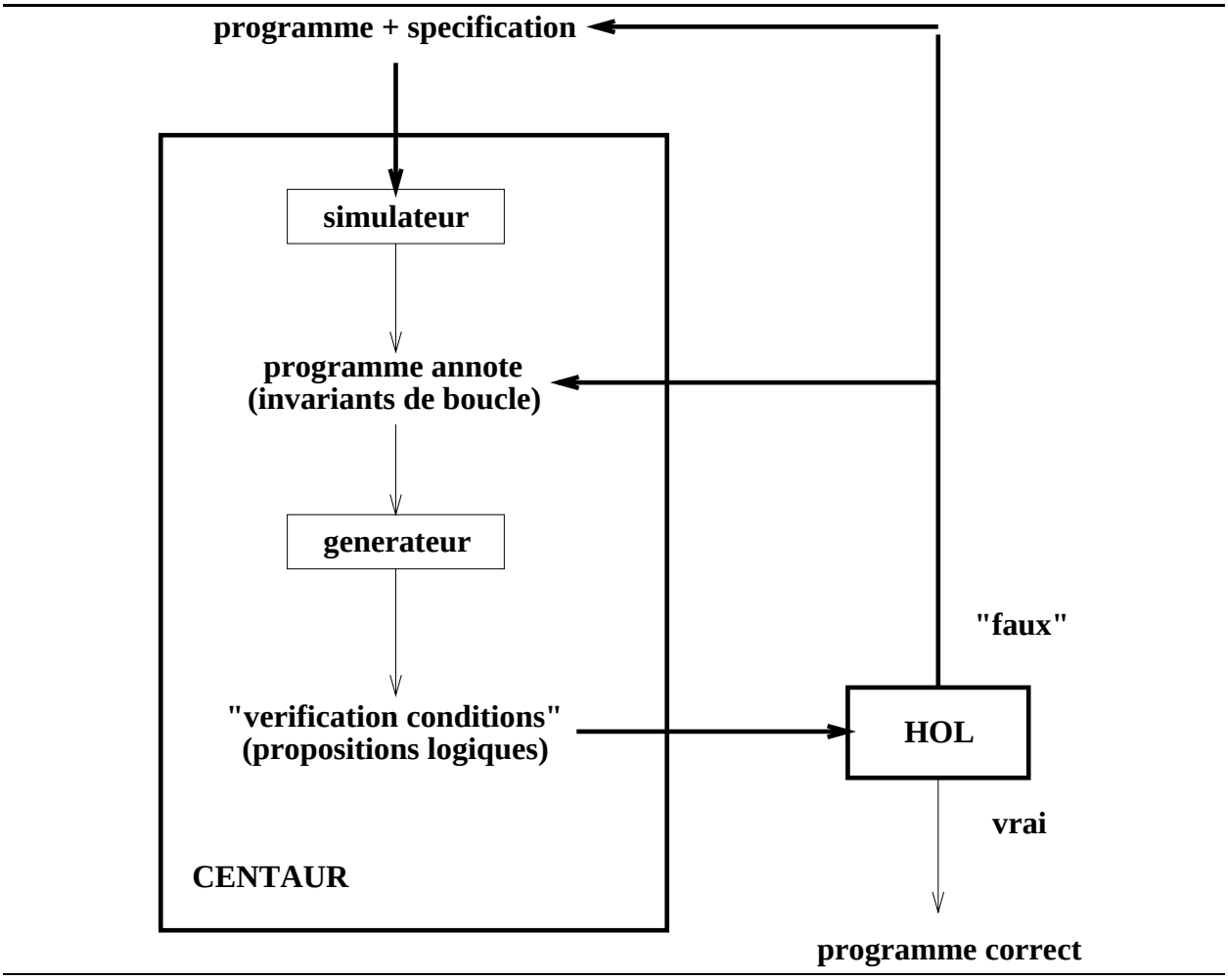

FIG. 12 - Architecture de l'environnement d'aide à la preuve

En conclusion, cette architecture présente deux caractéristiques principales qui nous paraissent importantes en pratique.

- Elle est modulaire, et en particulier les appels au *theorem prover* restent limités et très ciblés. Par suite, cet outil peut être vu comme un composant auxiliaire, et son choix peut donc être facilement remis en cause, indépendamment du reste de l'architecture.

- Toute interaction avec l'utilisateur se fait par l'intermédiaire de CENTAUR, ce qui présente l'avantage d'offrir une certaine homogénéité. En outre, le *theorem prover* reste ainsi transparent du point de vue de l'utilisateur.

5.2 Mise en œuvre sous Centaur

Le générateur d'environnement CENTAUR a été développé dans l'équipe de Kahn à l'INRIA [2]. L'un de ses objectifs est d'offrir une plate-forme pour l'étude et la conception des langages de programmation, et, à ce titre, il a été utilisé dans de nombreux projets. On rappelle tout d'abord les principales fonctionnalités de CENTAUR, puis on décrit brièvement les implémentations de l'interpréteur et du générateur de *verification conditions* réalisé à l'aide de ce système.

Sous CENTAUR, Le développement d'un environnement de programmation pour un langage donné s'effectue selon trois principaux axes.

Syntaxe du langage

Elle est décrite au moyen d'un méta-langage (METAL), qui permet de définir deux grammaires dédiées respectivement à la syntaxe abstraite et à la syntaxe concrète du langage. A partir de cette description, CENTAUR génère automatiquement un analyseur syntaxique. Par ailleurs, un second meta-langage, PPML, permet d'en spécifier un *pretty-printer*, qui, associé à l'analyseur, fournit un éditeur syntaxique pour le langage considéré. Le modèle de base utilisé au sein de CENTAUR pour représenter un programme (ou un fragment de programme) est un *arbre abstrait*. De nombreuses primitives sont donc également offertes pour manipuler les arbres abstraits.

Sémantique du langage

Les aspects sémantiques du langage peuvent être spécifiés à l'aide d'un troisième méta-langage, TYPOL, qui est basé sur la notion de *sémantique naturelle*. Concrètement, il permet de définir un ensemble d'axiomes et de règles d'inférences sur les termes d'un programme. Cette spécification est ensuite compilée en un ensemble de clauses PROLOG qui portent sur l'ensemble des arbres abstraits du langage. Par suite, TYPOL permet de définir différents outils relatifs à la sémantique du langage, et notamment des interpréteurs.

Interface utilisateur

Outre les formalismes directement liés à la description d'outils dédiés à la syntaxe et à la sémantique d'un langage, CENTAUR offre également un certain nombre de primitives de haut niveau qui permettent de leur associer une interface graphique. Par suite, dans l'environnement de programmation qui résulte de la combinaison de ces outils, les interactions avec l'utilisateur se font essentiellement à l'aide de menus et fenêtres.

Comme on l'a vu dans la section 3, une technique de recherche des invariants consiste à examiner le comportement du programme sur une exécution donnée. Pour faciliter cette recherche, nous avons étendu l'interpréteur développé par Levaire afin qu'il puisse prendre en compte des assertions intermédiaires insérées dans le programme. Plus précisément, lors de l'exécution d'un programme

annoté, pour chaque assertion $\{P, C\}$ rencontrée l'expression $[\![\{P, C\}]\!](\sigma, s)$ est évaluée en fonction de l'environnement courant σ et du contexte courant s, et le résultat de cette évaluation est retourné à l'utilisateur. Lorsque l'assertion s'avère invalide, un diagnostic plus précis est produit dans lequel il est précisé s'il s'agit de la partie contexte ou de la partie état de l'assertion qui est erronée, et, dans ce dernier cas, le sous-terme qui invalide l'assertion est exhibé. Ainsi, l'utilisateur a la possibilité dans un premier temps de valider rapidement, sur une exécution donnée, les invariants et les spécifications associés au programme, sans avoir à se lancer directement dans une preuve axiomatique complète, nécessairement plus lourde. Le cas échéant, l'interface utilisateur de CENTAUR permet de corriger facilement une assertion avant de ré-itérer le processus. Enfin, notons que la spécification TYPOL d'un tel interpréteur se déduit directement de la définition de la sémantique du langage et de celle des assertions.

Une fois que la spécification et les invariants ont été validés sur une exécution donnée, une preuve exhaustive du programme est alors envisageable. Compte tenu de la méthode de vérification présentée dans la section 4.1, la première étape consiste à construire les *verification conditions* associées au programme annoté. Nous avons donc implémenté en TYPOL un générateur dont les fonctionnalités sont les suivantes.

- Sélection d'un triplet $\{P, C\}\ S^*\ \{Q, D\}$ au sein du programme annoté initial : il peut s'agir du programme complet ou d'un fragment de programme encadré par deux assertions intermédiaires. Dans tous les cas, il est vérifié que les invariants sont bien présents et que le programme est sous une forme qui satisfait les contraintes décrites dans la section 1.3.

- Calcul de la plus faible précondition et des *verification conditions* : les calculs des fonctions pre et $VCgen$ ont lieu simultanément, les résultats étant retournés dans deux fenêtres distinctes. Plus précisément, la postcondition $\{Q, D\}$ est «remontée» à travers S^* (fonction pre), et les *verification conditions* sont générées «à la volée» (fonction $VCgen$) au fur et à mesure du calcul. On exprime ainsi que chaque invariant rencontré est correct et suffisant, et que la précondition $\{P, C\}$ est elle aussi suffisante.

Concrètement, le calcul de la fonction pre fait appel à un mécanisme de substitution d'une expression dans une assertion (axiomes de l'affectation et de communication). Dans le cas de l'axiome de communication, il est également nécessaire de «normaliser» l'assertion au préalable, de sorte que la variable X à substituer apparaisse dans un sous-terme de la forme «$X|_s$», où s est une expression scalaire. Ces deux primitives (substitution et normalisation) ont été implémentées directement en PROLOG.

5.3 Mise en œuvre sous Hol

Le système HOL (High Order Logic) est un *theorem prover* de type général, développé à l'université de Cambridge [12], qui repose sur la logique d'ordre supérieur. Différentes stratégies de preuve peuvent êtres combinées sous HOL, parmi lesquelles la preuve «en arrière» (*backward proof*) s'avère souvent la plus efficace en pratique. Elle consiste à décomposer successivement le théorème à prouver en sous-buts, tels que chaque décomposition soit justifiée par l'application d'une règle d'inférence du système de preuve qui modélise la logique de HOL (le moteur de preuve du système). La preuve se termine alors quand tous les sous-buts courants sont soit des axiomes de ce système de preuve, soit des instances de théorèmes déjà prouvés. Pour mettre en œuvre ces décompositions, le système HOL fournit une collection de «tactiques», qui peuvent être vues comme des suites de décomposition élémentaires, offrant ainsi des primitives de plus haut niveau pour effectuer des preuves par induction, par réécriture, ou encore par «analyse de cas». Enfin, ces tactiques peuvent elles-mêmes

être combinées à l'aide de *tacticals*, qui sont des opérateurs sur les tactiques, et qui permettent de programmer des outils de preuve assez puissants, aptes à vérifier automatiquement des classes de théorèmes similaires. Enfin, une librairie importante de théorèmes déjà prouvés, regroupés en théories, augmente encore les capacités du système.

Concrètement, HOL est construit à partir du langage ML, qui est également disponible au niveau de l'utilisateur pour toutes les interactions avec le système : preuve de théorèmes, définition de nouvelles tactiques, extension d'une théorie existante. Par ailleurs, les caractéristiques de ce langage, et notamment le contrôle strict du type des expressions manipulées, garantissent que le système HOL est *sûr* : tout théorème prouvé est obtenu par une suite d'applications des 8 règles d'inférence et des 5 axiomes qui constituent le moteur de preuve de HOL.

Un certain nombre d'arguments ont motivés le choix de ce *prover* dans notre environnement pour implémenter la preuve des *verification conditions*.

- En premier lieu, il s'agit d'un système relativement ancien, autour duquel une importante communauté existe. Par suite, il a été largement utilisé, notamment pour la preuve de circuits, et une large bibliothèque d'exemples et d'outils lui est associée, ainsi qu'une documentation conséquente.

- D' autre part, il existe un nombre assez important de théorèmes relatifs à l'arithmétique de Peano qui sont déjà prouvés au sein de ce système, ce qui le rend assez performant pour effectuer des preuves dans ce domaine (ce qui n'est pas le cas de tous les *theorem provers*).

- Enfin, une interface entre HOL et CENTAUR ayant déjà été réalisée, il nous a paru préférable de choisir ce système, et économiser ainsi l'effort de développer intégralement une nouvelle interface.

Notons également que s'il s'avérait en pratique que HOL soit peu adapté à la preuve des *verification conditions*, l'architecture modulaire de notre environnement offre la possibilité de remplacer facilement ce composant par un autre *theorem prover*.

On peut citer deux principales expériences menées avec HOL dans le domaine de la preuve axiomatique de programmes :

- Gordon [13] a de'veloppé un outil de preuve pour un langage séquentiel simple. L'approche qu'il a suivie est tout à fait similaire à celle décrite dans ce travail, et elle repose donc également sur un calcul de plus faibles préconditions et sur la génération et la preuve de *verification conditions*. La différence avec notre implémentation est que ces trois étapes sont mises en œuvre uniquement à l'aide de HOL. Par suite, l'utilisation de cet outil de vérification reste très délicate pour un utilisateur non expert en HOL. Néanmoins, l'intérêt majeur de cette approche est sa fiabilité, puisque l'ensemble de la preuve du programme peut se ramener à une preuve dans le système de preuve de HOL : ni nouveaux axiomes, ni nouvelles règles d'inférence ne sont introduits.

- Harrison [17] a utilisé HOL pour la preuve axiomatique de programmes distribués. Toutefois, étant donné la sémantique assez complexe du langage considéré, il ne lui a pas été possible de définir de façon suffisamment simple des fonctions de calcul de préconditions et de génération de *verification conditions*. Il a donc dû opter pour une approche différente, qui consiste à implémenter directement la sémantique axiomatique de son langage sous HOL, introduisant ainsi de nouveaux schémas d'axiomes et de nouvelles règles d'inférence. Enfin, des fonctions ML sont également proposées à l'utilisateur pour faciliter la génération de preuves directement dans ce système.

L'utilisation de HOL que l'on envisage ici est très proche du module dédiée à la preuve des *verification conditions* de l'outil développé par Gordon : seuls les langages sur lesquels sont définis les propositions logiques diffèrent. Elle nous paraît par conséquent assez réaliste en pratique.

5.4 Discussion

A ce jour, seule la partie «CENTAUR» de l'environnement a été réalisée. La version du langage $\mathcal{L}$ considérée est essentiellement celle décrite dans ce rapport, dans laquelle les variables parallèles considérées sont des vecteurs d'entiers de dimension 2 (tableaux). Le langage d'assertion a également été étendu en ajoutant certains opérateurs de réduction d'une variable parallèle en une expression scalaire (somme et produit de ses éléments). A partir de cette implémentation, les *verification conditions* ont pu être générées pour un certain nombre d'exemples de programmes non triviaux, notamment le programme de *scan* présenté dans la section 3. Dans tous les cas, la correction des *verification conditions* engendrées a pu être établie à la main.

La suite de ce travail d'implémentation porte donc sur différents points :

- l'implémentation sous HOL de la preuve des *verification conditions*, en s'inspirant de l'outil réalisé par Gordon ;
- la mise en œuvre de l'interface existant entre CENTAUR et HOL au sein de cet environnement, ce qui signifie en fait développer un outil de traduction dans la syntaxe HOL des *vérification conditions* générées sous CENTAUR ;
- la réalisation de fonctions d'aide à la gestion de la preuve, permettant notamment de mémoriser au niveau de CENTAUR les triplets déjà prouvés depuis le début de la session.

6 Conclusions

Ce travail se situe dans le cadre de la vérification formelle de programmes data-parallèles, et il s'articule autour d'un langage à la fois simple et expressif, le langage $\mathcal{L}$. Plus précisément, à partir d'une sémantique axiomatique de ce langage, notre objectif était de proposer une méthode de preuve de programmes qui soit automatisable, et qui puisse être étendue par la suite à des langages réels. Concrètement, ce travail a donc été mené selon deux axes.

Dans un premier temps, nous avons d'abord effectué «à la main» la preuve axiomatique d'un certain nombre de programmes data-parallèles écrits en langage $\mathcal{L}$, certains classiques, comme le *scan* (section 3.1), d'autres plus originaux, comme la construction de *quad-tree* (section 3.2). Cette étape était importante pour différentes raisons.

- En premier lieu, elle a permis de montrer que l'approche axiomatique était tout à fait envisageable dans le cas d'un modèle data-parallèle, et qu'elle pouvait également fournir une aide à la conception de programmes. En particulier, il s'est avéré sur ces exemples que l'écriture des invariants, qui est généralement une des difficultés de cette approche, restait d'une complexité raisonnable pour le programmeur.
- D'autre part, elle a également permis de valider le langage d'assertion qui accompagne la sémantique axiomatique de $\mathcal{L}$, notamment du point de vue de son expressivité. En effet, les propriétés à valider des différents programmes que l'on a considérés ont toujours pu être exprimées sous forme d'assertions.

- Enfin, elle a permis de définir plus précisément quelles étaient les différentes étapes de la preuve de programme qu'il était important d'automatiser, et quelles étaient celles qui pouvaient éventuellement rester à la charge du programmeur. Cette information a été utile lors de la spécification des outils d'aide à la preuve.

La seconde étape de ce travail a été de définir en détail un environnement d'aide à la preuve de programme $\mathcal{L}$. Nous avons donc tout d'abord proposé une méthode pour automatiser la génération de preuves dans le système axiomatique associé à ce langage. Cette méthode, qui est valide pour des programmes normalisés au préalable, a été justifiée formellement à partir de la correction du système de preuve. Enfin, cet environnement a été partiellement implémenté à l'aide des logiciels CENTAUR et HOL, et a été expérimenté sur quelques exemples.

En conclusion, la démarche que nous proposons ici pour la preuve de programmes data-parallèle nous semble réaliste, et il paraît clair que ce modèle de programmation, proche du modèle séquentiel, se prête bien aux méthodes de vérification de type axiomatique. En particulier, de nombreuses techniques issues du domaine séquentiel (comme la génération de *verification conditions*) peuvent lui être étendues.

Toutefois, un certain nombre de perspectives restent à envisager pour poursuivre ce travail :

- En ce qui concerne la sémantique axiomatique de $\mathcal{L}$, la preuve de la correction et éventuellement de la complétude du système de preuve semble désormais envisageable. Par ailleurs, l'application de cette méthode de preuve aux langages réels nécessite d'étendre la sémantique axiomatique de $\mathcal{L}$ aux constructions qui n'ont pas été envisagées dans ce rapport, comme les instructions *escape* ou *all*. Enfin, le système de preuve pourrait également être étendu pour prendre en compte la terminaison des programmes.

- De même que la sémantique axiomatique, la méthode de vérification automatique qui a été proposée dans ce rapport devra être étendue aux autres constructions du langage. Ainsi, les fonction pre et $VCgen$ devront être définies pour les instructions de type `everywhere` et `escape`. Par ailleurs, une deuxième amélioration possible consisterait à lever la restriction imposée sur les programmes à vérifier (section 1.3). Concrètement, ceci pourrait être réalisé soit en automatisant la normalisation des programmes, soit en l'intégrant au sein du système de preuve (ce qui est l'approche suivie dans [16]).

- Enfin, concernant la réalisation de l'environnement de programmation différentes perspectives ont été évoquées dans la section 5.4. Il paraît toutefois clair que, du point de vue de l'efficacité, la partie critique de l'implémentation se situe au niveau de la preuve des *verification conditions*. Pour ce point particulier, et conformément à ce qui existe dans le domaine de la preuve de circuits, le compromis entre utiliser un *theorem prover* de type général, et développer un outil de preuve spécifique resterait à étudier. L'utilisation de cet environnement à la vérification de nouveaux exemples de programme $\mathcal{L}$ devrait apporter des éléments de réponse.

A terme, l'ensemble de ces perspectives devrait permettre d'envisager d'appliquer cette méthode de preuve pour la vérification de programmes réels, écrits dans des langages comme C^*, MPL, ou POMPC.

Références

[1] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs*. Text and Monographs in Computer Science. Springer Verlag, 1990.

[2] P. Borras, D. Clement, T. Despeyroux, and J. Incerpi. Centaur : the system. Research Report RR 777, INRIA, 1987.

[3] M. Becerril. Implémentation en langage parallèle POMPC d'algorithmes de segmentation coopérative contour/région. Rapport de DEA ENSTA, ETCA/CREA-SP, 1992.

[4] L. Bougé and J-L. Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *FGCS*, 8:363–378, 1992.

[5] P. Bonnin. Méthode systématique de conception et de réalisation d'applications en vision par ordinateur. Thèse de doctorat de l'université de Paris 7, ETCA/CREA-SP, 1991.

[6] L. Bougé. On the semantics of languages for massively parallel architectures. Research Report 90-06, LIENS, ENS Ulm, Paris, 1990.

[7] L. Bougé. The Data-Parallel Programming Model : a Semantic Perspective. Research Report 92-45, LIP, ENS Lyon, 1992.

[8] S.A. Cook. Soundness and Completeness of an Axiom System for Program Verification. *SIAM J. COMPUT.*, 7(1), February 1978.

[9] P. Cousot. Methods and logics for proving programs. In Jan Van Leeuwen, editor, *Formal Models And Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 842–993. Elsevier, 1990.

[10] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[11] J. Gabarró and R. Gavaldà. An approach to correctness of data parallel algorithms. Technical Report, Univ. Politécnica de Catalunya, Oct. 1989.

[12] M.J.C. Gordon. HOL : A proof generating system for higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer Academic Press, 1988.

[13] M.J.C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving.* Kluwer Academic Press, 1988.

[14] M.J.C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall International, 1989.

[15] D. Gries. *The Science of Programming.* Text and Monographs in Computer Science. Springer Verlag, 1987.

[16] Y. Le Guyadec and B. Virot. Sémantique axiomatique et automatisation de la preuve des programmes data-parallèles. Research Report 93-2, LIFO, Université d'Orléans, 1993.

[17] W.L. Harrison, K.N. Levitt, and M. Archer. A HOL mechanization of the axiomatic semantics of a simple distributed programming language. In *Proc. of the HOL'92 Conference Meeting*, 1992.

[18] High Performance Fortran Forum. High Performance Fortran language specification (draft version). Version 1.0 Draft, CITI/CRPC. Rice Univ., Houston, Jan. 25, 1993.

[19] J.L. Levaire. Using the centaur system to design SIMD parallel languages and programming environments. In *Proc. of the ESOP'92 Conference*, 1992.

[20] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[21] A. Stewart. An axiomatic treatment of SIMD assignment. *Bit*, 30:70–82, 1990.

[22] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. Research Report RR 1684, INRIA, 1992.

[23] G. Utard. Un système axiomatique pour les langages massivement parallèle SIMD. Rapport de DEA, LIP, ENS Lyon, 1991.