Université Paul Sabatier
Master Informatique et Télécommunication
Nom du Laboratoire : IRIT-CNRS
Directeur : Luis Fariñas Del Cerro

Vérification de transformations de modèles

Mounira KEZADRI
Mars-Juin 2009

Directeur de Recherche : Mamoun Filali
Responsables du stage : Jean Paul Bodeveix et Martin Strecker

Résumé :
Ce rapport récapitule le processus suivi dans le cadre du stage pour la conception des transformations de modèles d'une abstraction du langage `Synoptic`. La formalisation de ces transformations en `Ocaml` et dans l'assistant de preuve `Coq` pour permettre par la suite la preuve de terminaison et la vérification de la préservation de propriétés sémantiques par ces transformations.

Table des matières

Chapitre 1

Introduction

Ce travail a pour but de spécifier et vérifier des transformations de modèles sur la syntaxe textuelle d'un langage synchrone. Le rapport est structuré en cinq parties. La définition d'une abstraction du langage `Synoptic`[1] permettant la vérification, l'expression de la sémantique du sous langage identifié, l'identification des différentes transformations et leur formalisation en langage `OCaml` [7], le passage au langage `Java`[2] dans un but comparatif et enfin la formalisation dans l'assistant de preuve `Coq` [11] et la vérification formelle de propriétés sémantiques.

1.1 Contexte

1.1.1 Systèmes réactifs

Les systèmes réactifs [2] sont des programmes qui ne sont pas uniquement descriptibles par des relations transformationnelles spécifiant des sorties à partir d'entrées, mais plutôt par des liens entre les sorties et entrées via leurs combinaisons possibles dans le temps. Ils sont les systèmes interagissant continuellement avec leur environnement. Des formalismes basés sur des approches synchrones ont été proposés pour modéliser le comportement des systèmes réactifs, parmi ces formalismes : StateCharts [5], Esterel et les SyncCharts [12], Lustre et Scade [4], Signal [6]...etc.

1.1.2 Approche synchrone

L'hypothèse synchrone définit une échelle de temps logique discret, constituée d'instants correspondant à chacune des réactions du système. Les événements ayant déclenché la réaction sont considérés comme simultanés. De plus, les réactions se font en temps nul : les signaux émis pendant une réaction sont donc simultanés avec les signaux qui ont provoqué la réaction (la production de la réponse a lieu au même instant logique). Une réaction est donc par construction instantanée, ce qui évite les réactions concurrentes partielles d'un système, sources d'indéterminisme. L'utilisation de l'approche synchrone (les sorties sont instantanées avec les entrées) garantit :

- **Le parallélisme** Le comportement global est exprimé comme étant la composition de comportement de processus distincts qui communiquent via des signaux.

[1] Synoptic est le langage de description du projet Spacify
[2] http ://java.sun.com/

– **Le déterminisme** Tous les processus ou threads qui testent en cours d'une réaction un signal verront exactement le même statut de celui-ci.

L'objectif des langages synchrones [1] est de permettre une programmation avec une sémantique mathématiquement formulée de systèmes réactifs souvent critiques[3]. Pour cela, ces langages ont adopté une approche de haut niveau qui s'appuie sur des modèles mathématiques simples, suffisamment expressifs et sur lesquels des analyses et des vérifications de propriétés peuvent être effectuées.

1.2 Problématique

Le stage effectué s'inscrit dans le cadre d'un travail en cours dans l'équipe `ACADIE`[4] sur la certification d'un compilateur `Synoptic`. `Synoptic` était défini dans le cadre du projet `Spacify`[5] qui a pour but de formaliser et d'outiller la démarche de conception de logiciels de vol pour le spatial et vise donc à définir un atelier pour le développement et l'amélioration de la qualité des fonctionnalités des logiciels de vol (LV), tout en abaissant leurs coûts et durées de développement.

Un des axes du projet est la certifiabilité par la preuve des analyses et les transformations successives opérées sur les spécifications du logiciel. Ceci est dans le but d'élaborer une librairie de transformations dont les propriétés sont prouvées. Le projet vise à appliquer le paradigme du développement correct par construction qu'il faut opposer aux méthodes basées sur une vérification a posteriori. Cette méthode de développement consiste à partir d'un modèle abstrait exprimant le problème à résoudre et à appliquer par étapes successives des opérateurs de transformation de modèles jusqu'à obtenir le modèle voulu.

La bibliothèque de transformations comportera plusieurs classes de transformations, dans le cadre de ce travail nous traiterons des transformations intra-modèle. Le but sera de prouver que ces transformations préservent la sémantique d'un modèle ou assurent certaines propriétés d'un modèle. L'objectif du stage est de décrire la sémantique d'une sous partie du langage, de spécifier des transformations sur le langage et la preuve de préservation de la sémantique dans l'outil `Coq`.

Dans un premier temps, on vise une description formelle des fonctions de transformations. Dans un second temps, nous aborderons la question de validation de transformation (la preuve de terminaison et la préservation de la sémantique).

[3] Un système critique est un système dont une panne peut avoir des conséquences dramatiques, tels des morts ou des blessés graves, des dommages ...

[4] Assistance à la Certification d'Applications DIstribuées et Embarquées

[5] http ://spacify.gforge.enseeiht.fr/index.php

Chapitre 2

Définition d'une abstraction du langage

La première étape est la spécification en `Ocaml`[1] d'une partie du langage `Synoptic`. Le sous langage identifié doit être : *Expressif* (capable de modéliser le comportement d'un système en exprimant au moins ses fonctionnalités de base) et *Minimal* pour permettre la vérification.

2.1 Syntaxe du langage identifié

Nous allons appeler notre langage résultant le langage de `Bloc`, son noyau élémentaire est le type `bloc`.

$$
\begin{array}{lcl}
B & ::= & B_1 \mid\mid B_2 \\
 & | & \text{var } x_1 x_2 ... x_n.B \\
 & | & b
\end{array}
$$

Le type `Ocaml` correspondant est le suivant :

```
type bloc =
   BlocES of name * int * var list        (*Appel de fonction*)
 | BlocESV of varl list * bloc            (*declaration de variables locales*)
 | Parallele of bloc * bloc;;             (*composition de blocs*)
```

Avec :

```
type name = string;;
type var = string;;
```

Une déclaration d'un `bloc` correspond à définir son nom, son interface ainsi que l'ensemble de ses opérations. La définition d'un `bloc` avec un comportement inconnu est une boite noire décrivant juste son nom et son interface (ses points d'interactions avec les autres composants).

```
type definitionBloc=
   Fonction of name * var list * bloc
 | BoiteNoire of name * var list;;
```

L'environnement est défini comme une liste de `definitionBloc` A.2.

[1] http ://caml.inria.fr/

2.2 Vérifications sur le langage

Un ensemble de vérifications est à effectuer sur notre langage.

2.2.1 Vérification que les variables utilisées dans un bloc sont déclarées

La fonction `verifDef` vérifie que toutes les variables libres utilisées sont déclarées dans l'interface de la fonction.

```
(*Verifier que les variables libres sont declarees*)
 let rec verifDef =function
      Fonction(n,v,b) -> inclus (Par.freeVars b) v
    |_ -> true;;(*cas d'une boite noire*)
```

Le module `Par` A.5 contient la définition de la fonction `freeVars` qui extrait les variables libres d'un bloc.

2.2.2 Vérification de l'arité du bloc dans l'environnement

Pour chaque déclaration d'un nouveau `bloc`, nous vérifions si sa déclaration est conforme à sa définition dans l'environnement. La fonction `verifArite` est définie comme suit :

```
(*Verifier l'arite du bloc dans l'environnement*)
let rec verifArite env = function
    BlocES(a,id,l) -> getArite env a=List.length l
  | Parallele(a,b) -> verifArite env a && verifArite env b
  | BlocESV(l,b) -> verifArite env b;;
```

La fonction `getArite`, permet d'extraire l'arité de la fonction à partir de l'environnement, en utilisant la fonction `arite` A.5.

```
(*Arite de la fonction a partir de l'environnement*)
let getArite env n = Par.arite(List.find(fun d->Par.funName d=n) env)
```

2.2.3 Vérification qu'une variable n'est pas multi-définie dans un bloc

La fonction `verifDoubB` vérifie que les variables ne sont pas multi-définies dans un `bloc`

```
(*Verifier s'il y a des doublons dans la declaration du bloc*)
let rec verifDoubB=function
    BlocES(a,id,l) -> ExList.duplicate l
  | Parallele(a,b) -> verifDoubB a && verifDoubB b
  | BlocESV(l,b) -> ExList.duplicate l && verifDoubB b;;
```

La fonction `verifDoubD` vérifie qu'il n'existe pas de doublons dans une définition de fonction, en vérifiant qu'il n'existe pas de doublons dans l'interface de la fonction, que les variables locales ne sont pas présentes dans l'interface et inversement et enfin elle vérifie qu'il n y a pas de variables multi-définies dans le corps de la fonction.

```
(*Verifier s'il y a des doublons dans la definition de fonction*)
let rec verifDoubD=function
    Fonction(n,l,b) -> ExList.duplicate (ExList.union l (Par.linkedVars b)) &&
        verifDoubB b
  |BoiteNoire(n,l) -> ExList.duplicate l;;
```

Remarque Les fonctions supplémentaires sur les listes ont été définies dans le module `EXList` A.1.

Chapitre 3

Définition de la sémantique du langage

Ce chapitre a pour but de spécifier la sémantique de notre sous-langage identifié, en spécifiant la sémantique de chacun de ses termes et de montrer sa capacité de modélisation en reprenant les fonctions de base des langages synchrones (`Lustre`, `Esterel` et `Signal`) et en l'utilisant pour modéliser quelques systèmes.

3.1 Sémantique de chaque terme du langage

`BlocES` est un `bloc` spécifiant une contrainte entre ses entrées et ses sorties. Il correspond à un composant qui à partir de ses entrée fait un certain nombre de traitements et génère des sorties, c'est le composant le plus élémentaire.

`BlocESV` est un `bloc` hiérarchique avec entrées sorties et variables locales. Il correspond à un ensemble de composants regroupés dans un seul composant, ces composants sont inter-connectés entres eux et ces interactions représentent les variables locales du `bloc` englobant.

`Parallele` est la composition de blocs, pour ne pas s'intéresser à l'ordonnancement entre composants, nous supposons que tous les composants sont en parallèle.

Remarque Dans notre langage, nous ne faisons pas de différence entre les entrées et les sorties de `bloc`, nous les considérons juste comme des points d'interactions.

3.2 Bibliothèques langages synchrones

Pour pouvoir utiliser des exemples écrits dans les langages synchrones : `Lustre`[4], `Signal`[6] et `Esterel`[3], une bibliothèque des fonctions primitives de chacun de ces langage était définie dans la syntaxe de notre langage.

3.2.1 Bibliothèque Lustre

Les trois fonctions primitives de `Lustre` sont essentiellement

```
(*Fonction primitives du langage Lustre*)
module Lustre=struct
  let fIf p1 p2 p3 p4=BlocES("if",1,[p1;p2;p3;p4])          (*if ... then ... else ...*)
  let fPre p1 p2=BlocES("pre",2,[p1;p2])                    (*pre(...)*)
  let fFollow_by p1 p2 p3=BlocES("->",3,[p1;p2;p3])         (*...->...*)
end
```

3.2.2 Bibliothèque Signal

```
(*Fonction primitives du langage Signal*)
module Signal=struct
  let eqClock p1 p2= BlocES("^=",4,[p1;p2])      (*egalite d'horloge*)
  let uClock p1 p2 p3= BlocES("^+",5,[p1;p2;p3])(*l'union des horloges*)
  let init p1 p2= BlocES("init",6,[p1;p2])       (*valeur initile*)
  let pre p1 p2= BlocES("pre",7,[p1;p2])       (*valeur a l'instant precedent*)
  let default p1 p2 p3= BlocES("default",8,[p1;p2;p3]) (*S:=A default B melange les
      signaux A et B avec priorite pour A*)
  let whenB p1 p2 p3= BlocES("when",9,[p1;p2;p3])       (*S:= val1 when cond1*)
end
```

3.2.3 Bibliothèque Esterel

```
(*Fonctions primitives du langage Esterel*)
module Esterel=struct
  let await p1 p2 p3=BlocES("await",10,[p1;p2;p3])      (*await...do...end*)
  let fExit p1 p2=BlocES("exit",11,[p1;p2])
  let present p1 p2 p3=BlocES("present",12,[p1;p2;p3]) (*present R then...end*)
  let loop p1 p2 p3=BlocES("loop",13,[p1;p2;p3])        (*loop...each TIC...end*)
  let emit p1 p2=BlocES("emit",14,[p1;p2])              (*emit SORTIE*)
  let every p1 p2 p3=BlocES("every",15,[p1;p2;p3])      (*every tick do...end*)
  let abort p1 p2 p3=BlocES("abort",16,[p1;p2;p3])      (*abort...when...*)
  let pre p1 p2=BlocES("pre",17,[p1;p2])
  let pause p1 p2=BlocES("pause",18,[p1;p2])            (*equivalent a await tick*)
  let suspend p1 p2 p3=BlocES("suspend",19,[p1;p2;p3]) (*suspend...when...*)
  let repeat p1 p2 p3=BlocES("repeat",20,[p1;p2;p3])    (*repeat...times...end*)
  let case p1 p2 p3=BlocES("case",21,[p1;p2;p3])        (*case C1 do S1*)
  let nothing p1=BlocES("nothing",22,[p1])              (*termine instantanement*)
  let halt p1 p2=BlocES("halt",23,[p1;p2])              (*attend indefiniment*)
  let call p1 p2=BlocES("call",24,[p1;p2])              (*appel de procedure*)
  let seq p1 p2=Parallele(p1,p2)                        (*mise en sequence*)
  let par p1 p2=Parallele(p1,p2)                        (*mise en parallele*)
  let weakAbort p1 p2 p3=BlocES("weak abort",25,[p1;p2;p3])(*weak abort p when d*)
end
```

3.3 Exemples

Pour démontrer l'expressivité de notre langage, nous allons l'utiliser pour modéliser des exemples tirés du [12].

3.3.1 Exemple de la minuterie

Le premier exemple est un compteur décroissant, à tout instant la variable `Raz` est testée. Si elle vaut `true` alors la valeur courante du compteur, représentée par `DelaiRestant`, est affectée avec la valeur de `DelaiInitial`. Sinon, la valeur courante de `DelaiRestant` sera affectée avec sa valeur

précédente moins 1, sauf à premier pas elle vaudra `DelaiInitial`. L'exemple était modélisé avec le langage `Lustre` dans [12] comme suit :

```
node Minuterie (Delai_initial: int; Raz:bool)
       returns (Delai_restant: int);
let equa eq_Minuterie[ , ]
   Delai_restant = if Raz then Delai_initial
                   else Delai_initial -> pre(Delai_restant) - (1);
tel;
```

Il était aussi modélisé avec l'équivalent graphique de `Lustre` comme le montre la Figure 3.1 :

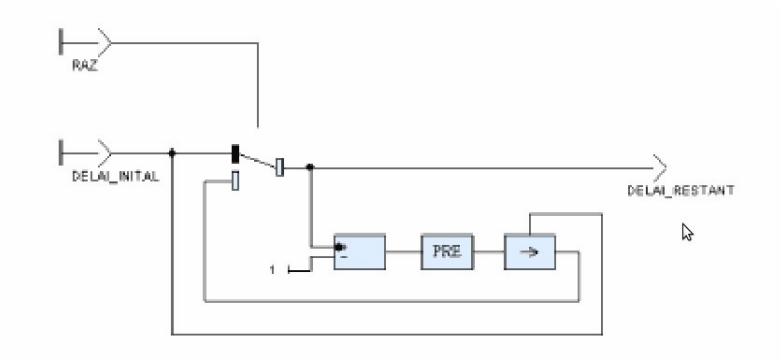

FIG. 3.1 – Noeud Minuterie

Les primitives de `Lustre` définies précédemment permettent la description du noeud `minuterie` dans notre langage.

```
let minuterie=
 Fonction ("Minuterie", ["DelaiInitial"; "Raz"; "DelaiRestant"],
    BlocESV(["DelaiRestant1";"PreDelaiRestant"; "Const 1"; "PreDelaiRestant-1"],
            Parallele
              (Parallele
                 (Lustre.f_if "Raz" "DelaiInitial" "DelaiRestant1" "DelaiRestant",
                  Parallele
                    (Parallele
                       (Lustre.f_pre "DelaiRestant" "PreDelaiRestant",
                        BlocES ("gen", 4, ["Const 1"])),
                     BlocES ("moins", 3,
                             ["PreDelaiRestant"; "Const 1"; "PreDelaiRestant-1"]))),
               (Lustre.f_follow_by "DelaiInitial" "PreDelaiRestant-1" "DelaiRestant1"))))
```

3.3.2 Exemple du contrôle du gestion alarmes

La Figure 3.2 montre la modélisation proposé avec l'équivalent graphique du `Lustre`.
On peut la décrire mathématiquement comme suit :

$$\exists \text{ DebitC, Debit, EtatRemplissageP (Plus(Debit, EtatRemplissageP, DebitC) } \wedge \text{ Moins(DebitE, DebitS, Debit) } \wedge \text{ Pre(EtatRemplissage, EtatRemplissageP) } \wedge \text{ TestEtatCiterne(DebitC, AlarmeHaut, AlarmeBas, Normal, EtatRemplissage))}$$

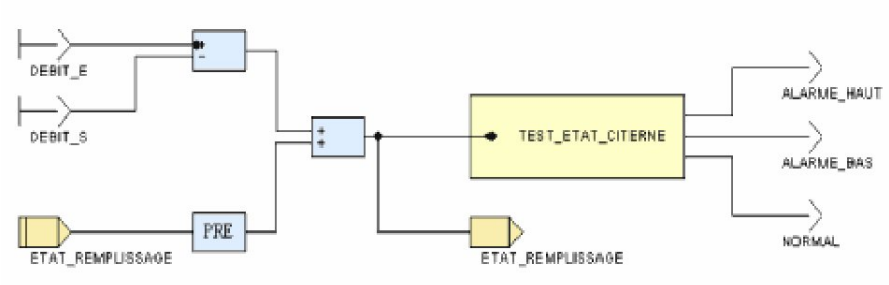

Fig. 3.2 – Gestion alarmes

L'exemple peut être décrit dans une autre notation, les variables dans la portée du var sont les points d'interactions entre les blocs.

var DebitC Debit EtatRemplissageP. (Plus(Debit, EtatRemplissageP, DebitC) || Moins(DebitE, DebitS, Debit) || Pre(EtatRemplissage, EtatRemplissageP) || TestEtatCiterne(DebitC, AlarmeHaut, AlarmeBas, Normal, EtatRemplissage))

Dans notre langage le `bloc GestionAlarme` peut être écrit comme suit :

```
let gestionAlarme=
 BlocESV (["DebitC"; "Debit"; "EtatRemplissageP"],
          Parallele
            (Parallele (BlocES ("Plus", 1, ["Debit"; "EtatRemplissageP"; "DebitC"]),
                 Parallele (BlocES ("Moins", 2, ["DebitE"; "DebitS"; "Debit"]),
                            BlocES ("Pre", 3, ["EtatRemplissage"; "EtatRemplissageP"]))),
             BlocES ("TestEtatCiterne", 4,
              ["DebitC"; "AlarmeHaut"; "AlarmeBas"; "Normal"; "EtatRemplissage"]))
```

Chapitre 4

Identification des transformations

Notre but dans ce chapitre est d'identifier des transformations endogène (le résultat de la transformation sera dans le même langage de modélisation), horizontale (dans le même niveau d'abstraction) et vérifiables sur notre langage. La première transformation étudiée est le déplacement d'un `bloc`. Une première application pour cette transformation est de faire sortir un composant qui a une fréquence différente du reste des composants contenus dans le même composant englobant.

4.1 Déplacement d'un bloc

Le déplacement d'un `bloc` consiste à surgir un composant particulier du système en le faisant sortir de son système englobant et le mettre dans un composant à part en conservant ses points d'interactions avec les autres composants. Cela entraîne la séparation de ce composant en créant une nouvelle interface pour le reste des composant et les mettre en parallèle avec le `bloc` surgi. Les Figures 4.1 et 4.2 montrent la transformation `Move` du `bloc C`.

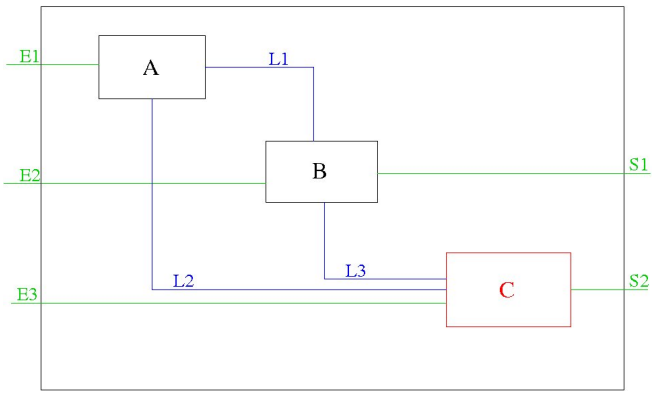

Fig. 4.1 – Bloc avant transformation

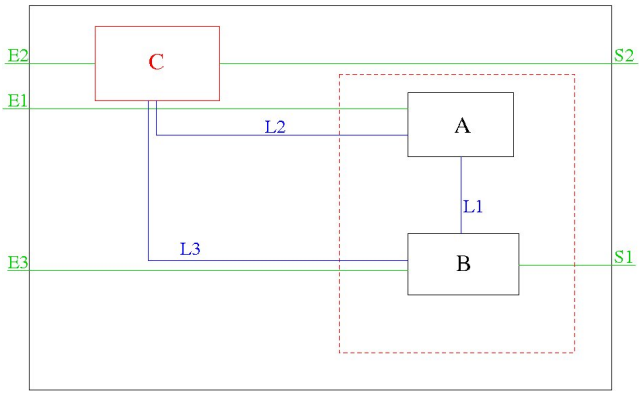

FIG. 4.2 – Résultat de la transformation Move C

Pour pouvoir identifier cette transformation, nous allons passer par plusieurs transformations plus élémentaires :

4.1.1 Move d'un bloc première version

Pour continuer avec l'exemple de `la Citerne`, le `bloc` `TestEtatCiterne` contient les trois `blocs` `GestionAlarme`,`Minuterie` et `Memorisation` en parallèle.

(GestionAlarme || Minuterie) || Memorisation

La transformation Move Minuterie appliquée au `bloc` TestEtatCiterne doit rendre le résultat suivant :

Minuterie || (GestionAlarme || Memorisation)

Remarque Cette transformation ne peut pas être constatée graphiquement.

Transformations élémentaires

Cette transformation est obtenue en combinant plusieurs transformations encore plus élémentaires essentiellement `swap`, `ShiftLeft`, `ShiftRight`, `appl1`, `appl2`, `descendVar` et `keepVar`. La transformation `swap` permet d'inverser l'ordre des deux arguments d'un `bloc` `Parallele`.

swap (B1 || B2) → (B2 || B1)
swap var $x_1x_2...x_n$.B = var $x_1x_2...x_n$.B
swap b = b

Les deux transformations `shiftRight` et `shiftLeft` permettent aussi de changer l'ordre dans un `bloc` `Parallele` contenant un autre `bloc` `Parallele` imbriqué.

shiftRight (B1 || B2) || B3 = B1 || (B2 || B3)

shiftLeft B1 || (B2 || B3) = (B1 || B2) || B3

Les deux fonctions `appl1` et `appl2` permettent dans l'ordre d'appliquer une fonction `f` sur le premier et le deuxième argument d'un `bloc Parallele`.

$$\textbf{appl1}\ \mathrm{f}\ (\mathrm{B1}\ ||\ \mathrm{B2}) = (\mathrm{f(B1)}\ ||\ \mathrm{B2})$$

$$\textbf{appl2}\ \mathrm{f}\ (\mathrm{B1}\ ||\ \mathrm{B2}) = (\mathrm{B1}\ ||\ \mathrm{f(B2)})$$

La fonction `descendVar` permet de descendre une variable x_1 si elle n'est pas dans les variables libres du `bloc B1`.

$$\textbf{descendVar}\ \mathrm{var}\ \mathrm{x_1x_2...x_n.(B1}\ ||\ \mathrm{B2)} = \mathrm{var}\ \mathrm{x_2...x_n.(B1}\ ||\ \mathrm{var}\ \mathrm{x_1.B2)}$$

La fonction `keepVar` permet de mettre x_1 à la dernière position si elle est dans les variables libres du `bloc B1`.

$$\textbf{keepVar}\ \mathrm{var}\ \mathrm{x_1x_2...x_n.(B1}\ ||\ \mathrm{B2)} = \mathrm{var}\ \mathrm{x_2...x_nx_1.(B1}\ ||\ \mathrm{B2)}$$

La fonction `applyInside` permet d'appliquer une fonction `f` sur le `bloc` interne.

$$\textbf{applyInside}\ \mathrm{f}\ \mathrm{var}\ \mathrm{x_1x_2...x_n.B} = \mathrm{var}\ \mathrm{x_1x_2...x_n.f(B)}$$

Toutes ces fonctions étaient utilisées essenciellement pour implémenter deux autres fonction `permute1` et `permute2`. Ces deux fonctions permettent dans l'ordre de remonter le `bloc` que nous voulons surgir s'il est dans la première projection et la deuxième projection d'un `bloc Parallele`. La fonction `moveOut` permet d'effectuer un appel récursif sur les deux projection du `bloc` (si le `bloc` est un `Parallele`) des fonctions `permute1` et `permute2` ce qui permet d'executer complètement la transformation.

4.1.2 Implémentation en Ocaml

Afin de rendre le code plus modulaire, les modules d'`Ocaml` [8] sont utilisés. Ici nous introduisons une brève définition de la signature et l'algèbre avant de les utiliser pour implémenter les transformations.

Signature définit l'interface abstraite qui contiendra les types ainsi que les opérations exportés par le module, cette signature est la donnée deux ensembles, $|\Sigma|$ le support de la signature qui contient les types, et Ω qui est l'ensemble des définitions de fonctions.

Algèbre Une algèbre est une implémentation de la signature.

Module MoveFirst Le Module `MoveFirst` définit la fonction `Move` qui met un `bloc` choisi comme premier argument du `bloc Parallele` le plus externe. L'interface abstraite qui contiendra les types ainsi que les opérations exportées par le module `MoveFirst`, est la signature `BnOp` (Binary Operation) citée dans A.3.

Spécialement dans notre cas c'est l'opération de synchronisation (paralellisme) entre les blocs, notre Σ algèbre associe à `t` le type `bloc` et implémente toutes les fonctions dont le type est défini dans la signature `MoveFirst`. Ainsi, nous pouvons définir le module `MoveFirst` paramétré par le module `BnOP`, qui définit la fonction `Move` faisant intervenir les deux fonctions `permute1` et `permute2`. Pour cela, deux styles de programmation peuvent être utilisés, le premier utilise une exception pour signaler que la fonction `permute1` n'est pas appliquée et appliquer `permute2` dans ce cas.

```
module MoveFirst(Bo:BnOp) = struct
  open Bo
  exception NotDone

 (*surgir le bloc d'identifiant id s'il est dans le bloc b1*)

 let permute1 id x=if is_o x then let b1,b2(Bo.p1 x, Bo.p2 x) in
    if is_o b1
    then let b11,b12 = (Bo.p1 b1, Bo.p2 b1) in
      if eq_id id b11
      then right x
      else
        if eq_id id b12
        then right(swap x)
        else raise NotDone
    else raise NotDone
 else raise NotDone

  (*surgir le bloc d'identifiant id s'il est dans le bloc b2*)

 let  permute2 id x=if is_o x then let b1,b2=(Bo.p1 x, Bo.p2 x) in
    if is_o b2
    then
      let b11,b12 = (Bo.p1 b2, Bo.p2 b2) in
        if eq_id id b11
        then right(swap x)
        else
          if eq_id id b12
          then right(swap(comp b1 (swap b2)))
          else x
    else
      if eq_id id b2
      then comp b2 b1
      else x
 else x

(*appel recursif de permute1 et de permute2*)
 let rec Move id x=
    if is_o x then  let x1,x2 = (Bo.p1 x, Bo.p2 x) in
      try
        permute1 id (comp (Move id x1) (Move id x2))
      with
          NotDone->permute2 id x (*si permute1 n'est pas faite*)
    else x
end
```

Dans cette version, l'utilisation de `raise` et `try` dans la fonction `Move` donne plus d'élégance au programme mais ne se prête pas aujourd'hui pour faire de la preuve. Pour cela nous allons le réecrire en utilisant le type `option`. Le code complet est listé dans A.4.

Remarque Remarquons que cette version simpliste, ne permet pas de sortir un `bloc` d'un `bloc` contenant des variables locales. Pour le faire nous devons traiter tous les cas concernant la liaison de ce dernier avec les variables locales du `bloc` englobant, c'est ce que nous essayerons de faire dans la transformation suivante.

4.1.3 Move d'un bloc deuxième version

Je me permets ici d'utiliser la notation avec var que nous avons définit dans 3.3.2 pour exprimer le problème et la solution proposée. Soit A, B et C trois blocs en parallèle dans un `bloc` définissant les variables locales x, y et z comme suit :

$$\text{var xyz.(A(x, y) || B(y, z) || C(t))}$$

Vue que le `bloc C` est libre, la transformation `move` du `bloc C` doit donner le résultat suivant :

$$\text{C(t) || (var xyz.(A(x, y) || B(y, z)))}$$

Nous définissons la fonction `moveF` qui permet de sortir le `bloc` de la portée du var après vérification qu'il est libre. Cette dernière fonction ne peut être appliquée que après application de la fonction `move` précédente et c'est le rôle de la fonction `moveFb`.

Règles de transformation Nous définissons deux règles :

- **La règle moveI** permet de surgir le `bloc` à l'intérieur du `bloc` englobant, son résultat sur l'exemple précédent est :
$$\text{var xyz.(C(t) || A(x, y) || B(y, z)}$$
- **La règle moveF** permet de sortir le `bloc` s'il est libre, ce qui donne le résultat de la fonction
$$\text{C(t) || (var xyz.(A(x, y) || B(y, z)))}$$

En prenant l'exemple du `bloc` GestionAlarme suivant :

```
BlocESV (["DebitC"; "Debit"],
 Parallele
  (Parallele (BlocES ("Plus", 1, ["Debit"; "EtatRemplissage"; "DebitC"]),
    Parallele (BlocES ("Moins", 2, ["DebitE"; "DebitB"; "Debit"]),
     BlocES ("Pre", 3, ["EtatRemplissage"]))),
  BlocES ("TestEtatCiterne", 4,
   ["DebitC"; "AlarmeHaut"; "AlarmeBas"; "Normal"])))
```

L'application de la transformation `moveF` du `bloc Pre` sur cette exemple, donne le résultat suivant :

```
Parallele (BlocES ("Pre", 3, ["EtatRemplissage"]),
 BlocESV (["DebitC"; "Debit"],
  Parallele
   (Parallele (BlocES ("Plus", 1, ["Debit"; "EtatRemplissage"; "DebitC"]),
     BlocES ("Moins", 2, ["DebitE"; "DebitB"; "Debit"])),
   BlocES ("TestEtatCiterne", 4,
    ["DebitC"; "AlarmeHaut"; "AlarmeBas"; "Normal"]))))
```

Remarque Dans cette version aussi, on ne peut sortir qu'un `bloc` libre d'un `bloc` contenant des variables locales, le but de la version suivante est de sortir même un `bloc` lié.

4.1.4 Move d'un bloc troisième version

Supposons `A`, `B` et `C` des blocs avec les ports `x`, `y` et `z` comme suit :

$$\text{var xyz.(A(x, y) || B(y, z) || C(x))}$$

La sémantique de cette expression est :

$$\exists \text{ xyz. } (\text{A(x, y)} \wedge \text{B(y, z)} \wedge \text{C(x)})$$

L'application de la transformation move sur le `bloc C` doit donner le résultat suivant :

$$\text{var x.(C(x) || (var yz.(A(x, y) || B(y, z)))}$$

Pour appliquer cette transformation, nous appliquons deux règles. La règle **moveI** qui consiste à appliquer la fonction `move` sur la partie interne du `bloc` :

$$\text{var xyz.(C(x) || (A(x, y) || B(y, z)))}$$

La règle **lDivide** permet de diviser la liste des variables locales en deux parties comme suit :

$$\text{var x.var yz. (C}(x)\text{ || (A(x, y) || B(y, z)))}$$

Enfin la règle **moveF1** permet de sortir le `bloc` C, on aura le résultat suivant :

$$\text{var x. (C}(x)\text{ || var yz.(A(x, y) || B(y, z)))}$$

4.1.5 Move d'un bloc au niveau fonction

Maintenant que nous avons les moyens pour appliquer la fonction `Move` au niveau `bloc`, nous pouvons les utiliser pour appliquer cette transformation à un niveau d'abstraction supérieur (le niveau des fonctions). Supposons que nous avons une fonction $\text{F(p}_1\text{...p}_\text{n}\text{)\{... || B(...) || ...\}}$ et que nous voulons sortir le `bloc B`. La première étape est d'appliquer la fonction `Move` sur le corps de la fonction à l'aide de la fonction `appF`, le résultat sera $\text{F(p}_1\text{...p}_\text{n}\text{)\{var varL}_1\text{...varL}_\text{m}\text{.B(...)||...\}}$.

```
(*Appliquer moveOutFv1 sur le corps de la fonction*)
let extentionBeta id f =appF (moveOutFv1 id) f
```

La deuxième étape est de construire à l'intérieur de la fonction `F` une autre fonction `FNew` qui contient tous les composants de F sauf le `bloc` surgi et qui a donc comme interface les paramètres de la fonction F avec les variables locales liées au `bloc` surgi. Cela se fait à l'aide de deux autres fonctions : La première est la fonction `varLG` qui ajoute les variables locales du `bloc` surgi au variables globales de la fonction `F`, l'application de cette fonction sur la fonction `F` donne la définition de la fonction `FNew` : $\text{FNew(p}_1\text{...p}_\text{n}\text{, vL}_1\text{...vL}_\text{m}\text{)\{}B(...)\text{||...\}}$. La deuxième est la fonction `moveB1` qui enlève le premier `bloc` du parallèle alors la fonction `FNew` sera : $\text{FNew(p}_1\text{...p}_\text{n}\text{, varL}_1\text{...varL}_\text{m}\text{)\{...\}}$.

```
(*F1(x,y){\t.C||A||B} -> newF1(x,y,z){A||B}*)
let generalise f=match (funCorps f) with
    Some b->moveB1(varLG f)
  |_->f (*cas d'une boite noire*)
```

Une fonction d'optimisation peut être appliquée sur la fonction `FNew` pour ne laisser que les variables libres du `bloc` implémentant la fonction dans l'interface de `FNew`, elle donne la dernière définition de `FNew` : $FNew(x_1...x_t)\{...\}$ qui sera enregistrée dans l'environnement.

```
(*Optimiser la definition d'une fonction*)
 let opt=function
      Fonction(n,l,b) -> Fonction(n,freeVars b,b)
    |b->b
```

Enfin, la définition de la fonction `F` doit être changée dans l'environnement avec ça nouvelle définition : $F(p_1...p_n)\{B(...)||FNew(x_1...x_t)\}$

4.2 Connexion d'un bloc

Pour étudier les différents cas qui s'imposent, nous prenons l'exemple d'une fonction F 4.3 où nous voulons connecter les deux blocs `B` et `C`.

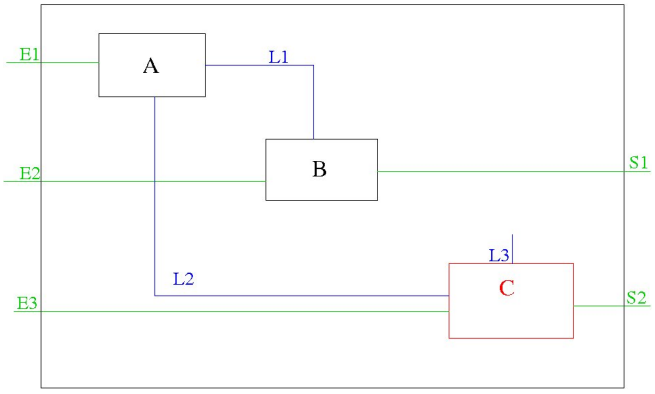

FIG. 4.3 – Fonction F

Une autre représentation de la Fonction F est :

$$F(E1, E2, E3, S1, S2)\ \{\text{var L1 L2 L3. } A(E1, L1, L2) \mid\mid B(E2, L1, S1) \mid\mid C(E3, L2, S2)\}$$

4.2.1 Connexion avec un port externe

Le renommage du port `S2` en `S1` cause la présence de deux variables de même nom dans l'interface de la fonction `F` ce qui n'est pas acceptable dans notre syntaxe, le paragraphe 2.2.3 parle en détail sur ce point.

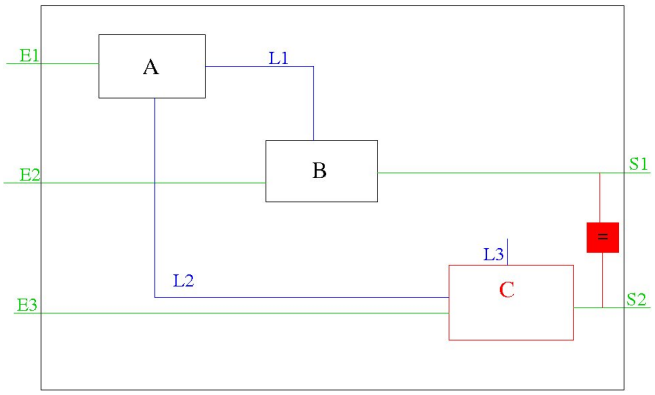

FIG. 4.4 – Connexion de S1 et S2

Avec la syntaxe textuelle cette transformation va donner :

$$F(E1, E2, E3, S1, S1)\ \{\text{var L1 L2 L3.A}(E1, L1, L2)\ ||\ B(E2, L1, S1)\ ||\ C(E3, L2, L3, S2)\}$$

4.2.2 Connexion avec un port interne non libre

L'utilisation du port `S1` et du port `L2` pour la connexion comme le montre la figure 4.5 engendre aussi une connexion avec le `bloc A` alors que nous ne voulons connecter que les blocs `B` et `C`.

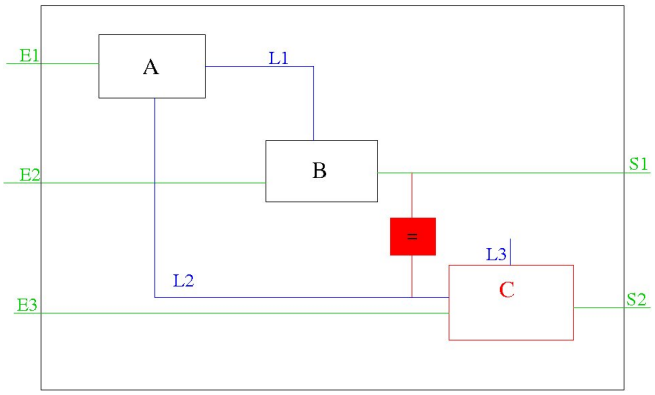

FIG. 4.5 – Connexion de S1 et L2

La syntaxe textuelle équivalente à cette fonction :

$$F(E1, E2, E3, S1, S2)\ \{\text{var L1 L3.A}(E1, L1, S1)\ ||\ B(E2, L1, S1)\ ||\ C(E3, S1, S2, L3)\}$$

4.2.3 Connexion avec un port interne libre

La connexion du port `S1` avec le port `L3` du `bloc C` ne pose aucun problème, nous ne risquons pas de connecter d'autres composants ni d'avoir les mêmes variables dans l'interface, la figure 4.6 montre le résultat de la connexion.

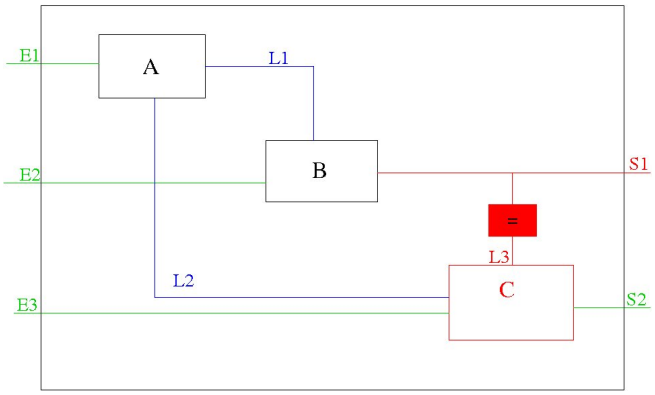

FIG. 4.6 – Connexion de S1 et L3

La Syntaxe textuelle de la fonction résultante est la suivante :

$$F(E1, E2, E3, S1, S2) \text{ var } L1\ S1.A(E1, L1, S1)||B(E2, L1, S1)||C(E3, L2, L3, S1)\}.$$

Une autre solution que nous pouvons imaginer pour la connexion dans le cas où il n y a pas de ports libres pour la connexion est l'ajout d'un nouveau port interne. Cette solution n'est pas permise car elle engendre un changement d'interface (l'arité) de la fonction.

Conclusion L'étude des différents cas permet d'extraire la notion de `bloc connectable`. Un `bloc` est `connectable` dans une définition de fonction s'il contient au moins un port interne qui n'est pas connecté avec un autre composant (un port interne libre) 4.2.3. La fonction connectable prenant en argument un `bloc b` et une définition de fonction `d`, permet d'extraire l'ensemble des ports de `b` acceptant d'être connectés dans `d`, rend `None` dans le cas ou le `bloc` n'est pas connectable.

```
(*les ports de connexion d'un bloc dans une definition*)
let connectable b d=match funCorps d with
      None-> None
  | Some c-> if (not(ExList.inclus (freeVars b) (funPar d)))(*il n'y a des ports locals
      lies au bloc*)
 then if(not(ExList.inclus  (freeVars b) (varsReste b c))) then Some(ExList.removeL (
     freeVars b) (ExList.interListe (varsReste b c) (freeVars b))) else None else None
```

La fonction `varsReste` utilisée, permet d'extraire de la définition de fonction, les variables utilisées dans les autres blocs que le `bloc b` et les éliminer pour avoir que les variables internes libre de `b`.

```
let rec varsReste b db=match db with
          Parallele(b1, b2)->if b1=b then varsReste b b2 else if b2=b then varsReste b b1
               else ExList.union (varsReste b b1) (varsReste b b2)
    | BlocESV(l, b1)->if b1=b then [] else varsReste b b1
    | BlocES(_,_,_) as b1->if(b1=b) then [] else freeVars b1
```

Une fois les ports de connexions du `bloc b` sont identifiés, il suffit d'ajouter un composant qui se charge de faire l'égalité entre le port choisi du `bloc b` et le port où nous voulons le connecter.

Chapitre 5

Passage de Caml à Java

Cette section a pour but de comparer les deux approches de programmation Caml et Java en réimplémentant le type Bloc, DefinitionBloc et les transformations précédemment implémentées en langage Java.

5.1 Définition du langage

En Java, le type`Bloc` peut être défini comme une classe abstraite définissant aussi des méthodes abstraites. Les types `BlocES`, `BlocESV` et `Parallele` sont définis comme des classes qui implémentent la classe abstraite `Bloc`. Java n'est pas capable de faire du pattern-matching. En revanche, pour les variants, nous pouvons nous en sortir, il suffit de mettre une méthode abstraite dans la classe abstraite `Bloc` et de l'implémenter dans les trois classes qui implémente `Bloc`. Un extrait de la classe `Bloc` est le suivant :

```
abstract class Bloc implements Serializable {
  abstract boolean is_o();
  abstract Bloc p1();
  abstract Bloc p2();
  abstract boolean eq_id(int id);
  abstract Bloc shiftRight();
  abstract Bloc swap();
  abstract ArrayList<String> linkedVars();
  abstract ArrayList<String> freeVars();
  abstract Bloc appl1(Function<Bloc, Bloc> f);
  abstract Bloc appl2(Function<Bloc, Bloc> f);
  abstract String afficher();
  abstract Bloc applyInside(Function<Bloc, Bloc> f);
  abstract Bloc descendVar();
```

Les classes `BlocES`, `BlocESV` et `Parallele` implémentent la classe `Bloc`, elles doivent contenir l'implémentation de chaque fonction de la classe abstraite `Bloc`. Voici un extrait de la définition de la fonction `Parallele`.

```
public class Parallele extends Bloc {

  private static final long serialVersionUID = 1L;

  Bloc b1;

  Bloc b2;

  public Parallele() {
    this.b1 = new BlocES();
    this.b2 = new BlocES();
  }

  Parallele(Bloc b1, Bloc b2) {
    this.b1 = b1;
    this.b2 = b2;
  }

  public Bloc p1() {
    return this.b1;
  }

  public Bloc p2() {
    return this.b2;
  }

  public boolean eq_id(int id) {
    return false;
  }

  public Bloc shiftRight() {
    if (this.is_o()) {
      if (this.p1().is_o()) {
        return new Parallele((this.p1()).p1(), new Parallele(this.p1()
        .p2(), this.p2()));
      } else {
        return this;
      }
    } else {
      return this;
    }
  }

  public Bloc swap() {
    return new Parallele(this.b2, this.b1);
  }
```

L'environnement est implémenté aussi en Java comme une liste de définitions de fonctions. Il est sérialisé à chaque fin de session et désérialisé au début de chaque session pour récupérer les définition de fonctions existantes sous forme d'un arbre de définitions. La structure ArrayList Java fournit un accès aux éléments de l'environnement pour des opérations d'ajout, suppression et de modification d'une définition dans l'environnement.

5.2 Implémentation des transformations

La traduction des fonctions de transformations est assez directe. Pour l'exemple la fonction `permute1` 4.1.2, peut être écrite simplement comme suit :

```
static Bloc permute1(int id, Bloc b) {

  if (b.is_o()) {
    if (b.p1().is_o()) {
      if (b.p1().p1().eq_id(id)) {
        return b.shiftRight();
      } else {
        if (b.p1().p2().eq_id(id)) {

          return b.appl1(swap()).shiftRight();
        } else {
          return null;
        }
      }
    } else {
      if (b.p1().eq_id(id)) {
        return b;
      } else {
        return null;
      }
    }
  } else {
    if (b.eq_id(id)) {
      return b;
    } else {
      return null;
    }
  }
}
```

Une modeste interface graphique était élaborée pour simplifier la saisie des définition des blocs, représenter les blocs dans un format graphique et visualiser les transformations. la Figure 5.1 montre notre interface graphique avec à gauche un environnement contenant les fonctions primitives des langages synchrones vues dans 3.2 et à droite l'exemple de la minuterie que nous avons présenté dans 3.3.1.

L'interface à travers le bouton `New` de son menu permet de créer une nouvelle définition de fonction (`Function` ou `Black Box`) et d'initialiser l'espace de travail (`Init`). Le bouton `Edit` donne les moyens permettant de construire ou de modifier le corps d'une définition de fonction :

- `Define` pour construire la fonction du bloc sélectionné dans l'espace de travail et l'ajouter à la bibliothèque.
- `Insert` pour insérer un nouveau bloc ou un bloc déjà défini dans la bibliothèque.
- `Remove` pour supprimer un bloc.
- `Save` pour enregistrer la définition permanentement dans la bibliothèque.

Le bouton `Transforme` propose le choix entre les transformations :

- En mettant une définition de fonction dans l'espace de travail (une définition existante dans la bibliothèque peut être ajoutée à l'espace de travail avec un double clique sur sa définition) et en cliquant sur le bouton `Move`, l'interface demande l'identifiant du bloc qu'on veut sortir et visualise la fonction résultante en changeant la définition de la fonction initiale dans la bibliothèque.
- Le bouton `connecte`, après vérification que le bloc sélectionné est connectable dans l'espace de travail et après demande d'un port de connexion, permet de connecter le bloc en renommant son port de connexion avec le nom du port de connexion choisi. Une deuxième solution pour faire la connexion est celle que nous avons cité dans 4.2.3.
- Le bouton `Abstract`, construit une définition pour le bloc sélectionné à l'environnement et le remplace par une boite noire (un appel de sa fonction) dans l'espace de travail.
- Le bouton `Fill` permet de remplacer le bloc sélectionné par sa définition si elle existe dans

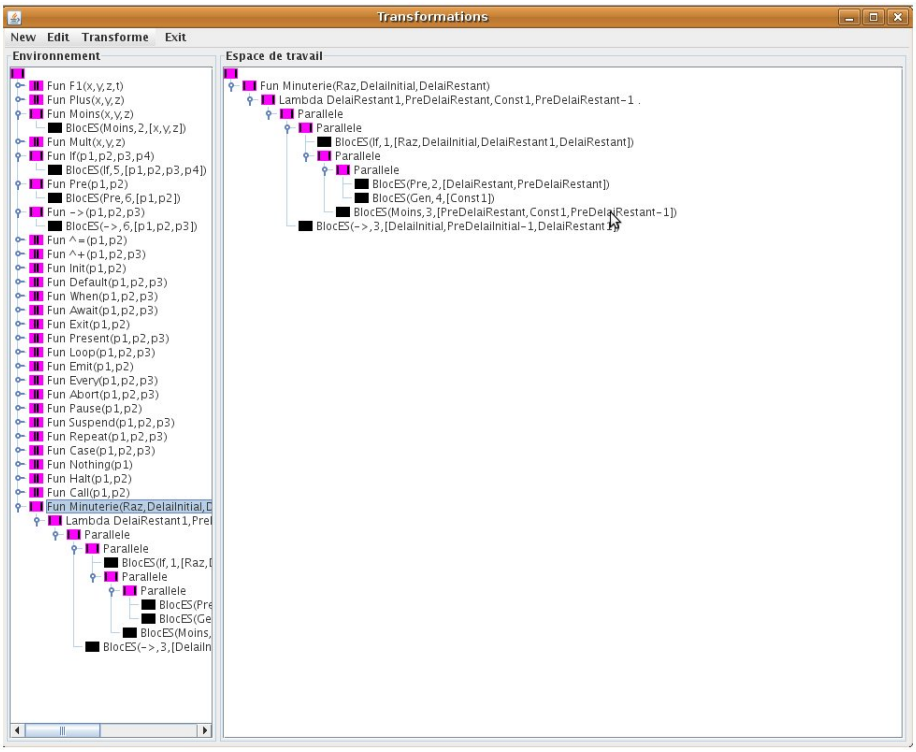

FIG. 5.1 – Interface Graphique

la bibliothèque (l'opération inverse d'abstract).

– Le bouton `Cut` permet de couper un bloc et le mettre dans un buffer et après le mettre dans un autre emplacement choisi avec le bouton `Paste`.

Chapitre 6

Vérification en Coq

Le but de ce chapitre est de vérifier la correction des transformations proposées dans 4. Pour atteindre cet objectif, une formalisation du langage ainsi que toutes les fonctions de transformation dans Coq était nécessaire.

6.1 Présentation de l'assistant de preuve Coq

Coq [1] est un assistant de preuve. C'est le résultat d'une dizaine d'années de recherche du projet Coq [2]. Pour communiquer avec Coq, il faut obéir à des règles d'un langage précis qui contient un nombre de commandes et des conventions syntaxiques. Le langage utilisé pour décrire les termes, les types, les preuves et les programmes est appelé Gallina et le langage de commande est appelé Vernacular. La définition précise de ce langage est donnée dans le manuel de référence [11].

6.2 Les types (Sorts) dans Coq

Les principaux types de Coq sont :
- Prop est le type des propriétés logiques.
- Set est le type des types de données.
- Type est le type de Set et de Prop.

Dans notre cas, les noms et les variables des blocs étaient déclarés de type Set.

```
Variable name:Set.                    (* nom des blocs de base *)
Variable var:Set.                     (*port d'un bloc*)
```

L'interface d'un bloc est déclaré comme une fonction qui associe à chaque nom d'un bloc élémentaire une list de variables.

```
Variable BBInterf: name->list var. (*interface des blocs de base*)
```

[1] http ://www.lix.polytechnique.fr/coq/
[2] http ://www.inria.fr/recherche/equipes/coq.en.html

6.3 Constructions inductives

Coq est un langage purement fonctionnel qui dispose de possibilités de récursion basées sur le calcul des constructions inductives [9]. Il donne la possibilité de définir des types récursifs qui représentent des ensembles infinis, puis des fonctions récursives sur ces types. Le type bloc 2.1 était défini de la manière suivante :

```
Inductive bloc:Set :=
   BlocES : name ->nat->bloc        (*fonction*)
 | BlocESV :list var->bloc->bloc (*declaration de variables locales*)
 | Parallele : bloc ->bloc->bloc.(*composition de blocs*)
```

Un principe d'induction est généré automatiquement par `Coq` en même temps que le type inductif pour permettre de raisonner par récurrence.

```
bloc_ind
     : forall P : bloc -> Prop,
       (forall (n : name) (n0 : nat), P (BlocES n n0)) ->
       (forall (l : list var) (b : bloc), P b -> P (BlocESV l b)) ->
       (forall b : bloc, P b -> forall b0 : bloc, P b0 -> P (Parallele b b0)) ->
       forall b : bloc, P b
```

De même pour le type `definitionBloc` 2.1, nous l'avons écrit sous forme d'un type inductive comme suit :

```
Inductive definitionBloc:Set :=
   Fonction: name->list var->bloc->definitionBloc
 | BoiteNoire: name-> list var->definitionBloc.
```

6.4 Vérification de la terminaison

En `Coq`, une fonction n'est définie que si `Coq` dispose de la preuve qu'elle se termine. Les programmes qui n'ont aucune itération terminent forcément. La définition d'une fonction non récursive est acceptée par `Coq`, si elle est bien écrite dans sa syntaxe. Nous prenons comme exemple la définition de la fonction `permute1` 4.1.2.

```
Definition permute1(id: nat)(x: bloc): option bloc :=(*ex: permute1 idC x*)
if is_o x
then
if is_o (p1 x)
     then
       if eq_id id (p1 ( p1 x))
       then Some (shiftRight x)          (*C||b21)||b2 -> C||(b21||b2)*)
       else
         if eq_id id (p2 (p1 x))
         then Some (shiftRight (appl1 swap x)) (*(b11||C)||b2*)
         else None
     else None                                (*C||(...) -> C||(...)*)
   else None                                  (*C ->C*).
```

La question de terminaison se pose donc que pour les fonctions récursives, car c'est le seul moyen pour faire des itérations en `Coq`. Il faut donc une manière de prouver qu'une fonction récursive termine. Il est facile pour `Coq` de prouver la terminaison des récursions sur les structures récursives définies par induction comme les listes à condition que l'appel récursif s'effectue sur un sous terme, car il exige grâce à son typeur que tous les tests de l'objet sur lequel on récurse soient complets, ce qui garantit d'avoir le cas de base pour tout terme inductif. Par exemple, `Coq` prouve seul la

terminaison de la fonction récursive `inclus` du module `EXList` A.1 juste en la définissant comme suit :

```
Fixpoint inclus(l1:list var) (l2:list var){struct l1}: (bool) :=
match l1 with
      nil=> true
    | x::q1=> if (In_dec  var_eq_dec x l2) then (inclus q1 l2) else false
    end.
```

Nous remarquons qu'il nous faut spécifier la variable sur laquelle on fait la récurrence structurelle dans une enclosure struct, car `Coq` ne pouvait pas la conclure seul. Dans d'autres fonctions la spécification de la variable de récurrence n'est pas obligatoire, nous citons pour l'exemple la fonction `duplicate` A.1.

```
Fixpoint duplicate(l:list var): bool :=
match l with
      nil=>true
    |x::l1=>if(In_dec  var_eq_dec x l1) then false else duplicate l1
    end.
```

Toutes les fonctions sur les listes A.1 étaient décrites de la même manière que `inclus` et `duplicate` dans la syntaxe de `Coq` B.2.

6.4.1 Terminaison des fonctions de transformations

Comme déjà expliqué la définition des fonctions non récursives ne pose pas de problèmes. Il faut juste la conformer avec la syntaxe de coq. Dans une première tentative ,nous avons repris le même algorithme qu'en `Ocaml` avec la syntaxe de `Coq` de la fonction `moveOutF` A.4.

```
Fixpoint moveOutF(x: bloc): bloc :=
   match (reste x) with
          Parallele x1 x2 =>
                  if(list_var_eq_dec (interListe (linkedVars x) (freeVars x1)) nil)
                  then
                  match list_var_eq_dec (linkedVars x) nil with
                       left h => descendVar x
                     | right h => moveOutF (descendVar x)
                     end
                  else x
          | _ => x
          end.
```

L'algorithme avec cette écriture déclenche une erreur "Recursive definition of moveOutF is ill-formed". Ceci est du au fait que l'appel récursif ne se fait pas sur un sous-terme de `x`. Donc la définition est rejetée par `Coq`. La méthode la plus simple pour procéder dans ce cas est de choisir une mesure pour le terme de la fonction et prouver qu'il décroît à l'appel récursif. Cela permet de convaincre `Coq` que la fonction va terminer. La première étape à faire est d'établir une mesure pour le bloc. Pour cela, nous avons défini une fonction `sizeB`.

```
Fixpoint sizeB(b: bloc): nat :=
match b with
      BlocES a id    => 0
    | Parallele a b => 0
    | BlocESV l b  =>1 + length l + sizeB b
end.
```

La fonction n'est pas définie simplement par le nombre de variables locales, pour traiter le cas exceptionnel d'une liste vide de variables locales. Nous avons maintenant la mesure, nous pouvons définir notre fonction avec la notation `Program` [10] comme suit :

```
Program Fixpoint moveOutF(x: bloc){measure  sizeB x}: bloc :=
  match (reste x) with
          Parallele x1 x2 =>
                  if(list_var_eq_dec (interListe (linkedVars x) (freeVars x1)) nil)
                  then
                  match list_var_eq_dec (linkedVars x) nil with
                        left h => descendVar x
                      | right h => moveOutF (descendVar x)
                      end
                  else x
          | _ => x
          end.
```

Une fois la fonction passée dans le compilateur, `Coq` nous génère donc une obligation de preuve :

```
1 subgoal
name : Set
var : Set
var_eq_dec : forall v1 v2 : var, {v1 = v2} + {v1 <> v2}
name_eq_dec : forall v1 v2 : name, {v1 = v2} + {v1 <> v2}
x : bloc
moveOutF : {x' : bloc | sizeB x' < sizeB x} -> bloc
x1 : bloc
x2 : bloc
Heq_anonymous : Parallele x1 x2 = reste x
H : length (interListe (linkedVars x) (freeVars x1)) = 0
h : length (linkedVars x) <> 0
Heq_anonymous0 : Utils.in_right = eq_nat_dec (length (linkedVars x)) 0
____________________________________(1/1)
sizeB (descendVar x) < sizeB x
```

Nous devons ici prouver que la mesure spécifiée décroît à chaque appel récursif. Notre preuve est la suivante :

```
Next Obligation.
 generalize dependent x; intro x; case x; clear x; simpl; intros.
 destruct h; auto.
 rewrite <- Heq_anonymous.
 generalize dependent l; intro l; case l; clear l; simpl; intros; auto.
 generalize dependent x2; intro x2; case x2; clear x2; simpl; intros; auto.
 destruct h;auto.
Qed.
```

Comme notre but est de prouver la correction de nos fonctions, nous avons intérêt à les simplifier et de les faire d'une façon générique pour avoir à prouver un minimum de fonctions. Ici nous avons défini une fonction `repeat` que nous avons utilisé pour réécrire toutes nos fonctions récursives. La fonction `repeat` a comme paramètres une fonction `f` à répéter, une mesure `m`, une condition de récursion `c` et l'hypothèse que le paramètre décroît à chaque appel récursif `h`. La définition de la fonction `repeat` est la suivante :

```
Section Repeat.
Variable bloc: Set.
Variable f: bloc->bloc.
Variable m: bloc->nat.
Variable c: bloc->bool.
Variable h: forall x, if c x then m (f x)<m x else True.

Program Fixpoint repeat (x: bloc){measure  m x}: bloc :=
  match (c x) with
     true=>repeat (f x)
   | false=>x
  end.
```

Comme expliqué précedemment cette définition génère une obligation à prouver en relation avec la taille de l'argument de l'appel récursif. Cet Obligation était prouvé comme suit :

```
Next Obligation.
 generalize (h x).
 revert Heq_anonymous.
```

```
  case (c x); intros; auto.
  inversion Heq_anonymous.
Defined.
```

Une fois la fonction `repeat` définie, la fonction `moveOutF` peut être réécrit simplement comme suit :

```
Definition moveOutF: bloc->bloc := repeat _ descendVar sizeB appliDescendVar h.
```

Où la fonction `appliDescendVar` retourne vrai si `descendVar` peut être répétée et faux dans le cas contraire. Sa définition est la suivante :

```
 Definition appliDescendVar(x: bloc): bool:=
match (reste x) with
| Parallele x1 x2 => if(eq_nat_dec (length (interListe (linkedVars x) (freeVars x1))) 0)
                          then match eq_nat_dec (length (linkedVars x)) 0 with
                                    left h => false
                                    | right h => true
                                     end
                          else false
| _ => false
end.
```

`h` est l'hypothèse que la mesure `sizeB` décroît si nous appliquons `descendVar`. La définition ainsi que la preuve de `h` est la suivante :

```
Theorem h: forall x, if appliDescendVar x then sizeB (descendVar x)<sizeB x else True.
Proof.
  intro.
  unfold  appliDescendVar; intros; auto.
  generalize dependent x; intro x; case x; clear x; simpl; intros;auto.
  generalize dependent b; intro b; case b; clear b; simpl; intros; auto.
  generalize dependent l; intro l; case l; clear l; simpl; intros;auto.
  case (In_dec var_eq_dec v (freeVars b)); intros; auto.
  case (eq_nat_dec (length (v :: interListe l (freeVars b))) 0); intros; auto.
  inversion e.
  simpl.
  case (eq_nat_dec (length (interListe l (freeVars b))) 0); intros; auto.
Qed.
```

De la même manière les autres fonctions récursives sont implémentées. Toutes les fonctions sont listées dans B.6.

6.5 Expression de la sémantique d'un bloc

Une interprétation est une fonction qui associe à chaque variable une valeur dans un domaine `d`.

```
Variable d: Set.  (*domaine semantique*)
Definition interV :=var->d. (*associer a chaque variable une valeur*)
```

`interB` permet de donner pour chaque bloc de base une interprétation.

```
Definition interB :=name->interV->Prop.
Variable BBSem: interB.
```

Toutes les interprétations obtenues en changeant les valeurs associées pour des variables qui ne sont pas dans l'interface du bloc par une interprétation `Sem` sont des interprétations pour ce bloc. Ce qui est exprimé dans L'hypothèse `BBSemOk`.

```
Variable BBSemOk: forall n x Sem v BBSem, ~In x (BBInterf n) -> (BBSem n (update Sem x v)
    <-> BBSem n Sem).
```

`BBInterf` permet de donner pour chaque nom du bloc de base son interface.

```
Variable BBInterf: name->list var. (*interface des blocs de base*)
```

L'expression de la sémantique d'un bloc sera écrite sous forme de règles d'inférence. La notation `interp D Sem BBSem B`, signifie que `Sem` est une interprétation du bloc `B` dans le domaine `D` avec `BBSem` la fonction qui donne les interprétations des blocs de bases. La règle `interpBES` 6.5 permet de donner la sémantique d'un bloc élémentaire. `Sem` est une interprétation pour le bloc élémentaire si elle est équivalente à l'interprétation donné par `BBSem` pour le même bloc dans le domaine `D`.

$$\frac{\text{BBSem name Sem}}{\text{Interp D Sem BBSem b}}\ interBES$$

Les deux règles `interpBESVNil` et `interpBESVCons` permettent de traiter le cas d'un bloc avec des variables locales. `Sem` est une interprétation pour un `BlocESV` avec une liste vide de variables locale si `Sem` est une interprétation de son bloc `b` interne.

$$\frac{\text{Interp D Sem BBSem b}}{\text{Interp D Sem BBSem (var } .b)}\ interpBESVNil$$

`Sem` est une interprétation pour un bloc `BlocESV` avec une liste non vide de variables locales, s'il existe une valeur que l'interprétation peut donner pour la première variable et qui rend l'interprétation valable pour le reste du bloc.

$$\frac{\exists v.\text{Interp D (Sem}(x := v)\text{) BBSem (var } x_1x_2...x_n.B)}{\text{Interp D Sem BBSem (var } x_1x_2...x_n.B)}\ interpBESVCons$$

Une interprétation `Sem` pour le premier et le deuxième paramètre d'un bloc parallèle est une interprétation pour le bloc parallèle.

$$\frac{\text{Interp D Sem BBSem B1} \wedge \text{Interp D Sem BBSem B2}}{\text{Interp D Sem BBSem (B1 || B2)}}\ interpPar$$

Cet ensemble de règles peut être écrit en `Coq` comme suit :

```
Inductive Interp: (interV)->(interB)->bloc->Prop :=
  InterpPar: forall Sem b1 b2 BBSem, Interp Sem BBSem b1 -> Interp Sem  BBSem b2 ->
      Interp Sem BBSem (Parallele b1 b2)
  | interpBESVNil: forall Sem b BBSem, Interp Sem BBSem b -> Interp Sem BBSem (BlocESV
      nil  b)
  | interpBESVCons: forall Sem b L x xd BBSem, Interp (update Sem x xd) BBSem (BlocESV L
       b) -> Interp Sem BBSem (BlocESV (x::L)  b)
  | interpBES: forall (BBSem: interB) (Sem: interV)  id n, BBSem n Sem -> Interp Sem
      BBSem (BlocES n id).
```

6.5.1 Règles de réécriture pour Interp

Plusieurs règles de réécritures étaient définies aussi sous forme de `Theorems` et démontrés pour faciliter la preuve de conservation de sémantique par les transformations. Nous présentons ici l'ensemble de règles qui étaient vraiment utilisés dans nos preuves.

`BESVRe` : (Interp D env BBSem (var $x_1 x_2 ... x_n.B$)) $\leftrightarrow$ (Interp D env BBSem (var x_1.var $x_2 ... x_n.B$))

`BESVNilRe` : (Interp D Sem BBSem (var $.B$)) $\leftrightarrow$ (Interp D Sem BBSem B)

`BESVReG` : (Interp D env BBSem (var $(xs@ys).B$)) $\leftrightarrow$ (Interp D env BBSem (var (xs).(var $ys.B$))).

`inverUpdate` : $x1 \neq x2 \rightarrow$ ($update$($update$ env x2 v2)x1 v1) $\leftrightarrow$ $update$($update$ env x1 v1)x2 v2).

`UpdateUpdate` : $update$($update$env x v2)x v1 $\leftrightarrow$ $update$env x v1.

`duplicateDiff` : $duplicate(x1 :: x2 :: l) \leftrightarrow true- > x2 \neq x1$.

`inver12V` : $(x1 \neq x2) \rightarrow$ (Interp D env BBSem (var $x_1 x_2 ... x_n.b$) $\leftrightarrow$ Interp D env BBSem (var $x_2 x_1 ... x_n b$)).

`DelFree` : $v \notin$ (freeVars b) $\rightarrow$ (Interp D ($update$env v xd)BBSem $b \leftrightarrow$ Interp D env BBSem b).

`DisVarsL` : $v \notin$ (freevars b) $\rightarrow$ ((Interp D env BBSem (b || (var $v.b1$))) $\rightarrow$ Interp D env BBSem (var v.(b || b1))).

Toutes ces règles étaient prouvées avant d'être utilisées pour la preuve du reste des fonctions. Pour l'exemple, nous présentons la preuve de la première règle de reecriture d'un `BlocESV` :

```
Theorem BESVRe: forall a l B env BBSem,(Interp env BBSem (BlocESV (a::l) B)) <-> (Interp
    env BBSem (BlocESV (a::nil)  (BlocESV l B))).
Proof.
  intros; split; intros.
  inversion_clear H.
  eapply inter_bESVCons.
  apply  inter_bESVNil.
  apply H0.
  inversion_clear H.
  eapply inter_bESVCons with (xd:=xd).
  inversion_clear H0.
  apply H.
Qed.
```

6.6 Vérification de la préservation de sémantique

6.6.1 Vérification de la correction des fonctions élémentaires

Nous introduisons la notion de transformation `correcte`. Une fonction de transformation est correcte si la sémantique après application de la fonction est préservée. En d'autre terme, si n'importe quelle interprétation du bloc avant transformation est une interprétation du bloc après transformation alors la transformation est correcte.

En Coq, notre fonction correcte est la suivante :

```
Definition correct (f: bloc->bloc):=forall env b,(verifDoubB b)=true->
((Interp env BBSem b<->Interp env BBSem (f b)) /\ (verifDoubB (f b))=true) .
```

Dans ce stade, nous devons démontrer que toutes nos fonctions de transformation sont correctes. Comme nous avons décomposé les transformations le maximum possible en transformations élémentaires, nous commençons par démontrer la correction des fonctions élémentaires. La preuve que swap 4.1.1 est une transformation correcte est la suivante :

```
Theorem SwapCorrect: correct (swap).
Proof.
  unfold correct.
  intros; split; intros.
  revert H; case b; simpl; split; intros; auto.
  inversion_clear H0.
  apply InterpPar; auto.
  inversion_clear H0.
  apply InterpPar; auto.
  revert H; case b; simpl; intros; auto.
  revert H; case (verifDoubB b0); case (verifDoubB b1); intuition.
Qed.
```

De la même façon, nous avons prouvé la correction de toutes les fonctions élémentaires. D'autres règles étaient définies pour permettre d'utiliser les fonctions correctes déjà prouvées pour prouver le reste des fonctions de transformation. La première règle est comCorrect qui dit que si deux fonctions f1 et f2 sont correctes alors la fonction obtenue en composant ces deux fonctions est correcte. Ce qui est décrit dans la règle comCorrect.

$$\frac{(\text{correct f1}) \wedge (\text{correct f2})}{\text{correct (var } x.f1(\text{f2 x}))}\ comCorrect$$

Sa définition ainsi que sa preuve en Coq est la suivante :

```
Theorem comCorrect: forall f1 f2, correct f1-> correct f2-> correct (fun b => f1 (f2 b)).
Proof.
  intros.
  unfold correct.
  intros.
  split.
  eapply iff_trans.
  apply (proj1 (H0 env _ H1)).
  generalize (proj2 (H0 env _ H1)); intros.
  apply (proj1 (H env _ H2)).
  generalize (proj2 (H0 env _ H1)); intros.
  generalize (proj2 (H env _ H2)); intros.
  auto.
Qed.
```

La règle ifCorrect exprime qu'une instruction if-then-else est correcte si ses deux branches sont correctes.

$$\frac{(\text{correct br1}) \wedge (\text{correct br2})}{\text{correct (if c then br1 else br2)}}\ ifCorrect$$

Toutes les règles que nous avons défini étaient prouvées B.8.

6.6.2 Vérification de la correction des transformations

Une fois que nous avons démontré la correction de toutes les fonctions, nous les avons utilisé pour prouver la correction de `permute1` et `permute2` qui sont utilisées pour prouver la préservation de la sémantique par la fonction `moveOut` et enfin nous avons réussi à prouver la correction de la fonction `MoveOutFv1` comme suit :

```
Theorem MoveOutFv1Correct: forall id, correct (moveOutFv1 id).

Proof.
  intro.
  unfold moveOutFv1.
  apply comCorrect.
  unfold moveOutF1.
  apply (repeatCorrect).
  apply descendreVarsCorrect.
  unfold moveOutI.
  apply comCorrect.
  apply applyInsCorrect.
  apply moveOutCorrect.
  apply idCorrect.

Qed.
```

Conclusion Nous avons obtenu grâce à `Coq` la garantie de terminaison de nos fonctions de transformations ainsi que la garantie de la préservation de la sémantique pour la transformation `Move d'un Bloc`. Nous avions pas assez de temps durant ce stage pour prouver le reste des transformations que nous avons proposé dans 4. Parmi les points qui restent aussi à traiter et qui peuvent construire le sujet d'un futur travail : la vérification des transformations vis-à-vis de la spécification (la vérification qu'une transformation fait exactement ce qu'elle est censée faire) et le traitement des fonctions.

Chapitre 7

Conclusion

Ce stage a été l'occasion pour moi de découvrir le domaine des méthodes formelles, ses outils et ses théories, et aussi d'approfondir ma compréhension générale de l'informatique. En passant de la formalisation en `Ocaml`, au programme `Java`, à la preuve en `Coq`, j'ai beaucoup apprécié la diversité des sujets abordés.

`Coq` est un langage qui demande énormément de prérequis pour être utilisé, et sur lequel on est souvent bloqué au début : il faut de la patience pour avancer. Cependant, une fois qu'on a compris ses principes et sa syntaxe, on ne peut qu'apprécier sa puissance et son exactitude.

Nous avons au cours de ce stage comme au long de ce rapport, suivi un processus de génie logiciel. De la conception d'algorithme de transformation aux preuves de la conservation de sémantique en passant par la preuve de terminaison. Ce processus nous offre plusieurs garanties. La généricité de nos algorithmes de transformation garantit la réutilisabilité et la pérennité du code. Le typage dur statique et l'utilisation d'un modèle de mémoire sûr (pas de pointeurs, etc), garantit contre les plantages et les failles de sécurité (segfaults, buffer overflows, ...). La preuve de terminaison garantit que le programme s'arrête (contre les plantages de type boucle infinie). le preuve de la correction garantit que la transformation ne change pas la sémantique d'un bloc.

J'avais hésité avant de choisir un thème pour mon projet de Master, je pense aujourd'hui que c'était le bon choix. Le sujet de la vérification des transformations avec un assistant de preuve s'est révélé très enrichissant, et je suis convaincue que ce travail constitue une expérience inestimable autant en mathématique qu'en informatique. J'ai le sentiment, grâce à ce stage, d'avoir donc pu franchir la barrière d'entrée de `Coq`.

D'un point de vue éducatif donc, j'ai trouvé ce stage profondément enrichissant et très intéressant. D'un point de vue plus personnel, j'anticipais ce stage pour découvrir un peu le milieu de la recherche dans le domaine des méthodes de formelles et la preuve des programmes, puisque j'hésite toujours dans mon orientation.

J'ai apprécié la qualité des discussions et la convivialité au sein de l'équipe `ACADIE`. Je tiens à remercier particulièrement M. Strecker, M.Bodeveix et M.Filali de m'avoir donner l'opportunité de réaliser ce stage, mais aussi et surtout pour leur aide et leurs disponibilité et leurs nombreuses réponses. J'exprime ma gratitude envers mes enseignants de l'université de Boumerdès (Algérie). Mon université d'origine où j'ai passé la plus grande partie de ma vie universitaire et j'ai appris les bases de l'informatique. En conclusion, je suis très satisfaite du déroulement de ce stage, pour avoir découvert, appris et compris et ensuite certifier la première transformation.

Annexe A

Code Ocaml

A.1 Module EXList

```
module ExList=struct
  (*fonction qui teste si une liste l1 est incluse dans l2*)
  let rec inclus l1 l2 = match l1 with
      []-> true
    | x::q1-> if List.mem x l2 then inclus q1 l2 else false;;

  (*fonction qui supprime une liste l2 d'une liste l1*)
  let rec removeL l1 l2 = match l1 with
      [] ->[]
    |x1::q1 -> if List.mem x1 l2 then removeL q1 l2 else x1::(removeL q1 l2)

  (*union de deux listes*)
  let rec union l1 l2 = match l1 with
      [] ->l2
    | x::q1 -> if List.mem x l2 then union q1 l2 else x::(union q1 l2);;

  (*l'intersection de deux listes*)
  let rec interListe l1 l2=List.filter(fun x -> List.mem x (l2)) l1

  (*verifier si un element est definit deux fois*)
  let rec duplicate l=match l with(*verifier si une liste contient des doublons*)
      []->true
    |x::l1->if List.mem x l1 then false else duplicate l1
end
```

A.2 Déclaration des types

```
(*type des noms et variables de blocs*)
type name = string
type var = string

(*declaration du type bloc*)
type bloc =
    BlocES of name * int * var list          (*Appel de fonction*)
  | BlocESV of var list * bloc               (*declaration de variables locales*)
  | Parallele of bloc * bloc;;               (*composition de blocs*)

(*Definition du type definitionBloc*)
type definitionBloc=
```

```
    Fonction of name * var list * bloc
  | BoiteNoire of name * var list

(*Definition du type environnement*)
type environnement =  definitionBloc list
```

A.3 La signature BnOp

```
module type BnOp = sig
  type t
  type f
(*Operations sur les blocs*)
  val is_o : t->bool
  val p1 : t->t
  val p2 : t->t
  val eq_id : int->t->bool
  val shiftRight : t->t            (*((a||b)||c)  ->  (a||(b||c))*)
  val shiftLeft : t->t             (*(a||(b||c))  ->  ((a||b)||c)*)
  val swap : t->t                  (*a||b  ->   b||a*)
  val linkedVars : t-> string list
  val freeVars : t -> string list
  val reste : t->t                 (*BlocESV(l,b)  ->  b*)
  val appl1: (t->t)->t->t          (*(a||b)  ->  (f(a)||b)*)
  val appl2 : (t->t)->t->t         (*(a||b)  ->  (a||f(b))*)
  val descendVar : t->t            (*\x1x2...xn. (a||b)  ->  \x2...xn. a||(\x1. b)*)
  val keepVar : t->t               (*\x1x2...xn. (a||b)  ->  \x2...xnx1. (a||b))*)
  val applyInside : (t->t)->t->t   (*\x1x2...xn. b  -> \x1x2...xn. f(b)*)


(*Operation sur les fonction*)
  val funName : f->string
  val funPar :f->string list
  val funCorps : f->t option
  val arite : f->int
  val app_f : (t->t)->f-> f
  val varLG : f->f
  val moveB1 : f->f
  val add : f->f list
  val fusionner : f->f->f
end
```

A.4 Le module MoveOutFirst paramétré par BnOp

```
module MoveOutFirst(Bo:BnOp) = struct
  open Bo

(*surgir le bloc d'identifiant id s'il est dans le bloc b1*)
(*ex: permute1 idC x*)
(*(C||b21)||b2 -> C||(b21||b2)*)
(*(b11||C)||b2 -> (C||b11)||b2 *)
 let permute1 id x=
   if is_o x
   then let b1,b2=(Bo.p1 x, Bo.p2 x) in
     if is_o b1
     then let b11,b12 = (Bo.p1 b1, Bo.p2 b1) in
       if eq_id id b11
       then Some (shiftRight x)
       else
         if eq_id id b12
         then Some (shiftRight(appl1 swap x))
```

```
          else None
      else
        if eq_id id b1
        then Some x
        else None
    else
      if eq_id id x
      then Some x
      else None

(*surgir le bloc d'identifiant id s'il est dans le bloc b2*)
(*ex: permute2 idC x*)
(*b1||(C||b22) -> (C||b22)||b1) -> (C||(b22||b1)) -> (C||(b1||b22)*)
(*(b1||(b22||C)) -> (b1||(C||b22)) -> (C||(b22||b1)) -> (C||(b1||b22)*)
 let  permute2 id x=
   if is_o x then let b1,b2=(Bo.p1 x, Bo.p2 x) in
     if is_o b2
     then
       let b11,b12 = (Bo.p1 b2, Bo.p2 b2) in
         if eq_id id b11
         then appl2 swap (shiftRight(swap x))
         else
           if eq_id id b12
           then appl2 swap (shiftRight(swap (appl2 swap x)))
           else x
     else
       if eq_id id b2
       then swap x
       else x
   else x

(*appliquer recursivement permute1 et permute2*)

 let rec moveOut id x= (*ex: moveOut idC (A||B||C) -> C||(B||C)*)
   if is_o x
   then
     match (permute1 id (appl1 (moveOut id) (appl2 (moveOut id) x))) with
         Some c->c
       |None-> permute2 id x(*si permute1 n'est pas faite*)
   else x


(*Sortir un bloc libre.  ex: \xyz.(A(t) || B(x,y,z)) -> A(t) || (\xyz. B(x,y,z))*)

(*premiere etape*)
(*ex: moveOut id (\xy. (A||b||c)) -> \xy. moveOut id (A||b||c)*)
 let moveOutI id x=applyInside (moveOut id) x

(*deuxieme etape*)
(*ex: \xy. (A(z,t) || B(x,y))*)
(*ex: \xy. (A(z,t)||B) -> moveOutF(\y. A(z,t)||(\x. B))
(appel recursif jusqu'a descendVar tous les variables)*)
 let rec moveOutF x=
   if is_o (reste x)
   then let x1= (Bo.p1 (reste x)) in
     if interListe (linkedVars x) (freeVars x1)=[]
     then
       if (linkedVars x)=[] then descendVar x
       else moveOutF(descendVar x)
     else x (*ex: \xy. (A(z,y) || B)*)
   else x

(*application des deux etapes*)
 let rec moveOutFb id x=moveOutF (moveOutI id x)


(*Sortir n'importe quel bloc*)

(*ex: \xyz. A(x,z,t)||B  ->  ([x,z])*)
```

```
 let getVars1 x=List.filter(fun y->List.mem y (freeVars (p1 (reste x)))) (linkedVars x)

(*ex: \xyz. A(x,g,t)||B -> moveOutF1(\yzx. A(x,g,t)||B)*)
(*ex: \yzx. A(x,g,t)||B -> moveOutF1(\zx. A(x,g,t)||(\y.B))*)
(*ex: \xy. A(x,y,t)||B -> \xy. A(x,y,t)*)
 let rec moveOutF1 x=
   if is_o (reste x)
   then
     let l1=(getVars1 x) in
       if l1=[]
       then moveOutF x
       else
         if ((l1=linkedVars x)) then x
         else
           match (linkedVars x) with
             a::l -> if List.mem a (freeVars (p1 (reste x)))
             then moveOutF1 (keepVar x)
             else moveOutF1 (descendVar x)
             | [] -> descendVar x
   else x

(*appliquer la fonction moveOutF1 apres l'application de la fonction moveOutI*)
 let moveOutFv1 id x=moveOutF1(moveOutI id x)


(*Transformations sur les fonctions*)

(*sortir un bloc de la definition*)
(*premiere etape: move le bloc dans le corp de la fonction*)
(*F1(x,y){A||B||C} -> F1(x,y){C||A||B}*)
let moveOutInF id f =app_f (moveOutFv1 id) f

(*deuxieme etape: ajouter les variables liees du bloc aux parametres de la fonction*)
(*F1(x,y){\t.C||A||B} -> newF1(x,y,z){A||B}*)
let generalise f=match (funCorps f) with
    Some b->moveB1(varLG f)
  |_->f (*cas d'une boite noire*)


(*optimiser la definition d'un bloc*)
(*ex: newF1(x,y,z){A(x)||B(z)} -> newF1(x,z){A(x)||B(z)} *)
let moveFun id f=generalise(moveOutInF id f)


(*fusionner deux definitions*)
let merge f1 f2=fusionner f1 f2


(*ajouter la fonction a env*)
let addFun f=add f
end
```

A.5 L'implementation du module BnOp

```
module Par= struct
  type t = bloc

  type f = definitionBloc

  let is_o = function
      Parallele(_,_) -> true
    |_ -> false

  let p1=function
      Parallele(a,b) -> a
```

```
    |b->b

  let p2=function
      Parallele(a,b) ->b
    |b->b

  let eq_id id=function
      BlocES(_,i,_) -> id=i
    |_->false

  let shiftRight=function
      Parallele(Parallele(a,b),c)->Parallele(a, Parallele(b,c))
    |b->b

  let shiftLeft=function
      Parallele(a, Parallele(b,c))->Parallele(Parallele(a,b),c)
      |b->b

  let swap=function
      Parallele(a,b)->Parallele(b,a)
    |b->b

  let linkedVars=function
      BlocESV(l,_)->l
    |_->[]

  let rec freeVars=function
      BlocES(a,id,l) -> l
    | Parallele(a,b) -> union (freeVars a) (freeVars b)
    | BlocESV(l,b) -> removeL (freeVars b) l

  let reste=function
      BlocESV(l,b1)->b1
    |b->b

  let appl1 f=function
      Parallele(a,b)->Parallele(f(a),b)
    |b->b

  let appl2 f=function
      Parallele(a,b)->Parallele(a,f(b))
    |b->b

  let applyInside f=function
      BlocESV(l,b1)->BlocESV(l,f(b1))
    |b->b

(*ex: \xyz. A||B -> \yz. A||(\x. B)*)
  (*ne peut etre appliquee que apres verification dans la fonction
    moveOutF ou moveOutF1 que la variable a n'est pas utillisee dans le
    bloc A*)
 let descendVar=function
  BlocESV(a::l,Parallele(b1,BlocESV(l2,b2)))->BlocESV(l,Parallele(b1,BlocESV(a::l2,b2)))
   |BlocESV(a::l,Parallele(b1,b2)) -> BlocESV(l,Parallele(b1,BlocESV([a],b2)))
   |BlocESV([],b)->b
   |b->b

(*ex: \xyz. A||B -> \yzx. A||B*)
  let keepVar=function
      BlocESV(a::l1,Parallele(b1,b2)) -> BlocESV(l1@[a],Parallele(b1,b2))
      |b->b

(*Fonctions pour les fonctions*)

(*Nom d'une fonction*)
  let funName=function
      Fonction(n,_,_) -> n
    | BoiteNoire(n,_) ->n
```

```
(*Arite d'une fonction*)
  let arite =function
      Fonction(n,l,b) -> List.length l
    |  BoiteNoire(n,l) -> List.length l

(*Parametres d'une fonction*)
  let funPar=function
      Fonction(_,l,_) -> l
    | BoiteNoire(_,l) ->l

(*Corps d'une fonction*)
  let funCorps =function
      Fonction(_,_,b) -> Some b
    | BoiteNoire(_,l) ->None


(*Applique une fonction au corps d'une fonction*)
(*ex: F(...){B} -> F(...){f(B)}*)
  let app_f f= function
      Fonction(n,l,b) -> Fonction(n,l,f(b))
    | b ->b (*BoiteNoire*)

(*ajouter les variables locales du corps aux variables globales*)
(*ex: F(x,y,z){\t. C(a,b,t)||B} -> F1(x,y,z,t){B}*)
  let varLG=function
      Fonction(n,l,BlocESV(l1,b)) -> Fonction("new" ^ n,union l l1,b)
    |b->b(*la fonction ne s'appelle pas dans ce cas*)

(*Supprimer le premier bloc du corps*)
(*ex: F(x,y,z){\t. C(a,b,t)||B} -> F1(x,y,z,t){B}*)
  let moveB1=function
      Fonction(n,l,BlocESV(l1,Parallele(b1,b2))) -> Fonction(n,l,BlocESV(l1,b2))
    |b->b

(*Optimiser la definition d'une fonction*)
 let opt=function
      Fonction(n,l,b) -> Fonction(n,freeVars b,b)
    |b->b

(*Ajouter une nouvelle definition d'une fonction a l'environnement*)
  let add f=f::env


(*Les variables liees au reste de blocs dans db (sauf b)*)
 let rec varsReste b db=match db with
          Parallele(b1, b2)->if b1=b then varsReste b b2 else if b2=b
          then varsReste b b1 else ExList.union (varsReste b b1) (varsReste b b2)
    | BlocESV(l, b1)->if b1=b then [] else varsReste b b1
    | BlocES(_,_,_) as b1->if(b1=b) then [] else freeVars b1

(*les ports de connexion d'un bloc dans une definition*)
let connectable b d=match funCorps d with
    None-> None
  | Some c-> if (not(ExList.inclus (freeVars b) (funPar d)))(*il y a des ports locals
      lies au bloc*)
    then if(not(ExList.inclus  (freeVars b) (varsReste b c)))
    then Some(ExList.removeL (freeVars b) (ExList.interListe (varsReste b c) (freeVars b)
        ))
    else None else None

end
```

Annexe B

Code Coq

B.1 Définition des types et des hypothèses

```
Require Import List.
Require Import Arith.
Require Import Ascii.
Require Import FunctionalExtensionality.
Require Import Wf.
Require Import Program.
Require Import Classical.

(* ================================================================ *)

Section Bloc.
(*** Types et Hypotheses ***)
Variable name:Set. (* nom des blocs de base *)
Variable var:Set.      (*port d'un bloc*)
Variable BBInterf: name->list var. (*interface des blocs de base*)
Variable var_eq_dec: forall v1 v2: var, {v1=v2}+{v1<>v2}. (*egalite variables decidable*)
Variable name_eq_dec: forall v1 v2: name, {v1=v2}+{v1<>v2}. (*egalite noms decidable*)
Variable list_var_eq_dec:forall L1 L2:list var,{L1=L2}+{L1<>L2}. (*egalite listes dec*)

(** Type bloc **)
Inductive bloc:Set :=
    BlocES : name ->nat->bloc          (* fonction*)
  | BlocESV :list var->bloc->bloc       (*declaration de variables locales*)
  | Parallele : bloc ->bloc->bloc.        (*composition de blocs*)

(** Type definitionBloc **)
Inductive definitionBloc:Set :=
    Fonction: name->list var->bloc->definitionBloc
  | BoiteNoire: name-> list var->definitionBloc.

(** Type environnement **)
Definition environnement:Set :=list definitionBloc.
```

B.2 Fonctions sur les listes

```
(** Fonction qui teste si une liste l1 est incluse dans l2 **)
Fixpoint inclus(l1:list var) (l2:list var){struct l1}: (bool) :=
match l1 with
  nil=> true
```

```
 | x::q1=> if (In_dec  var_eq_dec x l2) then (inclus q1 l2) else false
 end.

(** Fonction qui supprime une liste l2 d'une liste l1 **)
Fixpoint removeL (l1:list var) (l2:list var){struct l1}: (list var) :=
match l1 with
  nil =>nil
  |x1::q1 => if (In_dec  var_eq_dec x1 l2) then (removeL q1 l2) else x1::(removeL q1 l2)
  end.

(** Eliminer les doublons dans une liste **)
Fixpoint elimDoubL (l: list var):(list var):=
match l with
 nil => nil
 |a::l=> if(In_dec  var_eq_dec a l) then elimDoubL l else a:: (elimDoubL l)
end.

 (** Intersection de deux listes **)
Fixpoint interListe(l1:list var) (l2:list var){struct l1}: (list var) :=
match l1 with
 nil => nil
 | x :: l=>if (In_dec  var_eq_dec x l2) then x::(interListe l l2) else (interListe l l2)
 end.

(** Verifier si un element est defini deux fois **)
Fixpoint duplicate(l:list var): bool :=
match l with(*verifier si une liste contient des doublons*)
 nil=>true
 |x::l1=>if(In_dec  var_eq_dec x l1) then false else duplicate l1
 end.

(** Proprietes de duplicate **)
Lemma duplicateCons: forall x l, duplicate (x::l)=true -> duplicate l=true.
Proof.
 simpl; intros; auto.
 revert H; case (In_dec var_eq_dec x l); auto; intros.
 discriminate H.
Qed.

Lemma duplicateCons1:forall x1 x2 l, duplicate (x1::x2::l)=true->duplicate (x1::l)=true.
Proof.
 simpl; intros; auto.
 revert H; case (var_eq_dec x2 x1); intros; auto.
 discriminate H.
 revert H; case (In_dec var_eq_dec x1 l); intros; auto.
 revert H; case (In_dec var_eq_dec x2 l); intros; auto.
 discriminate H.
Qed.
```

B.3 Operations sur les blocs

```
(** Verifier si un bloc est un Parallele **)
Definition is_o (b:bloc):(bool) :=
match b with
      Parallele a b => true
    |_ => false
end.

(** Extraire le premier argument d'un Parallele **)
Definition p1(b:bloc):(bloc):=
match b with
      Parallele a b  => a
    |a=>a
end.
```

```
(** Extraire le deuxieme argument d'un Parallele **)
Definition p2(b:bloc):(bloc):=
match b with
      Parallele a b  => b
    |b=>b
end.

Definition eq_id(id:nat)(x:bloc):(bool):=
match x with
    BlocES a i =>if (eq_nat_dec i id)  then true else false
   |_=>false
end.

(** ((a||b)||c)  ->  (a||(b||c)) **)
Definition shiftRight(x:bloc):(bloc):=
match x with
      Parallele (Parallele a b)  c => Parallele a (Parallele b c)
    |b=>b
end.

(** (a||(b||c))  ->  ((a||b)||c) **)
Definition shiftLeft(x:bloc):(bloc):=
match x with
      Parallele a (Parallele b c)=>Parallele (Parallele a b) c
      |b=>b
end.

(** a||b  ->   b||a **)
Definition swap(x:bloc):(bloc):=
match x with
      Parallele a b =>Parallele b a
    |b=>b
end.

(** Extraire les variables liees d'un bloc **)
Definition linkedVars(x:bloc):(list var):=
match x with
      BlocESV l _ =>l
    |_=>nil
end.

(** Extraire les variables libres d'un bloc **)
Fixpoint freeVars (x: bloc) : (list var) :=
match x with
      BlocES a id  => BBInterf a
    | Parallele a b => elimDoubL (app (freeVars a) (freeVars b))
    | BlocESV l b  => removeL (freeVars b) l
end.

(** Extraire le bloc interne d'un bloc **)
Definition reste(x:bloc): bloc:=
match x with
      BlocESV l b1 => b1
    | b => b
end.

(** (a||b)  ->  (f(a)||f(b)) **)
Definition appl12(f: bloc->bloc)(x:bloc): bloc :=
match x with
      Parallele a b => Parallele (f a) (f b)
    |b => b
end.

(** (a||b)  ->  (f(a)||b) **)
Definition appl1(f: bloc->bloc)(x:bloc): bloc :=
match x with
      Parallele a b => Parallele (f a) b
    |b => b
end.
```

```
(** (a||b)  ->  (a||f(b)) **)
Definition appl2(f: bloc->bloc)(x:bloc): bloc:=
match x with
      Parallele a b => Parallele a (f b)
    |b => b
end.

(** \x1x2...xn. b  -> \x1x2...xn. f(b) **)
Definition applyInside(f: bloc->bloc)(x: bloc): bloc :=
match x with
      BlocESV l b1 => BlocESV l (f b1)
    |b => b
end.

(** \x1x2...xn. (a||b)  ->  \x2...xn. a||(\x1. b) **)
Definition descendVar(x: bloc) : bloc :=
match x with
    BlocESV (a::l) (BlocES n id) => BlocESV (l) (BlocESV (a::nil) (BlocES n id))
    |BlocESV (a::l) (BlocESV l1 b ) => BlocESV (l) (BlocESV (a::nil) (BlocESV  l1 b ))
    |BlocESV nil b =>b
    |BlocES n id =>BlocES n id
    |Parallele b1 b2 => Parallele b1 b2
    |BlocESV (a::l) (Parallele b c) => if((In_dec  var_eq_dec a (freeVars b)))
                        then x else BlocESV l (Parallele b (BlocESV (a::nil) c))
end.

(** \x1x2...xn. (a||b)  ->  \x2...xnx1. (a||b)) **)
Definition keepVar(x: bloc) : bloc :=
match x with
      BlocESV (a::l1) (Parallele b1 b2) => BlocESV (app l1 (a::nil)) (Parallele b1 b2)
      |b=>b
end.
```

B.4 Fonctions pour les fonctions

```
(** Nom d'une fonction **)
Definition funName(f:definitionBloc) : name :=
match f with
      Fonction n _ _  => n
    | BoiteNoire n _  =>n
end.

(** Arite d'une fonction **)
Definition arite(f:definitionBloc): nat :=
match f with
      Fonction n l b  => length l
    |  BoiteNoire n l  => length l
end.

(** Parametres d'une fonction **)
Definition funPar(f:definitionBloc): (list var) :=
match f with
      Fonction _ l _  => l
    | BoiteNoire _ l  => l
end.

(** Corps d'une fonction **)
Definition funCorps(f:definitionBloc): (option bloc) :=
match f with
      Fonction _ _ b => Some b
    | BoiteNoire _ l  =>None
end.
```

```
(*ex: F(...){B} -> F(...){f(B)}*)
Definition app_f(d: definitionBloc)(f: bloc->bloc): definitionBloc :=
match d with
      Fonction n l b  => Fonction n l (f b)
    | b =>b (*BoiteNoire*)
end.

(** ajouter les variables locales du corps aux variables globales **)
Definition varLG(d: definitionBloc): definitionBloc :=
match d with
      Fonction n l (BlocESV l1 b) => Fonction (n)  (elimDoubL (app l  l1)) b (*ex: F(x,y,
          z){\t. C(a,b,t)||B} -> F1(x,y,z,t){B}*)
    |b=>b(*la fonction ne s'appelle pas dans ce cas*)
end.

(** Optimiser la definition d'une fonction **)
 Definition opt(d: definitionBloc): definitionBloc :=
match d with
      Fonction n l b  => Fonction n (freeVars b) b
    |b=>b
end.
```

B.5 Fonctions de vérification

```
(** Arite de la fonction a partir de l'environnement **)
Fixpoint getArite(env: environnement)(n: name){struct env}: option nat :=
match env with
  | nil => None
  | x :: tl => if(name_eq_dec (funName x) n) then Some (arite x) else getArite tl n
 end.


(** Verifier que les variables libres sont declarees **)
Fixpoint verifDef(d: definitionBloc): bool :=
match d with
      Fonction n v b => inclus (freeVars b) v
    |_ => true(*cas d'une boite noire*)
    end.

(** Verifier l'arite du bloc dans l'environnement **)
Fixpoint verifArite(env: environnement)(b: bloc): bool :=
match b with
    | Parallele a b => if(verifArite env a) then verifArite env b else false
    | BlocESV l b => verifArite env b
    | BlocES bb id =>match (getArite env bb)  with
                      | Some a =>if(eq_nat_dec  a (length (BBInterf bb))) then true else
                          false
                      | None => true
                      end
                      end.

(** Verifier s'il y a des doublons dans la declaration du bloc **)
Fixpoint verifDoubB(b: bloc): bool :=
match b with
      BlocES a id   => if(duplicate (BBInterf a)) then true else false
    | Parallele a b => if(verifDoubB a) then verifDoubB b else false
    | BlocESV l b  => if(duplicate l) then verifDoubB b else false
end.

(* regle sur verifDoubB *)
Theorem verifDBpar: forall b1 b2, (verifDoubB (Parallele b1 b2))=true -> ((verifDoubB b1)
    =true) /\ ((verifDoubB b2)=true).
Proof.
  simpl.
  intro; intro.
```

```
  case (verifDoubB b1); case (verifDoubB b2); intuition.
Qed.

 (** Verifier s'il y a des doublons dans la definition de fonction **)
 Fixpoint verifDoubD(d: definitionBloc): bool :=
match d with
      Fonction n l b => if(duplicate (elimDoubL (app l (linkedVars b)))) then verifDoubB
          b else false
    |BoiteNoire n l => duplicate l
end.
```

B.6 Fonctions de transformations

```
Definition permute1(id: nat)(x: bloc): bloc :=(*ex: permute1 idC x*)
if is_o x
then
if is_o (p1 x)
     then
       if eq_id id (p1 ( p1 x))
       then (shiftRight x)          (*C||b21)||b2 -> C||(b21||b2)*)
       else
         if eq_id id (p2 (p1 x))
         then (shiftRight (appl1 swap x)) (*(b11||C)||b2*)
         else x
     else x                                (*C||(...) -> C||(...)*)
   else x                                  (*C ->C*).


(** Surgir le bloc d'identifiant id s'il est dans (p2 x) **)
(*ex: permute2 idC x*)
 (*b1||(C||b22) -> (C||b22)||b1) -> (C||(b22||b1)) -> (C||(b1||b22)*)
(*(b1||(b22||C)) -> (b1||(C||b22)) -> (C||(b22||b1)) -> (C||(b1||b22)*)
Definition  permute2(id: nat)(x: bloc) : bloc :=
if is_o x then
     if is_o (p2 x)
     then
         if eq_id id (p1 (p2 x))
         then appl2 swap (shiftRight(swap x))
         else
           if eq_id id (p2 (p2 x))
           then appl2 swap (shiftRight(swap (appl2 swap x)))
           else x
     else
       if eq_id id (p2 x)
       then swap x
       else x
   else x.


(** Appliquer recursivement permute1 et permute2 **)
Fixpoint moveOut(id: nat)(x: bloc){struct x}: bloc := (*ex: moveOut idC (A||B||C) -> C||(
    B||C)*)
  if is_o x
  then permute2 id (permute1 id (appl12 (moveOut id)  x))
  else x.

(*** Sortir un bloc libre.  ex: \xyz.(A(t) || B(x,y,z)) -> A(t) || (\xyz. B(x,y,z)) ***)

(** premiere etape **)
(*moveOut id (\xy. (A||b||c)) -> \xy. moveOut id (A||b||c)*)
 Definition moveOutI(id: nat)(x: bloc): bloc :=applyInside (moveOut id) x.

(** deuxieme etape **)
(* Definition d'une mesure pour le bloc **)
Fixpoint sizeB(b: bloc): nat :=
```

```
match b with
      BlocES a id    => 0
    | Parallele a b => 0
    | BlocESV l b  =>1 + length l + sizeB b
end.
(* Definition d'une fonction repeat *)
Section Repeat.
Variable bloc: Set.
Variable f: bloc->bloc.
Variable m: bloc->nat.
Variable c: bloc->bool.
Variable h: forall x, if c x then m (f x)<m x else True.

Program Fixpoint repeat (x: bloc){measure  m x}: bloc :=
  match (c x) with
     true=>repeat (f x)
   | false=>x
  end.

(** Preuve de l'obligation**)
Next Obligation.
generalize (h x).
revert Heq_anonymous.
case (c x); intros; auto.
inversion Heq_anonymous.
Defined.

(** Reecriture de repeat en match **)
Lemma repeat_eq_match: forall x, repeat x = match (c x) as h return h = c x -> bloc with
        true=> fun _ => repeat (f x)
     | false=> fun _ => x
     end refl.
Proof.
  unfold repeat; intros.
  rewrite fix_measure_sub_eq; simpl; intros; auto.
  generalize (refl: c x0 = c x0).
  pattern (c x0) at 1 3 7.
  apply bool_ind; intros; auto.
Qed.

(** Reecriture de repeat en if then else **)
Theorem repeat_eq: forall x, repeat x = if (c x) then repeat (f x) else x.
Proof.
  intro.
  rewrite (repeat_eq_match x).
  generalize (refl: c x = c x).
  pattern (c x) at 1 3 7.
  apply bool_ind; intros; auto.
Qed.

End Repeat.

(** Condition de repetition de descendVar **)
Definition appliDescendVar(x: bloc): bool:=
match (reste x) with
| Parallele x1 x2 =>
     if(list_var_eq_dec (interListe (linkedVars x) (freeVars x1)) nil)
     then        match eq_nat_dec (length (linkedVars x)) 0 with
         left h => false
       | right h => true
       end
     else false
| _ => false
end.

(** Decroissance de la taille du bloc apres application de descendVar **)
Theorem h: forall x, if appliDescendVar x then sizeB (descendVar x)<sizeB x else True.
Proof.
  intro.
```

```
  unfold  appliDescendVar; intros; auto.
  generalize dependent x; intro x; case x; clear x; simpl; intros;auto.
  generalize dependent b; intro b; case b; clear b; simpl; intros; auto.
  generalize dependent l; intro l; case l; clear l; simpl; intros;auto.
  case (list_var_eq_dec [] []); intros; auto.
  case (In_dec var_eq_dec v (freeVars b)); intros; auto.
  case (list_var_eq_dec (v :: interListe l (freeVars b)) []); intros; auto.
  inversion e.
  case (list_var_eq_dec (interListe l (freeVars b)) []); intros; auto.
  case (list_var_eq_dec [] []); intros; auto.
Qed.

(** Repetition de descendVar **)
Definition moveOutF: bloc->bloc := repeat _ descendVar sizeB appliDescendVar h.

(** application des deux etapes **)
Fixpoint moveOutFb(id: nat)(x: bloc){struct x} : bloc :=moveOutF (moveOutI id x).


(*** Sortir n'importe quel bloc ***)

(** Changer la position de la premiere variable**)
Definition permuteVars (x: bloc):=match x with
  BlocESV (a :: l) b => BlocESV l (BlocESV (a::nil) b)
        | _=>x
        end.
(** Descendre une variable locale**)
Definition descRight(x: bloc):=match x with
  BlocESV (a::nil) (Parallele b1 b2)=> if(In_dec var_eq_dec a (freeVars b1)) then x else
      Parallele b1 (BlocESV (a::nil) b2)
  | _=>x
  end.

(** Diviser la liste des variables locales**)
Definition split1(x: bloc):=match x with
  BlocESV (a :: l) b => BlocESV (a::nil) (BlocESV l b)
        | _=>x
        end.

(** Verifier qu'il y a une variable locale et un bloc interne Parallele**)
Definition OneVarPar(b:bloc): bool:=
  match b with
    BlocESV l b1 => match l with
                                                | (a::nil)=> match b1 with
                                                                Parallele _ _ => true
                                                                | _ => false
                                                                end
                                                | _ =>false
                                                end
    |b=>false
end.

(** Verifier qu'il y a une variable locale et un bloc interne non Parallele**)
Definition OneVarNotPar(b:bloc): bool:=
  match b with
    BlocESV l b1 => match l with
                                                | (a::nil)=> match b1 with
                                                                Parallele _ _ => false
                                                                | _ => true
                                                                end
                                                | _ =>false
                                                end
    |b=>false
end.


(** Verifier qu'il y a au moins deux varibles locales**)
Definition AtListTwoVars(b:bloc): bool:=
  match b with
```

```
    BlocESV l b1 => match l with
                                                  _::_::_ =>true
                                                | _=> false
                                                end
    |b=>false
end.

(** Descendre les variables libres **)
Definition descendreVars(b: bloc): bloc :=
  if (OneVarPar b)  then descRight b
  else if(OneVarNotPar b) then split1 b
  else if(AtListTwoVars b) then permuteVars b
  else b.

(** Condition de repetition de descendreVars **)
Definition AppliDescendreVars(x: bloc): bool:=
match x with
BlocESV (a::nil) b=>match b with
                    Parallele b1 b2=>if(eq_nat_dec 1 (length (linkedVars x)))
                                          then if(In_dec var_eq_dec a (freeVars b1)
                                                )
                                                then false
                                                else true else true
                    |_=> false
                    end
|_=>false
end.

(** Decroissance de la taille du bloc apres application de descendreVars **)
Theorem h1: forall x, if AppliDescendreVars x then sizeB (descendreVars x)<sizeB x else
    True.
Proof.
  simpl; intros.
  unfold AppliDescendreVars; intros; auto.
  case x; intros; auto.
  case l; intros; auto.
  case l0; intros; auto.
  case b; intros; auto.
  case (eq_nat_dec 1 (length (linkedVars (BlocESV (v :: nil) (Parallele b0 b1)))));
      intros.
  case (In_dec var_eq_dec v (freeVars b0)); intros; auto.
  simpl.
  unfold  descendreVars.
  simpl.
  revert n; case (In_dec var_eq_dec v (freeVars b0)); intuition.
  intuition.
Qed.

(** repetition de descendreVars **)
Definition moveOutF1: bloc->bloc := repeat _ descendreVars sizeB AppliDescendreVars h1.
```

B.7 Semantique des blocs

```
Section Semantique.

 (** domaine semantique **)
Variable d: Set.

(** associer a chaque variable une valeur **)
Definition interV :=var->d.

(** mise a jour de la fonction interV **)
Definition update(I: interV)(x: var)(dx: d): interV:=
  fun (z: var) => if (var_eq_dec z x) then dx else (I z).
```

```
(** associer a chaque identifiant de bloc une fonction d'interpretation
 et une proposition les valeurs attribuees par cette fonction **)
Definition interB :=name->interV->Prop.

(** semantique des blocs de bases **)
Variable BBSem: interB.

(** Regle pour la semantique des blocs de bases**)
Variable BBSemOk: forall n x Sem v BBSem,
 ~In x (BBInterf n) -> (BBSem n (update Sem x v) <-> BBSem n Sem).

(** interpretation d'un bloc **)
Inductive Interp: (interV)->(interB)->bloc->Prop :=
  inter_par: forall I b1 b2 BBSem, Interp I BBSem b1 -> Interp I  BBSem b2 -> Interp I
      BBSem (Parallele b1 b2)
  | inter_bESVNil: forall I b BBSem, Interp I BBSem b -> Interp I BBSem (BlocESV nil  b)
  | inter_bESVCons: forall I b L x xd BBSem, Interp (update I x xd) BBSem (BlocESV L b) -
      > Interp I BBSem (BlocESV (x::L)  b)
  | inter_bES: forall (BBSem: interB) (I: interV)  id n, BBSem n I -> Interp I BBSem (
      BlocES n id).

(** regles de reecriture **)

Theorem NotInElim: forall a l, ~In a (elimDoubL l) -> ~In a l.
Proof.
  intros; induction l; intros; auto.
  intro.
  inversion_clear H0.
  apply H; clear H; simpl.
  case (In_dec var_eq_dec a0 l); intros.
  rewrite H1 in i.
  apply NNPP; intro.
  apply (IHl H); auto.
  simpl.
  left; auto.
  simpl in H.
  apply H; clear H.
  case (In_dec var_eq_dec a0 l ); simpl; intros; auto.
  apply NNPP; intro.
  apply (IHl H); auto.
  right; apply NNPP; intro.
  apply (IHl H); auto.
Qed.

Theorem NotInApp: forall l1 l2 (a: var), ~ In a (app l1 l2) -> ~In a l1/\~In a l2.
Proof.
  intros; induction l1; intros; auto.
  split; intros.
  intro.
  apply H; clear H; simpl.
  inversion_clear H0.
  left; auto.
  right.
  apply NNPP; intro.
  elim (IHl1 H0); intros.
  apply H1; auto.
  intro; apply H; clear H; simpl.
  right.
  apply NNPP; intuition.
Qed.

Theorem NQ: forall (a: var) l, a::l=nil-> l=nil.
Proof.
  intros.
  inversion H.
Qed.

Theorem NotInRem: forall v l1 a l2, ~In v (removeL l1 (a::l2)) -> v=a \/ (v <> a /\ ~In v
      (removeL l1 l2)).
```

```
Proof.
  induction l1; intros; auto.
  elim (var_eq_dec v a); intros.
  left; auto.
  right; split; auto.
  elim (var_eq_dec v a0); intros.
  left; auto.
  right; split; auto.
  intro.
  apply H; clear H; simpl.
  case (var_eq_dec a0 a ); intros.
  rewrite e; auto.
  apply NNPP; intro.
  generalize (IHl1 a l2 H); intros.
  inversion_clear H1.
  apply b; rewrite e; auto.
  rewrite e in b.
  inversion_clear H2.
  apply H3; clear H3.
  simpl in H0.
  revert H0; case (In_dec var_eq_dec a l2); simpl; intros; auto.
  inversion_clear H0; auto.
  rewrite H2 in b; auto.
  contradiction b; auto.
  revert H0; simpl.
  case (In_dec var_eq_dec a l2 ); intros.
  apply NNPP; intro.
  elim (IHl1 a0 l2 H); intros.
  apply b; auto.
  intuition.
  simpl in H0.
  inversion_clear H0.
  rewrite H; simpl; auto.
  right.
  apply NNPP; intro.
  elim (IHl1 a0 l2 H0); intros; auto.
  intuition.
Qed.

Theorem NotInPar: forall l1 l2 v, ~In v (elimDoubL (app l1 l2)) -> ~In v l1 /\ ~In v l2.
Proof.
  intros.
  apply  NotInApp.
  apply NotInElim; auto.
Qed.

(** resultat sur Interp **)

Theorem BESVRe: forall a l B Sem BBSem,(Interp Sem BBSem (BlocESV (a::l) B)) <-> (Interp
    Sem BBSem (BlocESV (a::nil)  (BlocESV l B))).
Proof.
  intros; split; intros.
  inversion_clear H.
  eapply inter_bESVCons.
  apply  inter_bESVNil.
  apply H0.
  inversion_clear H.
  eapply inter_bESVCons with (xd:=xd).
  inversion_clear H0.
  apply H.
Qed.

Theorem BESVNilRe: forall Sem BBSem B,
Interp Sem BBSem (BlocESV nil B)<->Interp Sem BBSem B.
Proof.
  intros.
  split.
  intros.
  inversion_clear H; auto; intros.
```

```
  intros.
  apply inter_bESVNil; auto.
Qed.

Theorem BESVReG: forall xs ys B Sem BBSem,
(Interp Sem BBSem (BlocESV (app xs ys) B)) <-> (Interp Sem BBSem (BlocESV (xs)  (BlocESV
    ys B))).
Proof.

  intros; split.
  revert Sem; revert xs; revert ys.
  induction xs.
  simpl; intros.
  rewrite ->BESVNilRe; auto.
  simpl; intros.
  inversion_clear H.
  eapply inter_bESVCons with (xd:=xd).
  eapply IHxs; auto.
  revert Sem; revert xs; revert ys.
  induction xs.
  simpl; intros.
  inversion_clear H; auto.
  intros; simpl.
  inversion_clear H.
  eapply inter_bESVCons with (xd:=xd).
  eapply IHxs; auto.

Qed.

Theorem inverUpdate: forall x1 x2 Sem v1  v2, x1<>x2-> update (update Sem x2 v2) x1 v1=
    update (update Sem x1 v1) x2 v2.
Proof.

  intros.
  unfold update.
  match goal with |- ?f = ?g =>  apply (functional_extensionality (A:=var) (B:=d) f g)
      end; intro.
  case (var_eq_dec x x1); case (var_eq_dec x x2); intros; auto.
  rewrite <- e in H; rewrite <- e0 in H; intuition.

Qed.

Theorem UpdateUpdate: forall x Sem v1  v2, update (update Sem x v2) x v1=update Sem x v1.
Proof.

  unfold update; intros.
  match goal with |- ?f = ?g =>  apply (functional_extensionality (A:=var) (B:=d) f g)
      end; intro.
  case (var_eq_dec x0 x); auto.

Qed.

Theorem duplicateDiff: forall x1 x2 l, duplicate (x1::x2::l)=true-> x2 <>x1.
Proof.

  simpl; intros.
  revert H; case (var_eq_dec x2 x1); intuition.

Qed.

Theorem inver12V: forall x1 x2 l b I BI, (x1<>x2)->(Interp I BI (BlocESV (x1::x2::l) b)<-
    > Interp I BI (BlocESV (x2::x1::l) b)).
Proof.

  intros.
  split; intros.
  inversion_clear H0.
  inversion_clear H1.
  rewrite inverUpdate in H0.
```

```
  eapply inter_bESVCons.
  eapply inter_bESVCons.
  apply H0.
  auto.
  inversion_clear H0.
  inversion_clear H1.
  rewrite inverUpdate in H0.
  eapply inter_bESVCons.
  eapply inter_bESVCons.
  apply H0.
  auto.

Qed.

Theorem DisInterPar1: forall Sem BBSem l b b1, (Interp Sem BBSem (BlocESV l (Parallele b
    b1))->(Interp Sem BBSem (BlocESV l b))).
Proof.
  intros; revert H; revert Sem; induction l; intros.
  inversion_clear H.
  inversion_clear H0.
  apply inter_bESVNil; auto.
  inversion_clear H.
  apply inter_bESVCons with (xd:=xd).
  apply IHl; auto.
Qed.

Lemma DisInterPar2: forall Sem BBSem l b b1, (Interp Sem BBSem (BlocESV l (Parallele b b1
    ))->(Interp Sem BBSem (BlocESV l b1))).
Proof.

  intros; revert H; revert Sem; induction l; intros.
  inversion_clear H.
  inversion_clear H0.
  apply inter_bESVNil; auto.
  inversion_clear H.
  apply inter_bESVCons with (xd:=xd).
  apply IHl; auto.

Qed.

Lemma interBBSemNilRe: forall Sem BBSem B,
Interp Sem BBSem (BlocESV nil B)<->Interp Sem BBSem B.
Proof.

  intros.
  split.
  intros.
  inversion_clear H; auto; intros.
  intros.
  apply inter_bESVNil; auto.

Qed.

Theorem RemNil: forall l, removeL l nil = l.
Proof.

  intros; induction l; simpl; intros; auto.
  rewrite IHl; auto.

Qed.

Theorem DelFree:forall Sem BBSem v b xd, ~ In v (freeVars b) -> (Interp (update Sem v xd)
     BBSem b<->Interp Sem BBSem b).
Proof.

  intros; revert H; revert Sem; revert b; induction b; intros.
  simpl in H.
  split; intros.
  inversion_clear H0.
```

```
  apply inter_bES.
  rewrite BBSemOk  in H1; auto.
  inversion_clear H0.
  apply inter_bES.
  rewrite BBSemOk; auto.
  revert H; revert IHb; revert Sem;revert v;  induction l; intros; split; intros.
  apply inter_bESVNil.
  simpl in H.
  rewrite interBBSemNilRe in H0.
  rewrite RemNil in H.
  rewrite IHb in H0; auto.
  apply inter_bESVNil.
  inversion_clear H0.
  simpl in H; rewrite RemNil in H.
  rewrite <- IHb in H1; auto.
  inversion_clear H0.
  apply inter_bESVCons with (xd:=xd0).
  simpl in H.
  elim (NotInRem _ _ _ _ H); clear H; intros.
  rewrite H in H1.
  rewrite UpdateUpdate in H1; auto.
  inversion_clear H.
  rewrite inverUpdate in H1; auto.
  generalize (IHl  _ (update Sem a xd0) IHb H2); intro.
  intuition.
  inversion_clear H0.
  apply inter_bESVCons with (xd:=xd0).
  simpl in H.
  elim (NotInRem _ _ _ _ H); clear H; intros.
  rewrite H.
  rewrite UpdateUpdate; auto.
  inversion_clear H.
  rewrite inverUpdate; auto.
  generalize (IHl  _ (update Sem a xd0) IHb H2); intro.
  intuition.
  split; intros.
  inversion_clear H0.
  simpl in H.
  apply NotInPar in H.
  inversion_clear H.
  apply inter_par.
  rewrite IHb1 in H1; auto.
  rewrite IHb2 in H2; auto.
  simpl in H.
  apply NotInPar in H.
  inversion_clear H.
  inversion_clear H0.
  apply inter_par.
  rewrite IHb1; auto.
  rewrite IHb2; auto.

Qed.

Theorem DisVarsL: forall Sem BBSem b b1 v,~ In v (freeVars b)->
((Interp Sem BBSem (Parallele b (BlocESV (v::nil) b1)))->Interp Sem BBSem (BlocESV (v::
    nil) (Parallele b b1))).
Proof.

  intros.
  inversion_clear H0.
  inversion_clear H2.
  eapply inter_bESVCons with (xd:=xd).
  apply inter_bESVNil.
  apply inter_par.
  inversion_clear H0.
  generalize (DelFree Sem BBSem0 v _ xd H); intros.
  rewrite -> H0; auto.
  inversion_clear H0; auto.
```

```
Qed.

Theorem verifDBB:  forall n id, ((verifDoubB (BlocES n id))=true)-> ((duplicate (BBInterf
    n)) =true).
Proof.

  simpl; intros.
  revert H; case (duplicate (BBInterf n)); intuition.

Qed.

Theorem verifDBESV:forall l1 b, verifDoubB (BlocESV l1 b) = true -> ((duplicate l1)=true
    /\ (verifDoubB b)=true).
Proof.

  simpl.
  intro; intro.
  case (duplicate l1); case (verifDoubB b); intuition.

Qed.
```

B.8 Vérification de la correction des transformations

```
(** transformation correcte **)
Definition correct (f: bloc->bloc):=forall Sem b,(verifDoubB b)=true-> ((Interp Sem BBSem
    b<->Interp Sem BBSem (f b)) /\ (verifDoubB (f b))=true) .

(** la composition des transformations est correcte **)
Theorem comCorrect: forall f1 f2, correct f1-> correct f2-> correct (fun b => f1 (f2 b)).
Proof.

  intros.
  unfold correct.
  intros.
  split.
  eapply iff_trans.
  apply (proj1 (H0 Sem _ H1)).
  generalize (proj2 (H0 Sem _ H1)); intros.
  apply (proj1 (H Sem _ H2)).
  generalize (proj2 (H0 Sem _ H1)); intros.
  generalize (proj2 (H Sem _ H2)); intros.
  auto.

Qed.

(** if the else des transformations est correct **)
(* version1 *)
Theorem ifCorrect: forall br1 br2 (cond: bloc->bool), (correct br1)->(correct br2)->
    correct (fun x => if(cond x) then br1 x else br2 x).
Proof.

  intros.
  unfold correct.
  intros.
  case cond.
  split.
  apply (proj1 (H Sem _ H1)).
  generalize (proj2 (H Sem _ H1)); intros; auto.
  split.
  apply (proj1 (H0 Sem _ H1)).
  generalize (proj2 (H0 Sem _ H1)); intros; auto.

Qed.

(* version2 *)
```

```
Theorem ifSCorrect: forall T1 T2 (cond: forall (b:bloc), {T1 b}+{T2 b}) br1 br2, (correct
     br1)->(correct br2)->correct (fun x => if(cond x) then br1 x else br2 x).
Proof.

  intros.
  unfold correct.
  intros.
  case cond.
  intros.
  split.
  apply (proj1 (H Sem _ H1)).
  generalize (proj2 (H Sem _  H1)); intros; auto.
  intros.
  split.
  apply (proj1 (H0 Sem  _ H1)).
  generalize (proj2 (H0 Sem _  H1)); intros; auto.

Qed.

(** fonction donnant le meme bloc est correcte **)
Theorem correctId: correct( fun x=> x).
Proof.

  unfold correct.
  intros; split.
  reflexivity.
  auto.

Qed.

(** Transformation descendVar est correcte**)
Theorem descendVarCorrect: correct (descendVar).
Proof.

  unfold correct; intros; split; intros; revert H; case b; clear b; simpl; intros; auto.
  reflexivity.
  assert ((duplicate l= true) /\ ( verifDoubB b=true)).
  revert H; case (duplicate l); case (verifDoubB); intuition.
  clear H; inversion_clear H0.
  revert H1; revert H; case b;case l; clear b; clear l; intros; split; intros; auto.
  inversion_clear H0; auto.
  apply inter_bESVNil; auto.
  revert H0; revert H; revert Sem; revert l; induction l; intros; auto.
  apply inter_bESVNil; auto.
  generalize (duplicateCons1 _ _ _ H); intro.
  inversion_clear H0.
  inversion_clear H3.
  rewrite inverUpdate in H0.
  eapply inter_bESVCons.
  apply IHl; auto.
  eapply inter_bESVCons.
  eapply H0.
  eapply duplicateDiff with (l:=l); auto.
  revert H0; revert H; revert Sem; revert l; induction l; intros; auto.
  inversion_clear H0; auto.
  generalize (duplicateCons1 _ _ _ H); intro.
  inversion_clear H0.
  rewrite<-inver12V; try (apply duplicateDiff with (l:=l); tauto).
  eapply inter_bESVCons with (xd:=xd).
  apply IHl; auto.
  inversion_clear H0; auto.
  apply inter_bESVNil; auto.
  revert H0; revert H; revert Sem; revert l; induction l; intros; auto.
  apply inter_bESVNil; auto.
  inversion_clear H0.
  inversion_clear H2.
  eapply inter_bESVCons with (xd:= xd0); auto.
  generalize (duplicateCons1 _ _ _ H); intro.
  rewrite inverUpdate in H0.
```

```
apply inter_bESVCons in H0.
eapply IHl; auto.
eapply duplicateDiff with (l:=l); auto.
revert H0; revert H1; revert H; revert Sem;induction l; intros; auto.
inversion_clear H0.
inversion_clear H2.
eapply inter_bESVCons with (xd:= xd); auto.
rewrite<-inver12V; try (  apply duplicateDiff with (l:=l); tauto).
generalize (duplicateCons1 _ _ _ H); intro.
inversion_clear H0; auto.
eapply inter_bESVCons with (xd:=xd).
apply IHl; auto.
inversion_clear H0; auto.
apply inter_bESVNil; auto.
case (In_dec var_eq_dec v (freeVars b)); intros; auto.
revert H0; revert H1; revert H; revert Sem; induction l; intros; auto.
apply inter_bESVNil; auto.
inversion_clear H0.
inversion_clear H2.
inversion_clear H0.
apply inter_par.
rewrite <- BESVNilRe in H2.
apply inter_bESVCons in H2.
inversion_clear H2.
inversion_clear H0.
rewrite DelFree in H2; auto.
eapply inter_bESVCons with (xd:= xd); auto.
apply inter_bESVNil; auto.
inversion_clear H0.
inversion_clear H2.
rewrite inverUpdate in H0.
eapply inter_bESVCons with (xd:= xd0).
apply inter_bESVCons in H0.
generalize (duplicateCons1 _ _ _ H); intro.
apply IHl; auto.
apply duplicateDiff with (l:=l); auto.
revert H0.
case (In_dec var_eq_dec v (freeVars b)); intros; auto.
revert H0; revert H1; revert H; revert Sem; induction l; intros; auto.
inversion_clear H1.
inversion_clear H0.
inversion_clear H1.
rewrite <- BESVNilRe in H0.
inversion_clear H0.
inversion_clear H2.
inversion_clear H0.
eapply inter_bESVCons with (xd:=xd).
apply inter_bESVNil.
apply inter_par.
rewrite  DelFree.
auto.
auto.
auto.
rewrite<-inver12V; try (apply duplicateDiff with (l:=l)); auto.
inversion_clear H0.
eapply inter_bESVCons with (xd:= xd).
generalize (duplicateCons1 _ _ _ H); intro.
apply IHl; auto.
reflexivity.
assert ((duplicate l= true) /\ ( verifDoubB b=true)).
revert H; case (duplicate l); case (verifDoubB b); intuition.
inversion_clear H0.
revert H.
case l; intros; auto.
revert H; revert H1; induction b; intros; auto.
assert ((duplicate (v::l0)= true) /\ ( verifDoubB (BlocES n n0)=true)).
revert H; case (duplicate (v::l0)); case (verifDoubB (BlocES n n0)); intuition.
inversion_clear H0.
apply duplicateCons in H3.
```

```
  simpl.
  apply verifDBB in H4.
  revert H3; revert H4; case (duplicate l0); case (duplicate (BBInterf n)); intuition.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (BlocESV l1 b)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (BlocESV l1 b)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  simpl.
  apply verifDBESV in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (duplicate l1);
      case ( verifDoubB b); intuition.
  case (In_dec var_eq_dec v (freeVars b1)); intros; auto.
  simpl.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (Parallele b1 b2)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (Parallele b1 b2)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  apply verifDBpar in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (verifDoubB b1);
      case (verifDoubB b2); intuition.

Qed.

(** Transformation permuteVars est correcte **)
Theorem permuteVarsCorrect: correct permuteVars.
Proof.

  unfold correct; intros; split; intros; revert H; case b; clear b; simpl; intros; auto.
  reflexivity.
  assert ((duplicate l= true) /\ ( verifDoubB b=true)).
  revert H; case (duplicate l); case (verifDoubB); intuition.
  clear H; inversion_clear H0.
  revert H1; revert H; case l; clear l; intros; split; intros; auto.
  revert H; revert H0; revert Sem; induction l; intros; auto.
  inversion_clear H0.
  apply inter_bESVNil.
  eapply inter_bESVCons.
  eauto.
  generalize (duplicateCons1 _ _ _ H); intro.
  inversion_clear H0.
  inversion_clear H3.
  eapply inter_bESVCons with (xd:= xd0).
  rewrite inverUpdate in H0.
  apply inter_bESVCons in H0.
  apply IHl; auto.
  eapply duplicateDiff with (l:=l); auto.
  revert H; revert H0; revert Sem; induction l; intros; auto.
  inversion_clear H0.
  inversion_clear H2.
  eapply inter_bESVCons with (xd:=xd); auto.
  generalize (duplicateCons1 _ _ _ H); intro.
  inversion_clear H0.
  rewrite<-inver12V; try (apply duplicateDiff with (l:=l); tauto).
  eapply inter_bESVCons with (xd:= xd).
  apply IHl; auto.
  reflexivity.
  assert ((duplicate l= true) /\ ( verifDoubB b=true)).
  revert H; case (duplicate l); case (verifDoubB b); intuition.
  inversion_clear H0.
  revert H.
  case l; intros; auto.
  revert H; revert H1; induction b; intros; auto.
  assert ((duplicate (v::l0)= true) /\ ( verifDoubB (BlocES n n0)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (BlocES n n0)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  simpl.
```

```
  apply verifDBB in H4.
  revert H3; revert H4; case (duplicate l0); case (duplicate (BBInterf n)); intuition.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (BlocESV l1 b)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (BlocESV l1 b)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  simpl.
  apply verifDBESV in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (duplicate l1);
      case ( verifDoubB b); intuition.
  case (In_dec var_eq_dec v (freeVars b1)); intros; auto.
  simpl.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (Parallele b1 b2)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (Parallele b1 b2)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  apply verifDBpar in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (verifDoubB b1);
      case (verifDoubB b2); intuition.
  simpl.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (Parallele b1 b2)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (Parallele b1 b2)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  apply verifDBpar in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (verifDoubB b1);
      case (verifDoubB b2); intuition.

Qed.

(** Transformation descRightCorrect est correcte**)
Theorem  descRightCorrect: correct descRight.
Proof.

  unfold correct; intros; split; intros; revert H; case b; clear b; simpl; intros; auto.
  reflexivity.
  revert H.
  case l; intros; auto.
  revert H; induction b; intros; auto; reflexivity.
  case l0.
  case b.
  reflexivity.
  reflexivity.
  intros; case (In_dec var_eq_dec v (freeVars b0)); intuition.
  inversion_clear H0.
  apply inter_par.
  inversion_clear H1.
  inversion_clear H0.
  rewrite DelFree in H1; auto.
  inversion_clear H1.
  inversion_clear H0.
  eapply inter_bESVCons.
  apply inter_bESVNil; eauto.
  inversion_clear H0.
  inversion_clear H2.
  inversion_clear H0.
  apply inter_bESVCons with (xd:=xd).
  apply inter_bESVNil.
  apply inter_par; eauto.
  rewrite DelFree; intuition.
  reflexivity.
  reflexivity.
  revert H; case l; intros; auto.
  revert H; case l0; intros; auto.
  revert H; case b; intros; auto.
  assert ((duplicate (v::nil)= true) /\ ( verifDoubB (Parallele b0 b1) =true)).
```

```
  revert H; case (duplicate (v::nil)); case (verifDoubB (Parallele b0 b1)); intuition.
  inversion_clear H0.
  case (In_dec var_eq_dec v (freeVars b0)); intros; auto.

Qed.

(** Transformation split1 est correcte**)
Theorem  split1Correct:correct split1.
Proof.

   unfold correct; intros; split; intros; revert H; case b; clear b;  simpl; intros; auto
       .
   reflexivity.
   revert H.
   case l; intros; auto.
   reflexivity.
   split; intros.
   inversion_clear H0.
   eapply inter_bESVCons with (xd:=xd).
   apply inter_bESVNil; auto.
   inversion_clear H0.
   inversion_clear H1.
   apply inter_bESVCons with (xd:=xd); auto.
   reflexivity.
   simpl.
   assert ((duplicate l= true) /\ ( verifDoubB b=true)).
   revert H; case (duplicate l); case (verifDoubB b); intuition.
   inversion_clear H0.
  revert H.
  case l; intros; auto.
  revert H; revert H1; induction b; intros; auto.
  assert ((duplicate (v::l0)= true) /\ ( verifDoubB (BlocES n n0)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (BlocES n n0)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  simpl.
  apply verifDBB in H4.
  revert H3; revert H4; case (duplicate l0); case (duplicate (BBInterf n)); intuition.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (BlocESV l1 b)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (BlocESV l1 b)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  simpl.
  apply verifDBESV in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (duplicate l1);
      case ( verifDoubB b); intuition.
  case (In_dec var_eq_dec v (freeVars b1)); intros; auto.
  simpl.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (Parallele b1 b2)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (Parallele b1 b2)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  apply verifDBpar in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (verifDoubB b1);
      case (verifDoubB b2); intuition.
  simpl.
  assert ((duplicate(v::l0)= true) /\ ( verifDoubB (Parallele b1 b2)=true)).
  revert H; case (duplicate (v::l0)); case (verifDoubB (Parallele b1 b2)); intuition.
  inversion_clear H0.
  apply duplicateCons in H3.
  apply verifDBpar in H4.
  inversion_clear H4.
  revert H; revert H3; revert H0; revert H5; case (duplicate l0); case (verifDoubB b1);
      case (verifDoubB b2); intuition.

Qed.
```

```
(** Transformation descendreVars est correcte**)
Theorem descendreVarsCorrect: correct (descendreVars).
Proof.

  unfold descendreVars.
  apply ifCorrect.
  apply descRightCorrect.
  apply ifCorrect.
  apply split1Correct.
  apply ifCorrect.
  apply permuteVarsCorrect.
  apply correctId.

Qed.

(** Transformation applyInside est correcte **)
Theorem applyInsCorrect: forall (f: bloc-> bloc), correct f-> correct (applyInside f).
Proof.

  unfold correct.
  intros; split; intros.
  revert H; revert H0; case b; intros.
  simpl.
  reflexivity.
  simpl.
  apply verifDBESV in H0.
  inversion_clear H0.
  generalize (H Sem _ H2); intros.
  inversion_clear H0.
  revert H1; revert H3; revert H; revert Sem; induction l; intros; auto.
  split; intros.
  apply inter_bESVNil.
  inversion_clear H0.
  apply (proj1 H3); auto.
  apply inter_bESVNil.
  inversion_clear H0.
  apply (proj2 H3); auto.
  split; intros.
  inversion_clear H0.
  apply inter_bESVCons with (xd:=xd).
  apply duplicateCons in H1.
  assert ( verifDoubB (BlocESV l b0) = true ).
  simpl.
  revert H2; revert H1; case (duplicate l); case (verifDoubB b0); intuition.

  clear H3.
  revert H1; revert H; revert H5; revert Sem; induction l; intros; auto.
  apply inter_bESVNil.
  generalize (H (update Sem a xd) _ H2); intros.
  inversion_clear H3.
  inversion_clear H5.
  apply (proj1 H6); auto.
  generalize (H (update Sem a xd) _ H2); intros.
  inversion_clear H3.
  generalize (IHl (update Sem a xd) H H6 H1); intros.
  apply (proj1 H3); auto.
  inversion_clear H0.
  apply inter_bESVCons with (xd:=xd).
  clear H3.
  generalize (H (update Sem a xd) _ H2); intros.
  inversion_clear H0.
  apply duplicateCons in H1.
  generalize (IHl (update Sem a xd) H H3 H1); intros.
  apply (proj2 H0); auto.
  simpl.
  reflexivity.
  unfold applyInside.
  revert H0; case b; intros; auto.
  apply verifDBESV in H0.
```

```
  inversion_clear H0.
  simpl.
  generalize (H Sem b0 H2); intros.
  inversion_clear H0.
  revert H1; revert H4; case (duplicate l); case (verifDoubB (f b0)); intuition.

Qed.

(** Transformation swap est correcte**)
Theorem SwapCorrect: correct (swap).
Proof.

  unfold correct.
  intros; split; intros.
  revert H; case b; intros.
  simpl.
  reflexivity.
  simpl.
  reflexivity.
  simpl; split; intros.
  inversion_clear H0.
  apply inter_par; auto.
  inversion_clear H0.
  apply inter_par; auto.
  revert H; case b; intros.
  simpl.
  apply verifDBB in H.
  revert H; case (duplicate (BBInterf n)); intuition.
  simpl.
  simpl in H.
  revert H; case (duplicate l); case (verifDoubB b0); intuition.
  simpl.
  simpl in H.
  revert H; case (verifDoubB b0); case (verifDoubB b1); intuition.

Qed.

(** Transformation shiftRight est correcte**)
Theorem ShiftRightCorrect: correct (shiftRight).
Proof.

  unfold correct.
  intros; split; intros.
  revert H; case b; intros.
  simpl.
  reflexivity.
  simpl.
  reflexivity.
  simpl; split; intros.
  revert H; revert H0; case b0; intros; auto.
  inversion_clear H0.
  inversion_clear H1.
  apply inter_par; auto.
  apply inter_par; auto.
  revert H; revert H0; case b0; intros; auto.
  inversion_clear H0.
  inversion_clear H2.
  apply inter_par.
  apply inter_par; auto.
  auto.
  revert H; case b; intros.
  simpl.
  apply verifDBB in H.
  revert H; case (duplicate (BBInterf n)); intuition.
  simpl.
  simpl in H.
  revert H; case (duplicate l); case (verifDoubB b0); intuition.
  simpl.
  simpl in H.
```

```
  revert H; case b0; intros; auto.
  assert (((verifDoubB (Parallele b2 b3))=true) /\ ((verifDoubB b1)=true)).
  revert H; case (verifDoubB (Parallele b2 b3)); case (verifDoubB b1); intuition.
  inversion_clear H0.
  apply verifDBpar in H1.
  inversion_clear H1.
  simpl.
  revert H0; revert H2; revert H3; case (verifDoubB b1); case (verifDoubB b2); case (
      verifDoubB b3); intuition.

 Qed.

(** Transformation appl1 est correcte**)
Theorem appl1Correct:forall( f: bloc->bloc), correct f -> correct (appl1 f).
Proof.

  unfold correct.
  intros; split; intros.
  revert H; revert H0; case b; intros.
  simpl.
  reflexivity.
  simpl.
  reflexivity.
  simpl; split; intros.
  inversion_clear H1.
  apply inter_par.
  simpl in H0.
  assert (((verifDoubB b0)=true)/\((verifDoubB b1)=true)).
  revert H0; case (verifDoubB b0); case (verifDoubB b1); intuition.
  inversion_clear H1.
  generalize (H Sem b0 H4 ); intros.
  inversion_clear H1.
  rewrite -> H6 in H2; auto.
  auto.
  inversion_clear H1.
  apply inter_par.
  simpl in H0.
  assert (((verifDoubB b0)=true)/\((verifDoubB b1)=true)).
  revert H0; case (verifDoubB b0); case (verifDoubB b1); intuition.
  inversion_clear H1.
  generalize (H Sem b0 H4 ); intros.
  inversion_clear H1.
  rewrite <- H6 in H2; auto.
  auto.
  revert H; revert H0; case b; intros.
  simpl.
  apply verifDBB in H0.
  revert H0; case (duplicate (BBInterf n)); intuition.
  simpl.
  simpl in H0.
  revert H0; case (duplicate l); case (verifDoubB b0); intuition.
  simpl.
  simpl in H0.
  assert (((verifDoubB b0)=true) /\ ((verifDoubB b1)=true)).
  revert H0; case (verifDoubB b0); case (verifDoubB b1); intuition.
  inversion_clear H1.
  generalize (H Sem b0 H2); intros.
  inversion_clear H1.
  revert H5; revert H3; case (verifDoubB (f b0)); case (verifDoubB b1); intuition.

Qed.

(** Transformation appl2 est correcte **)
Theorem appl2Correct:forall( f: bloc->bloc), correct f ->correct(appl2 f).
Proof.

  unfold correct.
  intros; split; intros.
  revert H; revert H0; case b; intros.
```

```
  simpl.
  reflexivity.
  simpl.
  reflexivity.
  simpl; split; intros.
  inversion_clear H1.
  apply inter_par; auto.
  simpl in H0.
  assert (((verifDoubB b0)=true)/\((verifDoubB b1)=true)).
  revert H0; case (verifDoubB b0); case (verifDoubB b1); intuition.
  inversion_clear H1.
  generalize (H Sem b1 H5); intros.
  inversion_clear H1.
  rewrite -> H6 in H3; auto.
  inversion_clear H1.
  apply inter_par; auto.
  simpl in H0.
  assert (((verifDoubB b0)=true)/\((verifDoubB b1)=true)).
  revert H0; case (verifDoubB b0); case (verifDoubB b1); intuition.
  inversion_clear H1.
  generalize (H Sem b1 H5); intros.
  inversion_clear H1.
  rewrite <- H6 in H3; auto.

  revert H; revert H0; case b; intros.
  simpl.
  apply verifDBB in H0.
  revert H0; case (duplicate (BBInterf n)); intuition.
  simpl.
  simpl in H0.
  revert H0; case (duplicate l); case (verifDoubB b0); intuition.
  simpl.
  simpl in H0.
  assert (((verifDoubB b0)=true) /\ ((verifDoubB b1)=true)).
  revert H0; case (verifDoubB b0); case (verifDoubB b1); intuition.
  inversion_clear H1.
  generalize (H Sem b1 H3); intros.
  inversion_clear H1.
  revert H5; revert H2; case (verifDoubB (f b1)); case (verifDoubB b0); intuition.

Qed.

(** Transformation permute1 est correcte**)
Theorem permute1Correct: forall id, correct (permute1 id).
Proof.

  intros.
  unfold permute1.
  apply ifCorrect.
  apply ifCorrect.
  apply ifCorrect.
  apply ShiftRightCorrect.
  apply ifCorrect.
  apply comCorrect.
  apply ShiftRightCorrect.
  apply appl1Correct.
  apply SwapCorrect.
  apply correctId.
  apply correctId.
  apply correctId.

Qed.

(** Transformation permute2 est correcte**)
Theorem permute2Correct: forall id, correct (permute2 id).
Proof.

  intros.
  unfold permute2.
```

```
  apply ifCorrect.
  apply ifCorrect.
  apply ifCorrect.
  apply comCorrect.
  apply appl2Correct.
  apply SwapCorrect.
  apply comCorrect.
  apply ShiftRightCorrect.
  apply SwapCorrect.
  apply ifCorrect.
  apply comCorrect.
  apply appl2Correct.
  apply SwapCorrect.
  apply comCorrect.
  apply ShiftRightCorrect.
  apply comCorrect.
  apply SwapCorrect.
  apply appl2Correct.
  apply SwapCorrect.
  apply correctId.
  apply ifCorrect.
  apply SwapCorrect.
  apply correctId.
  apply correctId.

Qed.

(** Transformation moveOut est correcte **)
Theorem moveOutCorrect: forall id, correct (moveOut id).
Proof.

  intro.
  unfold correct; intros.
  revert H; revert b; induction b; intros; auto.
  split; intros.
  simpl.
  reflexivity.
  simpl.
  simpl in H.
  revert H; case (duplicate (BBInterf n)); intuition.
  split; intros.
  simpl.
  reflexivity.
  simpl.
  simpl in H.
  revert H; case (duplicate l); case (verifDoubB b); intuition.
  split; intros.
  simpl.
  split; intros.
  inversion_clear H0.
  apply verifDBpar in H.
  inversion_clear H.
  generalize (IHb1 H0); intros.
  inversion_clear H.
  generalize (IHb2 H3); intros.
  inversion_clear H.
  rewrite -> H4 in H1.
  rewrite -> H6 in H2.
  generalize (inter_par Sem _ _ BBSem H1 H2); intros.
  assert (verifDoubB (Parallele (moveOut id b1) (moveOut id b2)) = true).
  simpl.
  revert H5; revert H7; case (verifDoubB (moveOut id b1)); case (verifDoubB (moveOut id
      b2)); intuition.
  generalize (permute1Correct); intros.
  unfold correct in H9.
  assert(Interp Sem BBSem (Parallele (moveOut id b1) (moveOut id b2))<->Interp Sem BBSem
      (permute1 id (Parallele (moveOut id b1) (moveOut id b2)))).
  generalize (H9 id Sem _ H8);intros.
  inversion_clear H10.
```

```
  exact H11.
  generalize (permute2Correct); intros.
  unfold correct in H11.
  assert(Interp Sem BBSem (permute1 id (Parallele (moveOut id b1) (moveOut id b2)))<->
      Interp Sem BBSem (permute2 id (permute1 id (Parallele (moveOut id b1) (moveOut id
      b2))))); intros.
  assert (verifDoubB (permute1 id (Parallele (moveOut id b1) (moveOut id b2))) = true).
  generalize (H9 id Sem _ H8);intros.
  inversion_clear H12.
  exact H14.
  generalize (H11 id Sem _ H12);intros.
  inversion_clear H13; auto.
  rewrite -> H10 in H.
  rewrite -> H12 in H; auto.
  apply verifDBpar in H.
  inversion_clear H.
  generalize (IHb1 H1); intros.
  inversion_clear H.
  generalize (IHb2 H2); intros.
  inversion_clear H.
  assert (verifDoubB (Parallele (moveOut id b1) (moveOut id b2)) = true).
  simpl.
  revert H4; revert H6; case (verifDoubB (moveOut id b1)); case (verifDoubB (moveOut id
      b2)); intuition.
  generalize (permute1Correct); intros.
  unfold correct in H7.
  generalize (H7 id Sem _ H);intros.
  inversion_clear H8.
  generalize (permute2Correct); intros.
  unfold correct in H8.
  generalize (H8 id Sem _ H10);intros.
  inversion_clear H11.
  rewrite <-H12 in H0.
  rewrite <-H9 in H0.
  inversion_clear H0.
  rewrite <-H3 in H11.
  rewrite <-H5 in H14.
  apply inter_par; auto.
  simpl.
  apply verifDBpar in H.
  inversion_clear H.
  generalize (IHb1 H0); intros.
  inversion_clear H.
  generalize (IHb2 H1); intros.
  inversion_clear H.
  assert (verifDoubB (Parallele (moveOut id b1) (moveOut id b2)) = true).
  simpl.
  revert H3; revert H5; case (verifDoubB (moveOut id b1)); case (verifDoubB (moveOut id
      b2)); intuition.
  generalize (permute1Correct); intros.
  unfold correct in H6.
  generalize (H6 id Sem _ H);intros.
  inversion_clear H7.
  generalize (permute2Correct); intros.
  unfold correct in H7.
  generalize (H7 id Sem _ H9);intros.
  inversion_clear H10; auto.

Qed.

(** repeat d'une fonction est correcte**)
Theorem repeatCorrect:forall( f: bloc->bloc)(m: bloc->nat)(c: bloc->bool)(h: forall x, if
     c x then m (f x)<m x else True), correct f -> correct (repeat _ f m c h).
Proof.

  intros.
  unfold correct.
  intros.
  revert H;revert H0.
```

```
  pattern b.
  apply (Fix_measure _ m); intros.
  rewrite repeat_eq.
  case_eq (c x); intros.
  generalize (h0 x).
  rewrite H2; simpl; intros.
  generalize (H (exist (fun y => m y< m x) (f x) H3)); simpl; intros.
  generalize (H1  Sem x H0); intros.
  rewrite (proj1 H5).
  generalize (H4 (proj2 H5) H1); intros.
  inversion_clear H5; auto.
  split; intros; auto.
  reflexivity.

Qed.

(** La fonction Move d'un bloc est correcte **)
Theorem MoveOutFv1Correct: forall id, correct (moveOutFv1 id).

Proof.
  intro.
  unfold moveOutFv1.
  apply comCorrect.
  unfold moveOutF1.
  apply (repeatCorrect).
  apply descendreVarsCorrect.
  unfold moveOutI.
  apply comCorrect.
  apply applyInsCorrect.
  apply moveOutCorrect.
  apply correctId.

Qed.

End Semantique.
End Bloc.
```

Bibliographie

[1] C. André. Comparaison des styles de programmation de langages synchrones. *Projet AOSTE-Juin*, 2005. 5

[2] G. Berry, P. Couronné, G. Gonthier, and J. Banatre. Programmation synchrone des systèmes réactifs : le langage Esterel. *TSI. Technique et science informatiques*, 6(4) :305–316, 1987. 4

[3] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9) :1293–1304, Sep 1991. 8

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow language LUSTRE. *Proceedings of the IEEE*, 79(9) :1304–1320, 1991. 4, 8

[5] D. Harel et al. Statecharts : A visual formalism for complex systems. *Science of computer programming*, 8(3) :231–274, 1987. 4

[6] B. Houssais. Cours de programmation en langage synchrone signal. Technical report, IRISA, Equipe ESPRESSO, 2004. 4, 8

[7] X. Leroy, D. Doligez, J. Garrigue, D. Remy, and J. Vouillon. The objective caml system release 3.11. *INRIA, may*, 2008. 4

[8] P. Narbel. *Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml*. Vuibert, 2005. 14

[9] C. Paulin-Mohring, B. Werner, B. Barras, and H. Herbelin. Calcul des constructions inductives. *Course, January*, 1999. 26

[10] M. Sozeau. Subset coercions in Coq. In T. Altenkirch and C. McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006. 27

[11] T. C. D. Team. The Coq proof assistant reference manual. *INRIA. Version 8.2*, 6(11), 2009. 4, 25

[12] L. Zaffalon. *Programmation synchrone de systèmes réactifs avec Esterel et les SyncCharts*. Presses polytechniques et universitaires romandes, 2005. 4, 9, 10