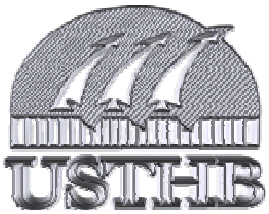
N° d'ordre : 06 / 2007- M / IN

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE



Faculté d'Electronique et Informatique

MEMOIRE
Présenté pour l'obtention du Diplôme de Magister
En : Informatique

Spécialité : Programmation et Systèmes

Par : Abbas Messaoud

Thème

Transcription des spécifications UML vers le système FoCAL

Soutenu le 04/07/2007, devant le Jury composé de :

Mr -	**Mezghiche Mohamed**	**Professeur**	**U.M.B. Boumerdes**	**Président**
Mr -	**Ben-Yelles Choukri-Bey**	**Professeur**	**U.P.M.F (Grenoble 2)**	**Directeur de Thèse**
Mr -	**Belkhir Abdelkader**	**Maître de conférence**	**U.S.T.H.B**	**Examinateur**
Mme -	**Alimazighi Zaia**	**Maître de conférence**	**U.S.T.H.B**	**Examinatrice**

Remerciements

Je tiens à remercier mon directeur de thèse le Prof. Choukri-Bey BEN-YELLES, pour m'avoir fait confiance en me proposant ce sujet de thèse.

J'exprime aussi ma profonde reconnaissance et mes vifs remerciements aux membres de l'équipe FoCAL de l'USTHB, Choukri-Bey BEN-YELLES, Lies KADDOURI et Karim BENABADJI, pour leur encadrement, leurs critiques et suggestions et pour les lectures attentives de mes rapports qui ont été d'un grand apport pour la finalité de ce travail.

Je remercie vivement M. Mohamed MEZGHICHE d'avoir accepté de présider ce jury de Magister.

Je remercie Mme Zaia ALI-MAZIGHI et M. Abdelkader BELKHIR d'avoir accepté d'examiner ce travail.

J'adresse également mes sincères remerciements, aux membres de l'équipe FoCAL du LIP6 pour leurs conseils et remarques, et plus particulièrement à M. Mathieu Jaume et M. Stéphane Fechter.

Un grand MERCI à tous les enseignants du département d'informatique de l'USTHB.

Résumé

Le système FoCAL est un atelier de développement orienté objet et de programmation certifiée. La méthode graphique UML permet de représenter des systèmes de manière synthétique et intuitive. Cependant, elle n'est pas dotée d'outils permettant de faire de la vérification. Dans cette thèse, nous abordons une approche de transcription d'un modèle UML exprimé par des diagrammes de classes, des diagrammes d'état-transition et de contraintes OCL dans le système FoCAL. La transcription proposée, permet d'obtenir une spécification FoCAL, exprimée par une hiérarchie d'espèces, à partir du modèle UML puis d'utiliser Zenon (l'outil de preuve automatique de FoCAL) pour vérifier certaines propriétés telles que la cohérence entre les diagrammes et les contraintes OCL.

Mots clés : FoCAL, UML, OCL, Formel, preuve, Zenon.

Sommaire

Introduction

La maîtrise du logiciel passe par une spécification rigoureuse dès les premières étapes du cycle de vie. Malgré les progrès enregistrés dans ce domaine, nous continuons à souffrir du manque d'un cadre adéquat de spécification [CAR00]. D'un côté nous avons des langages à base de notations graphiques, tels qu'UML [BOO03], [GUN01] qui permettent de représenter des systèmes de manière synthétique et intuitive, mais auxquels il manque les bases formelles nécessaires pour faire de la vérification [LED03], [MEY01], [PAL99]. De l'autre côté, nous avons des langages formels, tels que FoCAL [PRE03], [JAU06], [FOC06] qui fournissent des notations mathématiques permettant de spécifier très précisément les propriétés du système à construire. Le modèle est alors moins intuitif, mais peut être validé grâce à des techniques de preuve formelle [JAU05], [ETI05].

Le système formel FoCAL, acronyme de "Formal Ocaml Coq Algebraic Library", est un atelier intégré de construction modulaire de logiciels certifiés, développé au sein du LIP6, de l'INRIA et du CNAM.

Notre objectif est d'utiliser le système FoCAL pour vérifier des propriétés et appliquer des preuves formelles au sein d'un modèle UML muni de contraintes OCL. Le processus de vérification passe d'abord par la traduction du modèle UML dans le système FoCAL, puis par l'utilisation des techniques de vérification et de preuve formelle offertes par ce système.

Ainsi, nous commençons par la dérivation des diagrammes de classes en une hiérarchie d'espèces, où toute classe sera transcrite en une espèce. Les relations entre classes (héritage, association, …) seront maintenues entre les espèces. Concernant l'aspect comportemental, à savoir la communication entre les objets au sein des diagrammes d'état-transition (State-Charts), des ensembles énumérés ainsi que des méthodes seront introduits dans les espèces dérivées. Les contraintes OCL seront traduites vers des propriétés pour les utiliser lors de la preuve comme des axiomes.

L'aspect preuve sera axé principalement sur la recherche d'éventuelles contradictions entre les diagrammes d'état-transition et les contraintes OCL. Pour prouver des théorèmes, le système FoCAL utilise COQ [COQ97], un outil d'aide à la preuve. Dans sa dernière version,

FoCAL a été renforcé par un autre outil de preuve appelé Zenon [DOL04] qui permet d'automatiser le processus de preuve.

Plusieurs travaux se sont intéressés à l'application des techniques de preuves formelles sur un modèle UML. Parmi ces travaux, nous pouvons citer essentiellement ceux qui se sont intéressés à la traduction d'un modèle UML vers un langage formel tels que Promela [PAL99], SMV [KWO00], LOTOS [CAR00] ou B [LED02]. La traduction depuis UML vers ces langages impose généralement des restrictions fortes sur les constructions UML qu'il est possible d'utiliser. En effet, la plupart de ces travaux sont attachés à des State-charts en isolation (modèles statiques) sans tenir compte de la communication entre eux.

Cette thèse est composée de cinq chapitres :
Le premier chapitre présente le système FoCAL à travers un exemple permettant de découvrir étape par étape, les différents concepts sous-jacents du langage FoCAL, sa modalité de développement et les outils de preuve associés au système FoCAL, à savoir Coq et Zenon.
A la fin du chapitre, une comparaison entre les concepts apportés par le langage FoCAL et les concepts orientés objets est proposée.
Le second chapitre donne une description succincte du langage de modélisation graphique UML ainsi que du langage de contraintes OCL associé. Pour plus de clarté, nous avons présenté les différents types de diagrammes (statique et dynamique) et les contraintes OCL à travers un exemple simple et classique, il s'agit du "distributeur de boissons ".
Dans le troisième chapitre, un état de l'art sur les méthodes et techniques formelles appliquées autour d'un modèle UML est présenté.
Le quatrième chapitre montre la démarche de transcription des spécifications UML/OCL vers le système FoCAL.
Et enfin, dans le cinquième chapitre nous traiterons d'une application pratique : la transcription d'un modèle UML/OCL décrivant le système de contrôle de trains vers le système FoCAL. Des tests de preuves ont été pratiqués sur la spécification FoCAL obtenue pour valider la consistance du modèle.

Chapitre I

Le Système FoCAL

Les méthodes formelles visent à prouver mathématiquement la consistance d'un système après en avoir donné une description mathématique. L'objectif de la programmation certifiée est d'intégrer les preuves dans la programmation même des logiciels. Pour cela il faut donc disposer d'outils permettant d'intégrer au même niveau des spécifications, des programmes, et des preuves. Le but de cette approche est de guider l'utilisateur pour l'amener à donner les propriétés nécessaires à la cohérence de son modèle.

Le système FoCAL est un atelier de développement orienté objet et de programmation certifiée. FoCAL fournit deux constructions de base : les espèces et les collections. Une espèce sert à spécifier et implanter des méthodes permettant la manipulation des éléments d'un type support. Une espèce est construite en utilisant les traits objets tels que : raffinement, héritage, redéfinition, liaison retardée, paramétrage. Pour utiliser une espèce, nous devons définir une collection par l'abstraction du type support [MOR06].

Le système FoCAL, dont l'objectif est de prouver les propriétés et les théorèmes introduits par le développeur, utilise le système d'aide à la preuve COQ [COQ97] (un environnement pour développer des preuves formelles). Pour cela, FoCAL produit du code COQ. Dans sa dernière version, FoCAL utilise Zenon [DOL04], un outil de preuve de théorèmes automatique. En cas de succès des phases de vérification, le système FoCAL produit une source Ocaml. Dans le présent chapitre, nous présenterons les principaux constructeurs de FoCAL à travers un exemple complet de développement. (Une présentation plus détaillée de la syntaxe de FoCAL se trouve dans [FOC06], [BOU00] et [BOU01].)

1.1 Présentation des Concepts de base du langage FoCAL

Dans cette section, les différents concepts FoCAL (espèce, collection, méthodes définies et déclarées, héritage, espèce paramétrée, etc.) seront introduits à travers le développement du produit $Z \times Z$, inspiré de la présentation faite dans [FOC03], en suivant les étapes suivantes :

1. définition d'un setoïde : ensemble muni d'une relation d'équivalence (**egal**) ;
2. définition d'un monoïde : setoïde muni d'une opération binaire (**multiplie**) et d'un élément neutre **(un)** ;
3. définition du produit cartésien de deux setoïdes (qui est aussi un setoïde) ;
4. définition du produit cartésien de deux monoïdes (qui est aussi un monoïde) ;
5. définition de l'ensemble Z comme un monoïde sur le type **int** ;
6. définition de $Z \times Z$.

La bibliothèque **basics.foc** est constituée d'une hiérarchie d'espèces prédéfinies dont la racine est l'espèce **basic_object**. Cette dernière définit les méthodes **print** et **parse** et déclare **rep**, appelé type-support, le type des éléments de l'espèce.
Généralement un développement FoCAL commence par l'utilisation des directives :

```
uses basics;;
open basics;;
```

La première directive, **uses basics,** permet l'utilisation des définitions contenues dans le fichier **basics.foc.** La deuxième directive, **open basics,** permet d'éviter de préfixer les noms des méthodes par le nom de la bibliothèque : on écrira **f** au lieu de **basics#f**.

1.1.1 Espèce

Une espèce définit un ensemble de valeurs (les entités) ainsi qu'un ensemble de méthodes (opérations, type-support, énoncés, preuve) pour manipuler ces entités. Il est à noter que chaque espèce doit avoir un unique type-support.

1.1.2 Héritage

Un setoïde est un ensemble non vide doté d'une relation d'équivalence.
L'espèce **setoïde** définie ci-dessous hérite de la hiérarchie **basic_object**.

```
species setoïde inherits basic_object =
  sig egal in  self->self->bool;
  sig element in self;
```

```
  let different (x,y) = basics#not_b(!egal(x,y));
  property refl: all x in self, !egal(x,x);
  property symm: all x y in self, !egal(x,y) -> !egal(y,x);
  property trans: all x y z in self,
      !egal(x,y) -> !egal(y,z) -> !egal(x,z);
end
```

- **rep (le type support) :** le type de représentation est uniquement déclaré dans l'espèce basic_object. Dans l'espèce setoïde, ce type a un statut abstrait (non défini), destiné à être raffiné ultérieurement.
- **self :** désigne le type-support. Par contre l'expression **self!m** (ou simplement **!m**) désigne la méthode **m** de la collection qui implémente l'espèce (*"self reference"*). Ainsi, **!egal** est un raccourci de **self!egal** qui dénote la méthode **egal** de toute collection correspond à une implémentation de l'espèce setoïde.
- **all :** dénote le quantificateur universel .
- **sig (méthode déclarée "calculatoire") :** dans l'espèce **setoïde**, les méthodes **egal** et **element** sont déclarées, mais non définies. Les définitions explicites de ces méthodes seront données au fur et à mesure du raffinement. Le type de la méthode **egal** (**self->self->bool**) spécifie clairement qu'elle est une opération binaire entre les éléments de **setoide**. La méthode **element** (de type **self**) est déclarée pour garantir qu'un setoïde est un ensemble non vide.
- **let (méthode définie "calculatoire")** : le mot clé **let** est utilisé pour définir le corps d'une méthode. La méthode **different** est une méthode définie. Le corps de cette méthode utilise la fonction de négation entre booléens **not_b** de **basics.foc**.
- **property :** permet d'exprimer des propriétés qui doivent être satisfaites par les éléments du setoïde. Par conséquent, ces propriétés doivent être prouvées avant toute implémentation de l'espèce. Les trois méthodes : **refl**, **symm** et **trans** représentent les propriétés de réflexivité, symétrie et transitivité qui garantissent que la relation **egal** est une relation d'équivalence. Ce sont des méthodes déclarées ("non calculatoires") : lors de l'implémentation de l'espèce setoïde, ces trois propriétés devront être prouvées.
- **espèce terminale/complète :** espèce où toutes les méthodes sont définies
- **Interface d'une espèce :** liste de toutes les méthodes considérées comme déclarées.

Nous pouvons maintenant définir l'espèce monoïde par héritage de l'espèce setoïde. Notre monoïde est un setoïde doté d'une opération binaire (**multiplie**) et un élément particulier

(**un**). L'élément **un** permet de définir la méthode déclarée **element** de l'espèce **setoïde** par l'expression !multiplie(!un,!un);

```
species monoïde  inherits setoïde =
  sig multiplie in self -> self -> self;
  sig un in self;
  let element = !multiplie(!un,!un);
  property assoc: all x y z in self,
       !egal(!multiplie(x,!multiplie(y,z)),!multiplie(!multiplie(x,y),z));
  property neutre: all x in self,
    !egal(!multiplie(x,!un),x) and !egal(!multiplie(!un,x),x);
  theorem un_unique: all o in self,
    (all x in self,(!egal(x,(!multiplie(x,o))))) -> !egal(!un,o)
  proof:
    decl: neutre trans;
    assumed;
end
```

Le type de la méthode définie **element** de l'espèce monoïde doit être le même que le type de **element** de l'espèce setoïde : le type d'une méthode redéfinie doit respecter le type de la méthode d'origine.

- **theorem :** la méthode **un_unique** est une propriété introduite par le mot clé **theorem**. La définition de cette méthode est une preuve qui est donnée juste après le mot clé **proof :** c'est une méthode "non calculatoire" définie.
- **decl / def :** avant de prouver un théorème nous devons exprimer des dépendances entre méthodes. Il y a deux types de dépendance :
  - dépendance introduite par le mot clé **decl**, qui exprime une dépendance aux méthodes vues comme déclarées. Dans ce cas, la preuve dépendra seulement des types de ces méthodes.
  - dépendance introduite par le mot clé **def**, qui exprime une dépendance aux méthodes vues comme définies. Dans ce cas, la preuve dépendra non seulement des types, mais aussi des définitions de ces méthodes.

  Dans notre exemple, la preuve dépend seulement des types de **neutre** et **trans**.
- **assumed :** le mot clé **assumed** permet d'éviter d'avoir à produire la preuve de la propriété, auquel cas la propriété sera considérée comme un axiome.

1.1.3 Espèce paramétrée

Le système FoCAL permet de définir une espèce paramétrée par des collections. Le produit cartésien de deux setoïdes *a* et *b* est un setoïde défini à travers l'espèce suivante :

```
species setoïde_produit(a is setoïde, b is setoïde)
  inherits setoïde =

  rep = a * b;

  let egal(x,y)=
     #and_b(a!egal(#first(x),#first(y)),b!egal(#scnd(x),#scnd(y)));
  let element = self!creer(a!element,b!element);

  let creer(x,y) in self = #crp(x,y);

  let print = fun x ->
    #sc("(",
        #sc(a!print(#first(x)),
            #sc(",",
                #sc(b!print(#scnd(x)),
                    ")"))));

  proof of refl=
    def: egal;
    assumed;

  proof of symm =
    def: egal;
    assumed;
  proof of trans =
    def: egal;
    assumed;
end
```

- Le type support de **setoïde_produit** est le produit des types supports de *a* et de *b :* **rep = a * b;** (représentant **a!rep * b!rep**).
- Les fonctions : **#crp**, **#first** et **#scnd** sont des primitives définies dans la bibliothèque FoCAL permettant respectivement de composer une paire, de récupérer le premier composant, ou le deuxième composant d'une paire.
-

1.1.4 Héritage multiple

FoCAL supporte l'héritage multiple entre les espèces. Le produit cartésien de deux monoïdes peut être réalisé par la définition d'une espèce qui hérite à la fois de l'espèce **monoïde** et de l'espèce **setoïde_produit(a,b)**. La définition de l'espèce **monoïde_produit** est donnée comme suit :

```
species monoïde _produit(a is monoïde , b is monoïde )
  inherits monoïde , setoïde_produit(a,b) =
```

```
  let un = !creer(a!un,b!un);
  let multiplie(x,y)=

#crp(a!multiplie(#first(x),#first(y)),b!multiplie(#scnd(x),#scnd(y)));
  proof of assoc = assumed;
  proof of neutre = assumed;
end
```

- **type support unique :** chaque espèce dans une liste d'héritage doit définir le même type support.
- **définitions multiples pour une méthode :** l'ordre de l'héritage est important. En cas de définitions multiples pour une méthode, la stratégie adoptée par FoCAL consiste à garder la définition la plus à droite dans la liste des espèces héritées. De plus les types de ces méthodes doivent être unifiables. Dans notre exemple, la méthode **element** est définie dans les deux espèces héritées, la définition considérée est celle de l'espèce **setoïde_produit** (la dernière espèce la contenant).
- **refaire les preuves** : si une méthode est redéfinie, alors toutes les preuves liées à l'ancienne définition de la méthode peuvent devenir fausse. Ainsi le développeur doit refaire à nouveau ces preuves.

1.1.5 Collection

La collection est une instance d'une espèce complète (terminale). Cette instance permet d'utiliser les méthodes de l'espèce. Aucune modification ne peut être apportée sur une collection.

Nous pouvons définir maintenant la collection **entiers** qui implémente l'ensemble Z. La méthode **rep** de l'espèce **monoïde** est définie par la valeur **int** et les méthodes comme des opérations sur **int**.

```
collection entiers implements monoïde  =
  rep = int;
  let un = 1;
  let multiplie = basics#int_mult;
  let egal = basics#base_eq;
  let print = basics#string_of_int;
  let parse = basics#int_of_string;

 proof of refl = assumed;

  proof of symm = assumed;
```

```
  proof of trans = assumed;

  proof of assoc=assumed;
  proof of neutre = assumed;
end
```

Pour obtenir Z x Z, il suffit maintenant d'implémenter l'espèce **monoïde_produit** en donnant aux paramètres la valeur **entiers** (la collection définie en haut) comme suit :

```
collection entiers_2 implements monoïde_produit(entiers,entiers)= end
```

- **exemples d'utilisations :**

```
let cinq = entiers!parse("5");;

let cinq_carre = entiers!multiplie(#cinq,#cinq);;

let foo = entiers_2!creer(#cinq,#cinq_carre);;

let foo2 = entiers_2!multiplie(#foo,#foo);;

#print_string(entiers_2!print(#foo2));;
```

La figure1-1 montre la hiérarchie des espèces et des collections ainsi que les relations d'héritage, paramétrage et implémentation inter espèces et collections.

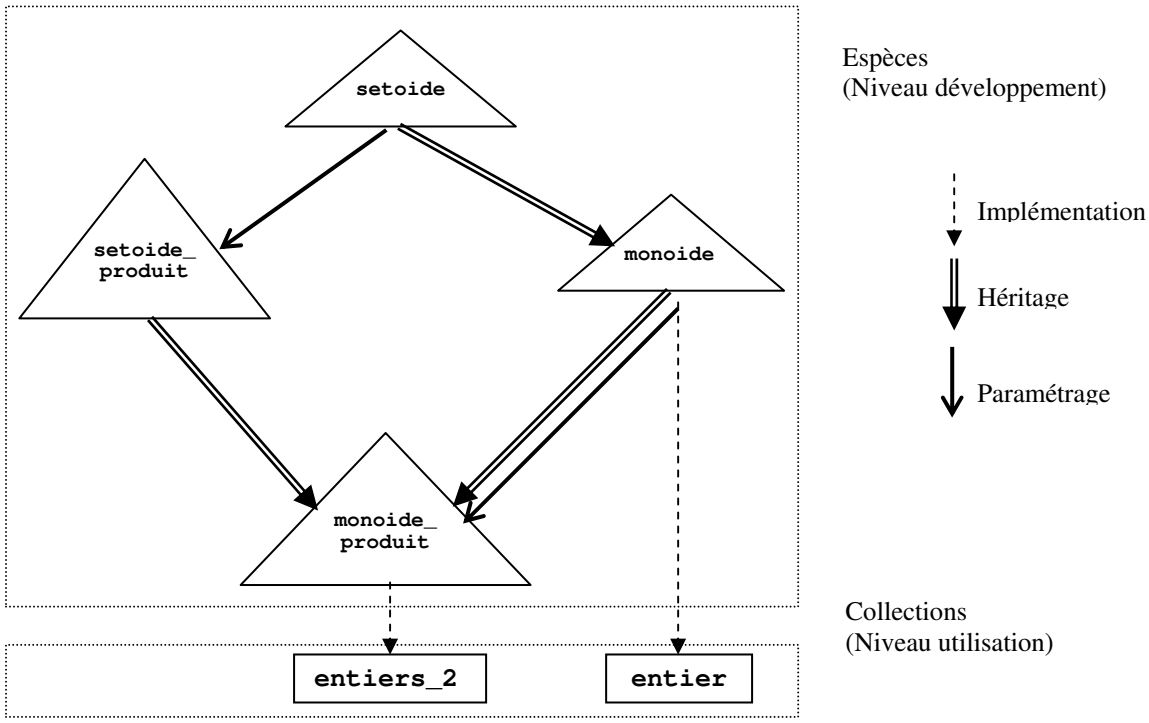


Fig 1- 1 Hiérarchie des espèces et des collections et relations entre elles.

Toutes les portions de code FoCAL présentées ci-dessus sont regroupées dans un même fichier nommé **exemple.foc** qui sera utilisé lors de la compilation.

1.2 Le compilateur FoCAL

Le modèle de développement proposé par FoCAL (la notion d'espèce et de collection) a été proposé et formalisé par Sylvain Boulmé dans [BOU00].

La charte décrite dans [BOU00] consiste à utiliser le système d'aide à la preuve Coq, puis un modèle catégorique. Sylvain Boulmé a montré qu'avec le langage OCaml, les espèces peuvent être codées par des classes paramétrées avec des variables de types. Ces variables modélisent le type support.

Le compilateur FoCAL permet d'automatiser un certain nombre de tâches et de vérifications, de la charte. Ce compilateur a été réalisé et est maintenu par V. Prevosto [PRE03].

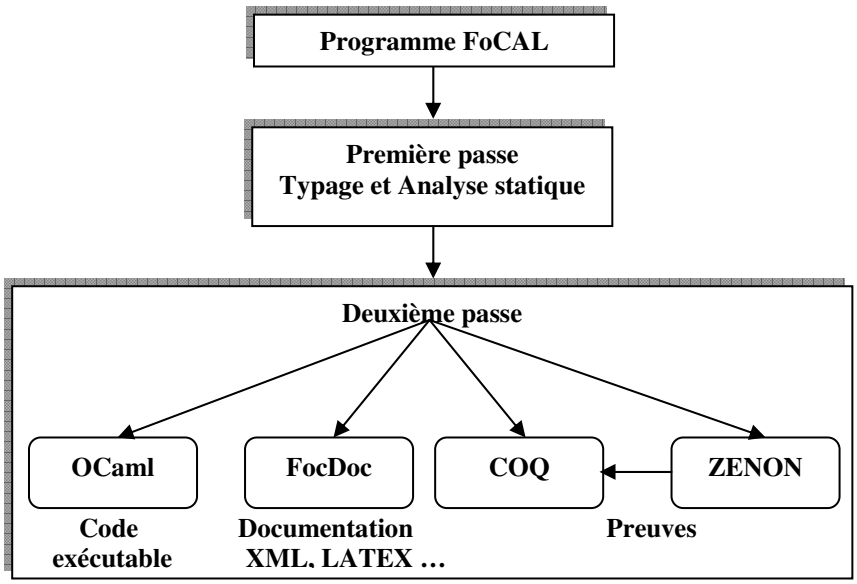


Fig 1- 2 compilation de FoCAL.

La compilation d'un programme FoCAL se déroule en deux passes :

- La première passe prend en charge la vérification des types et l'analyse statique.
- La deuxième passe fournit quatre sorties :

- Source Ocaml (exécution) ;
- Source pour un langage intermédiaire (FocDoc) permettant de produire des documentations automatiques sous format HTML, XML et LATEX ;
- Code Coq permettant de réaliser une preuve en Coq ;
- Source pour Zenon ;

Ainsi, la compilation de notre exemple **exemple.foc** est obtenue en suivant les étapes suivantes :

1. Production du source Ocaml **exemple.ml**, l'interface **exemple.mli** et du source Zenon **exemple.zv** :

   ```
   [root@localhost foc]# focc exemple.foc
   ```

2. Faire la preuve par Zenon (la commande zvtov) et produire un fichier source de preuve Coq (**exemple.v**) :

   ```
   [root@localhost foc]# zvtov exemple.zv
   ```

3. Faire la preuve par Coq :

   ```
   [root@localhost foc]# coqc exemple.v
   ```

4. Compilation de la source Ocaml et production de l'exécutable (nommé par défaut, **a.out**) :

   - `[root@localhost foc]# ocamlc exemple.mli`
   - `[root@localhost foc]# ocamlc -w mv caml_basics.cmo openmath.cmo basics0.cmo exemple.ml`

5. Exécution du programme :

   ```
   [root@localhost foc]# ocamlrun a.out
   (25,625) (* résultat de l'exécution *)
   ```

6. Génération des interfaces des espèces **exemple.fmi** :

   ```
   [root@localhost foc]# focc -i  exemple.foc
   ```

7. Génération de graphe d'héritage qui permet de voir clairement la relation d'héritage entre toutes les espèces et collections se trouvant dans **exemple.foc**

   ```
   [root@localhost foc]# inhgraph exemple.foc
   ```

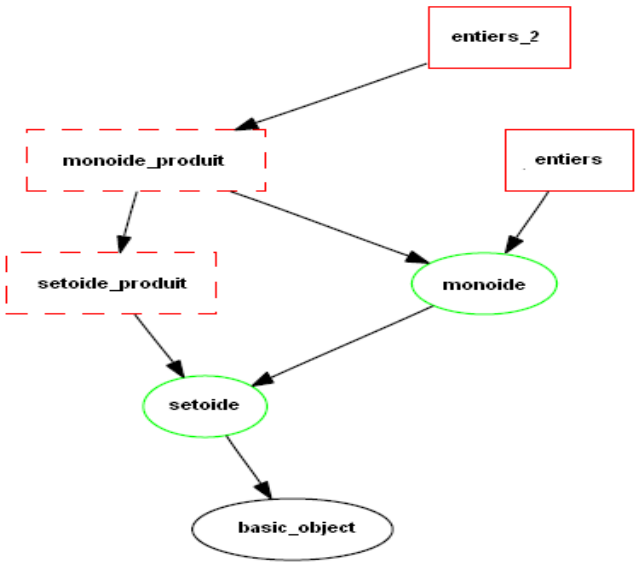


Fig 1- 3 Graphe d'héritage entre les espèces.

La figure 1.3 s'interprète comme suit :

- une collection est représentée par un rectangle rouge.
- une espèce complète est représentée par un rectangle rouge pointillé.
- une espèce définie dans le fichier actuel (exemple.foc) est représentée par une ellipse verte.
- une espèce définie en dehors du fichier, et de laquelle d'autres espèces héritent, est représentée par une ellipse noire.

8. Génération de graphe de dépendances entre les méthodes de chaque espèce dans **exemple.foc**

```
[root@localhost foc]# depgraph exemple.foc
```

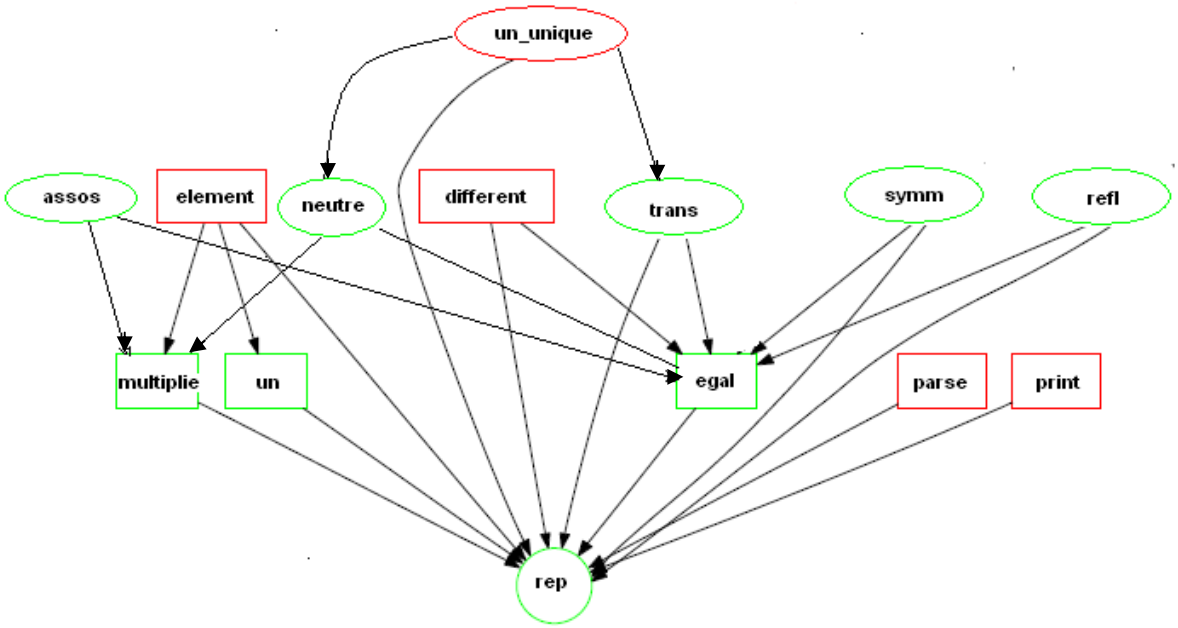


Fig 1- 4 Graphe de dépendances de l'exemple.

La figure 1.4 s'interprète comme suit :

– une decl-dépendance est représentée par une flèche noire.

– une méthode déclarée est entourée d'une forme verte.

– une méthode définie est entourée d'une forme rouge.

– une fonction est entourée d'un rectangle.

– une propriété est entourée par une ellipse.

– le type de représentation (support) est entouré d'un cercle

1.3 Les outils d'aide à la preuve associés à FoCAL

Les outils d'aide à la preuve permettent au développeur de construire la preuve (d'une manière plus ou moins automatique) d'une propriété (ou d'un théorème) en fonction d'indications fournissant le raisonnement nécessaire pour faire la preuve.

Dans ses dernières versions, FoCAL dispose d'un prouveur automatique de théorèmes basé sur la méthode des tableaux [DOL04]. Le processus de preuve en Zenon utilise une spécification du premier ordre en entrée, et retourne une preuve formelle complète en sortie directement vérifiable en Coq.

1.3.1 COQ

- Coq est un Logiciel développé à l'INRIA et à l'université Paris Sud[1] basé sur un langage issu de la théorie des types.
- Prouver une propriété dans Coq consiste à trouver le λ-terme[2] correspondant à la propriété.
- L'environnement Coq est destiné au développement et à la certification de logiciels sans erreurs dans des domaines sensibles (logiciels embarqués, protocoles de communication, commerce électronique, etc.)
- Le système Coq n'est pas orienté objet, par conséquent lors de passage de FoCAL à Coq, les espèces doivent subir un dépliage de l'héritage.

Reprenons notre exemple de l'espèce **monoïde** où nous remplaçons le mot clé **assumed** par un script de preuve Coq pour le théorème **un_unique** :

```
species monoïde  inherits setoïde =
  sig multiplie in self -> self -> self;
  sig un in self;
  let element = !multiplie(!un,!un);
  property assoc: all x y z in self,
       !egal(!multiplie(x,!multiplie(y,z)),!multiplie(!multiplie(x,y),z));
  property neutre: all x in self,
    !egal(!multiplie(x,!un),x) and !egal(!multiplie(!un,x),x);
  theorem un_unique: all o in self,
    (all x in self,(!egal(x,(!multiplie(x,o))))) -> !egal(!un,o)
  proof:
    decl: neutre trans;
          (* script de preuve Coq *)
        {* EAuto.
```

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

[2] Dans le système Coq, prouver une propriété consiste à trouver un λ-terme typé dont le type est égale exactement à la propriété à prouver ex: la preuve de $(A \to B \to C) \to (A \to B) \to A \to C = \lambda f_1^{A\to B\to C}.\lambda f_2^{A\to B}.\lambda a^{A}.(f_1\ a)(f_2\ a)$.

```
            Elim (abst_neutre o); Trivial.
            Apply (abst_trans abst_un (abst_multiplie abst_un o)).
            Exact __lemma_1_1.
            Exact __lemma_1_2.
          *};
end
```

Bien sûr, le script Coq est obtenu après avoir fait la preuve pas à pas avec COQ en utilisant la source COQ produit par le compilateur **focc** (une section COQ est générée pour chaque théorème), puis nous insérons ce script à sa place dans l'espèce **monoïde**.

lemma_1_1 et **lemma_1_2** sont de lemmes intermédiaires introduits par COQ dans la section correspond au théorème **un_unique**

1.3.2 ZENON

- Zenon est un outil de preuve **automatique** de théorèmes basé sur la méthode des tableaux [DOL04].
- A partir d'une spécification du premier ordre en entrée, Zenon retourne une preuve formelle au format Coq.

Reprenons notre exemple de l'espèce **monoïde**, pour avoir une idée sur la façon dont Zenon traite les preuves et comment l'utilisateur doit fournir des indications de preuves à Zenon. La preuve du théorème **un_unique** est la suivante :

```
species monoïde  inherits setoïde =
  sig multiplie in self -> self -> self;
  sig un in self;
  let element = !multiplie(!un,!un);
  property assoc: all x y z in self,
       !egal(!multiplie(x,!multiplie(y,z)),!multiplie(!multiplie(x,y),z));

  property neutre: all x in self,
    !egal(!multiplie(x,!un),x) and !egal(!multiplie(!un,x),x);

  theorem un_unique: all o in self,
    (all x in self,(!egal(x,(!multiplie(x,o))))) -> !egal(!un,o)
  proof:
    decl: neutre trans;
                          (* debut preuve Zenon *)
    <1>1
      assume o in self
             Hyp:all x in self,!egal(x,(!multiplie(x,o)))
      prove !egal(!un,o)
      <2>1 prove !egal(!un,!multiplie(!un,o))
         by <1>:Hyp

      <2>2 prove !egal(!multiplie(!un,o),o)
         by !neutre

      <2>3 prove !egal(!un,o)
```

```
        by <2>1 , <2>2 , !trans
      <2>4 qed

    <1>2 qed
  ;
                          (* Fin preuve Zenon *)

end
```

- En utilisant Zenon, le développeur n'a besoin de donner que des indications de preuve de ce type, c'est exactement comme si nous faisions la preuve sur papier, le reste est à la charge de FoCAL et Zenon.
- Il est à noter que dans le cas où Zenon échoue, le développeur doit interagir pour guider Zenon pour mener la preuve à son terme.

1.4 FoCAL et Ocaml

- Caml est un langage de programmation générale [REM02]. De paradigme fonctionnel très expressif et facile à utiliser, il est développé et distribué par l'INRIA depuis 1985.
- Objective Caml (Ocaml) étend le noyau du langage Caml avec une couche orientée objet et un système de modules puissant.
- L'extension objet d'Objective Caml s'intègre dans le système de types du langage. Une classe définit un type du nom de la classe. L'instance d'une classe définit une valeur du type de la classe.

1.4.1 Utilisation de code Ocaml dans Focal

La possibilité d'utiliser du code Ocaml dans FoCAL donne une puissance importante de programmation à FoCAL. Pour illustrer ce point, nous traitons l'exemple suivant :
Dans Ocaml il y a une bibliothèque qui comprend la plupart des fonctions pour manipuler des listes (**long, Hd, tl, nth, rev, append ...etc**.). L'utilisation de ces fonctions dans FoCAL est possible par importation de code Ocaml à travers un fichier intermédiaire (d'extension **.fml**).
Commençons par l'écriture du code Ocaml qui sera enregistré dans un fichier **exemple_caml.ml**. Il est à noter que le nom de fichier Ocaml doit être différent du fichier FoCAL qui va importer le code.
Le code Ocaml est le suivant :

```
let long(x) = List.length(x);;
```

```
let hd(x) = List.hd(x);;
let tl(x) = List.tl(x);;
let nth x y = List.nth x y;;
let rev x  = List.rev x ;;
let append x y = List.append x y ;;
let rev_append x y = List.rev_append x y ;;
```

Maintenant, écrivons le contenu du fichier **exemple_focal.fml** qui définit la correspondance entre le code FoCAL et le code Ocaml. Ce fichier doit être de même nom que le fichier FoCAL qui va importer le code Ocaml (**exemple_focal.foc**).
La clause **import** dans le fichier intermédiaire sert à déterminer l'endroit où le code importé sera exactement utilisé :

- soit dans des définitions globales (toplevel),
- soit à l'intérieur d'une espèce particulière.

```
ocaml header {* open Exemple_caml *}
import for toplevel
long x = {* long x *};
hd x = {* hd x *};
tl x = {* tl x *};
nth x y = {* nth x y *};
rev x  = {* rev x *};
append x y = {* append x y *};
rev_append x y = {* rev_append x y *};
```

Il reste à donner le fichier FoCAL **exemple_focal.**foc qui importe et utilise le code de **exemple_caml.ml** à travers **exemple_focal.fml** :

```
uses basics;;
open basics;;
              (* définition et initialisation de deux listes *)
let ya in (list(int)) = #Cons(3,#Cons(2,#Cons(1,#Nil))) ;;
let yb in (list(string)) = #Cons("3",#Cons("2",#Cons("1",#Nil))) ;;
                       (* importation de code Ocaml *)
let long (x in (list('a)) ) in int = caml long ;;
let hd (x in (list('a)) ) in ('a) = caml hd ;;
let tl (x in (list('a)) ) in (list('a)) = caml tl ;;
let nth (x in (list('a)), y in (int) ) in ('a) = caml nth ;;
let rev (x in (list('a)) ) in (list('a)) = caml rev ;;
let append (x in (list('a)), y in (list('a)) ) in (list('a))=caml append;;
```

```
let rev_append (x in (list('a)), y in (list('a)) ) in (list('a)) = caml
rev_append ;;
                    (* Utilisation de code importé *)
#print_int (#long (#ya))
#print_int (#long (#yb));;
#print_int (#hd (#ya));;
#print_int (#nth (#ya , 1));;
let list = (#rev_append(#ya,#yb)) ;;
```

Lors de l'importation nous devons donner explicitement les types de toute définition importée.

1.5 L'approche FoCAL et l'approche orienté objet

Dans le langage FoCAL nous retrouvons des traits objets (classe abstraite, héritage, liaison retardée, etc.) néanmoins il ne peut pas être qualifié de langage orienté objet. En effet, l'existence de nouveaux constructeurs, imposés par l'aspect preuve, le démarque plus ou moins de la famille de ces langages. Les travaux réalisés dans [FEC05] renforcent ces propos. En effet, dans ces travaux, une sémantique orientée objet n'a pu être donnée qu'à un sous-système de FoCAL, appelé Mini-Foc.

Les tableaux qui suivent présentent une comparaison entre les deux approches [FEC05], qui sera prise en compte lors de la transcription de UML vers FoCAL.

Convergences :

L'approche FoCAL	L'approche Orienté Objet
Espèce	Classe / classe abstraite /méta classe
Méthodes et déclarations	Méthodes et méthodes virtuelles
Liaison retardée	Liaison retardée
Espèce paramétrée	Classe paramétrée
Héritage et héritage multiple	Héritage et héritage multiple
Les collections sont les instances d'espèces	Les objets sont les instances de classes
Encapsulation de données.	Encapsulation de données.

Divergences :

L'approche FoCAL	L'approche Orienté Objet
La notion d'état mémoire est remplacée par la notion de type support.	L'objet est représenté par son état mémoire (attributs)
Méthodes pour implanter des algorithmes à fin de manipuler les éléments de type support.	Méthodes pour manipuler l'état d'un objet
Abstraction de l'ensemble des éléments du type support	abstraction de données
Méthodes ne sont jamais paramétrées par des collections.	Méthodes paramétrées par des objets
la collection est complètement figée	L'objet n'est pas figé, peut évoluer

Chapitre II

Présentation d'UML

La modélisation orientée objet consiste à créer une présentation informatique du monde réel sans se préoccuper de l'implémentation, ce qui signifie indépendamment d'un langage de programmation. Il s'agit donc d'identifier les objets présents et d'isoler leurs données et les fonctions qui les utilisent.

Les langages orientés objets constituent chacun une manière spécifique d'implémenter le paradigme objet. Ainsi, une méthode objet permet de définir le problème à un haut niveau sans rentrer dans les spécifications d'un langage [BOO03].

Le développement orienté objet implique dans un premier temps une conception abstraite d'un modèle objet (c'est le rôle de la méthode objet) et dans un second temps son implémentation à l'aide d'un langage orienté objet (tel que C++ / Java/…).

De nombreuses méthodes orientées objets ont été mises au point ces dernières années. UML (Unified Modeling language) [BOO03] est le résultat de la fusion des trois méthodes suivantes :

- La méthode OMT de Rumbaugh [RUM91]
- La méthode BOOCH'93 de Booch [BOO91]
- La méthode OOSE de Jacobson [JAC92]

C'est un langage de modélisation à base de notations graphiques permettant de modéliser un système réel. Le modèle est décrit à travers différentes vues, chacune d'elle est représentée par un ou plusieurs diagrammes. UML définit des diagrammes structurels et comportementaux pour représenter respectivement des vues statiques et dynamiques d'un système. La version actuelle d'UML (UML 2.0) propose 13 types de diagrammes.

Il est à signaler qu'UML n'est pas une méthode dans la mesure où elle ne présente aucune démarche, mais plutôt est considéré comme un formalisme de modélisations orienté objet [BOO03].

Pour décrire des contraintes additionnelles, UML utilise le langage OCL (Object Constraint Language) [SCH02] [WAR03]. Ce langage permet d'exprimer les contraintes du système à modéliser, impossible à décrire qu'avec les notations graphiques d'UML.

2.1 Les diagrammes d'UML

Un diagramme donne à l'utilisateur un moyen de visualiser et de manipuler des éléments de sa modélisation. Généralement, neufs types de diagrammes sont suffisants pour modéliser la majorité des systèmes [MUL00]. Ces diagrammes sont répartis en deux groupes :

Cinq diagrammes structurels pour représenter des vues statiques :

- Diagramme de cas d'utilisation
- Diagramme de classe
- Diagramme à objet
- Diagramme de composants
- Diagramme de déploiement

Quatre diagrammes comportementaux pour représenter des vues dynamiques :

- Diagramme de collaboration
- Diagramme de séquence
- Diagramme état-transition
- Diagramme d'activités

Dans ce qui suit, nous allons donner une description détaillée des diagrammes UML à travers un exemple classique, il s'agit du ***distributeur de boissons***.

2.1.1 Les diagrammes de cas d'utilisation

Ils présentent sous forme d'un ensemble d'actions et d'acteurs le comportement d'un système en face d'un utilisateur. Ce diagramme consiste à répondre à la question par qui et dans quel cas ce système est utilisé ?
Une vue générale de l'utilisation d'un système de distributeur de boissons est donnée dans la figure 2-1a.

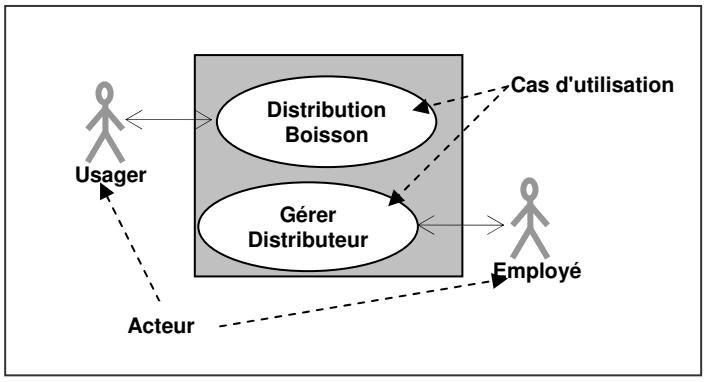


Fig 2-1a Diagramme de cas d'utilisation du système de distribution de boisson.

Une vue plus détaillée du cas d'utilisation "distribution boisson" représente l'interaction entre un usager et le sous système de distribution de boissons. Dans ce cas, l'utilisateur demande un type particulier de boisson. Si la boisson est disponible, le système lui demande d'insérer la monnaie. Après introduction de la monnaie, le système le sert.

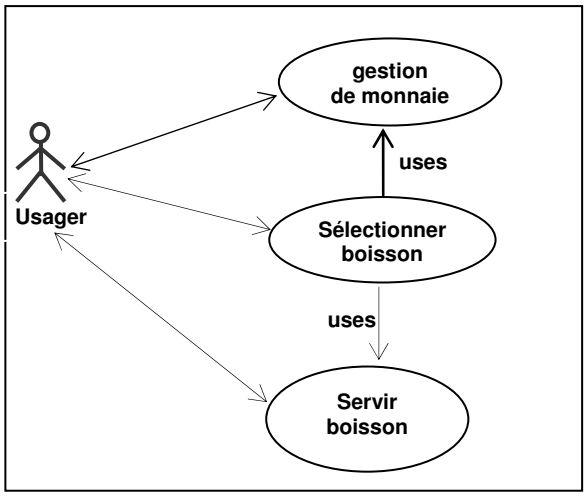


Fig 2- 1b Diagramme de cas d'utilisation du sous système "distribution boisson"

2.1.2 Les diagrammes de collaboration

Ils présentent les interactions entre les différents objets qui constituent le système et les messages qu'ils interchangent entre eux.

Dans notre exemple la collaboration se fait comme suit :

Un objet utilisateur choisit une boisson par envoi du message1 (*choisir_boisson()*) à l'objet distributeur. Ce message est décomposé en trois sous messages : 1.1(*teste_disponibilite()*) envoyé à l'objet boisson, 1.2 (*recupere_prix()*) envoyé à l'objet boisson et

1.3(*affiche_monnaie(prix)*) envoyé à l'objet écran. L'événement d'insertion de monnaie de l'objet utilisateur déclenche l'envoi du message2 (*monnaie_inseree (prix)*) vers l'objet gestionnaire_monnaie. Le massage 2 à son tour, invoque l'envoi du message2.1 (*autorise()*) vers l'objet distributeur pour exécuter l'action donner boisson 2.2 (*donner_boisson*()).

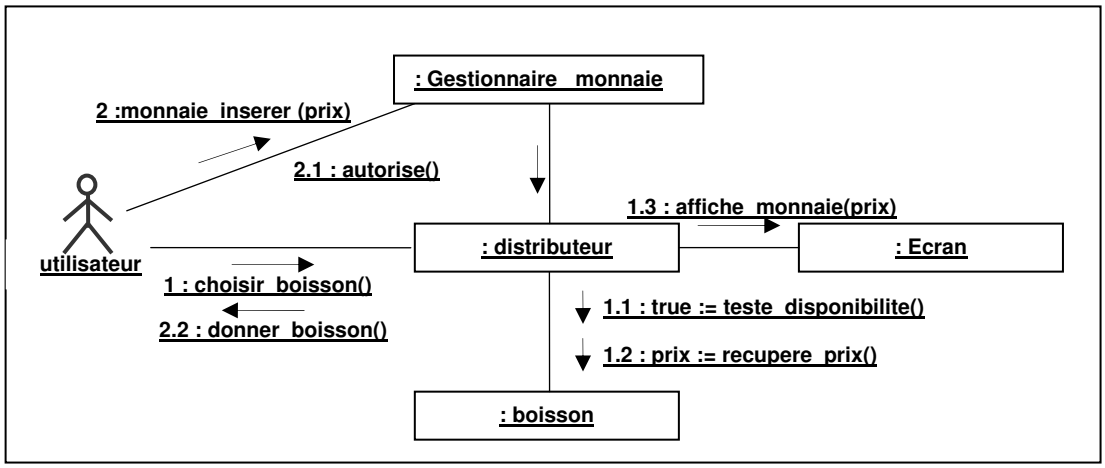


Fig 2- 2 Diagramme de collaboration du système de distribution de boisson.

2.1.3 Les diagrammes de séquences

Ils décrivent l'échange de messages entre objets intervenants en tenant compte du séquencement chronologique des messages.

Ainsi, le scénario relatif au cas boisson disponible entre l'usager et les objets du système peut être schématisé par le diagramme de séquences de la figure 2.3

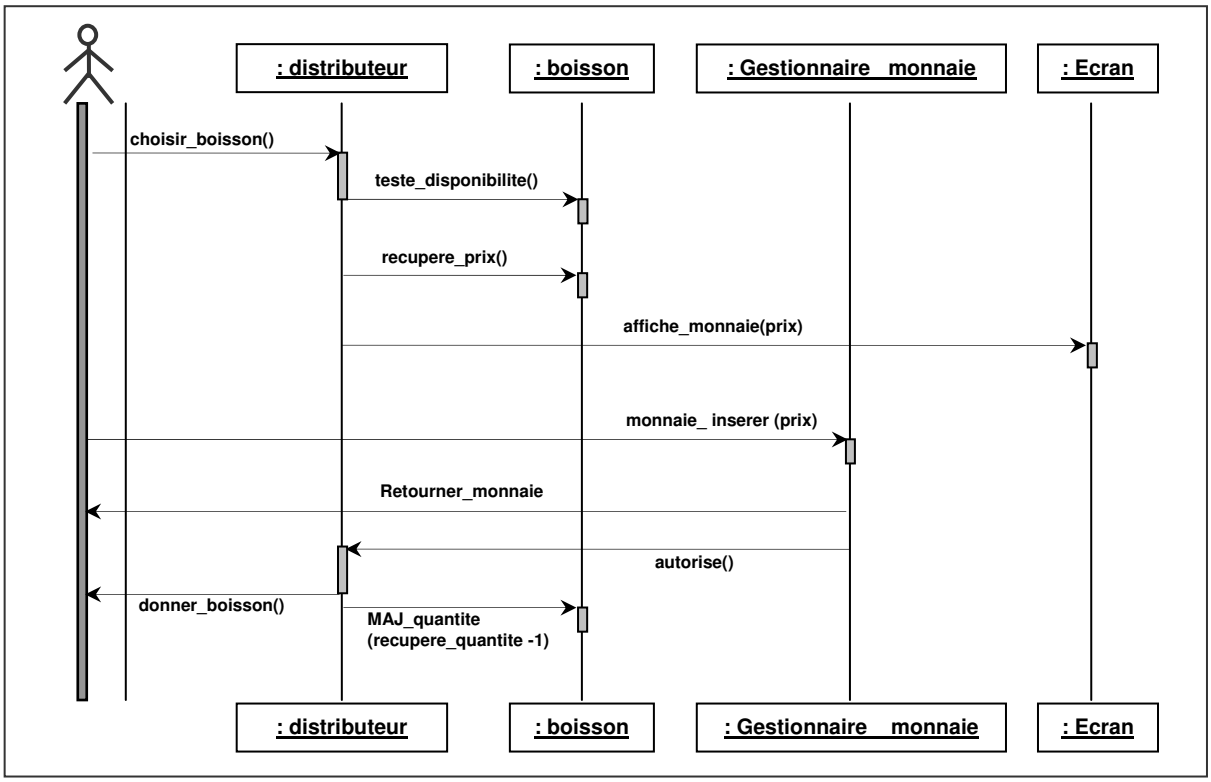


Fig 2- 3 Diagramme de séquences du scénario distribution boisson disponible.

2.1.4 Les diagrammes de classes

Ils expriment la structure d'un système par un ensemble de classes et de relations (héritage, association, dépendance, ...) entre ces classes. Une classe regroupe un ensemble d'objets partageant les mêmes propriétés (attributs, méthodes et relations).

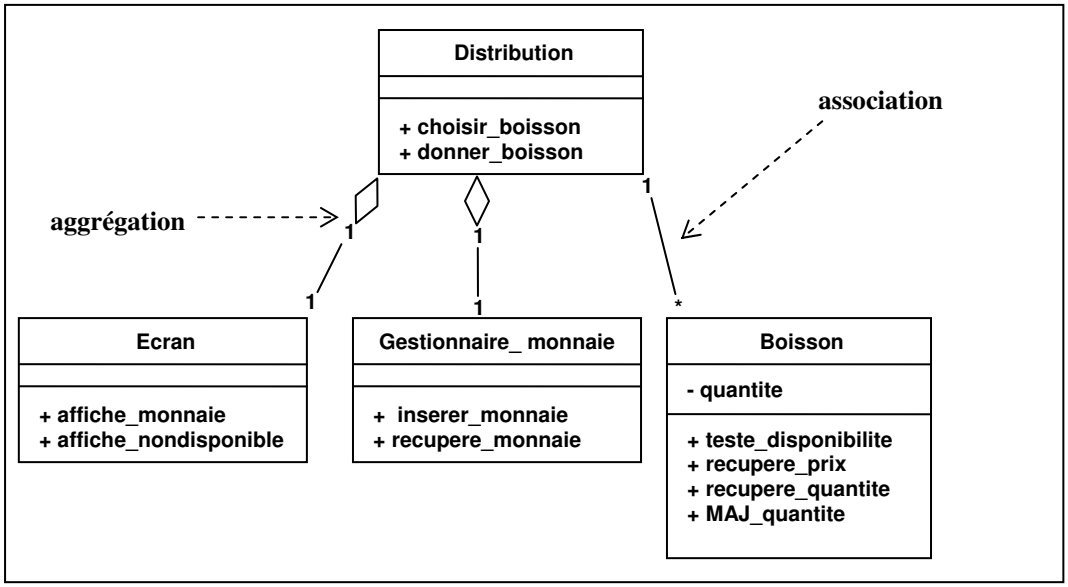


Fig 2- 4 Diagramme de classes du système de distribution de boisson.

Attribut : est une propriété nommée partagée par les objets d'une classe (ex : l'attribut **quantite** de la classe **boisson**).

Méthode : est un service que l'on peut demander à un objet pour réaliser un comportement (ex : la méthode **choisir_boisson** de la classe **Distribution**).

Association : est une connexion sémantique exprimée entre deux ou plusieurs classes (ex : l'association entre la classe **Distribution** et la classe **Boisson** qui met en association la distribution avec plusieurs boissons).

Héritage : est un mécanisme par lequel des éléments plus spécifiques incorporent la structure et le comportement d'éléments plus généraux (ex : classe **utilisateur** hérite de la classe **personne**)

2.1.5 Les diagrammes d'objets

Ils représentent une instanciation du diagramme de classe. C'est-à-dire les relations entre des objets (de classe) à un instant donné.

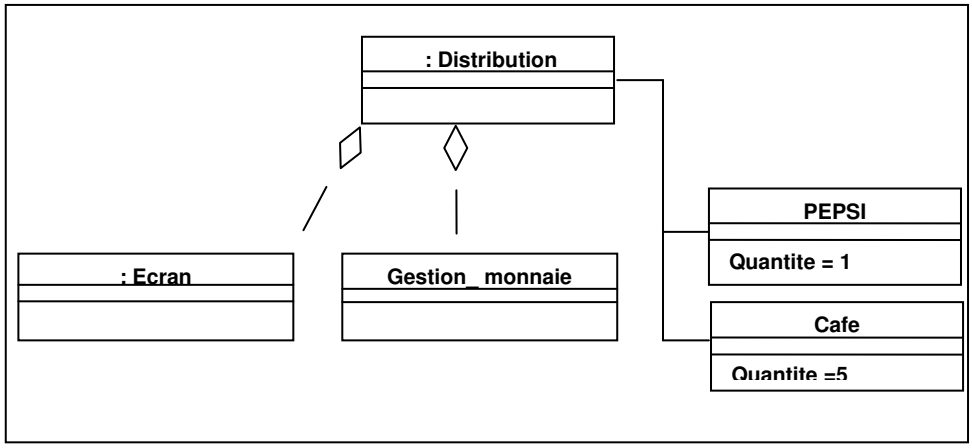


Fig 2- 5 Diagramme d'objets du système de distribution de boisson.

2.1.6 Les diagrammes d'état-transition

Se sont des automates d'états finis, composés d'états, de transitions et d'événements d'activités. Ces diagrammes décrivent les états et le comportement des instances d'une classe en réponse à des événements extérieurs. Dans ce type de diagrammes, nous nous intéressons principalement au séquencement d'opérations.

Le diagramme suivant décrit l'état-transition de la classe distributeur de boisson :

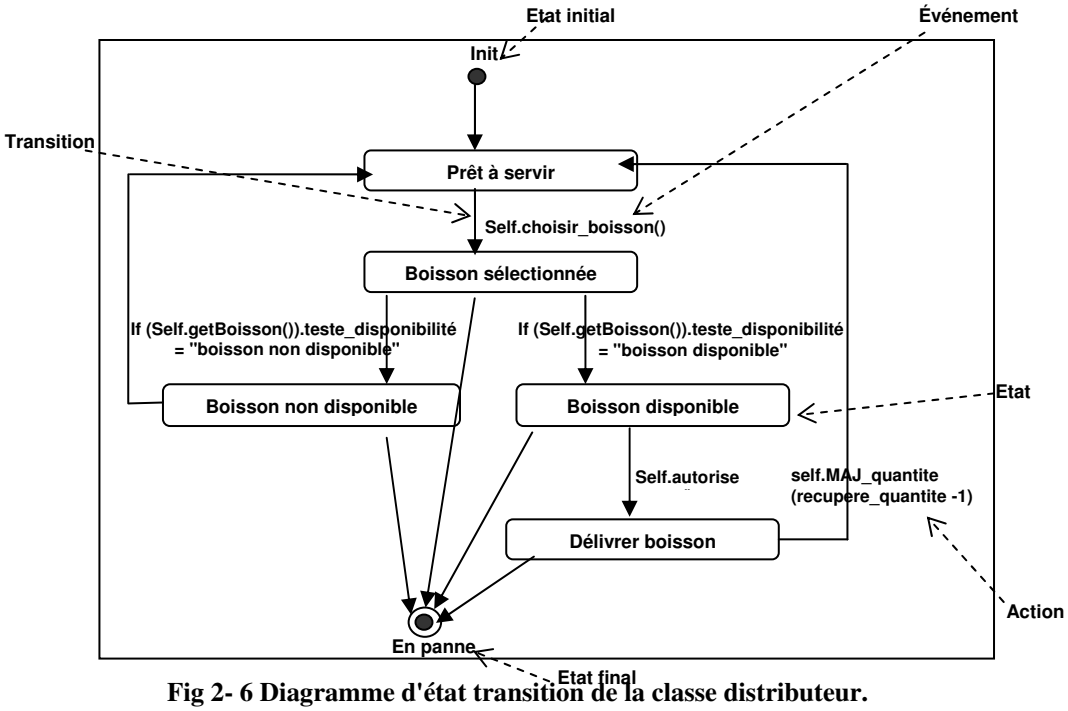


Fig 2- 6 Diagramme d'état transition de la classe distributeur.

Etat : représente l'état d'un objet à un moment donné.

Etat initial : l'état d'un objet juste après sa création.

Etat final : l'état d'un objet juste avant sa destruction.

Transition : représente le passage d'un objet d'un état à un autre, déclenché par un événement.

Action : opération exécutée en cours d'une transition.

2.1.7 Les diagrammes des composants

Un composant correspond généralement à une ou plusieurs classes ou interfaces. Les diagrammes de composants consistent à regrouper sous forme de paquetages les différents composants d'un système. Ils s'intéressent généralement à la vue d'implémentation statique et logique du système.

Dans notre exemple nous distinguons le paquetage Ecrans et distributeurs.

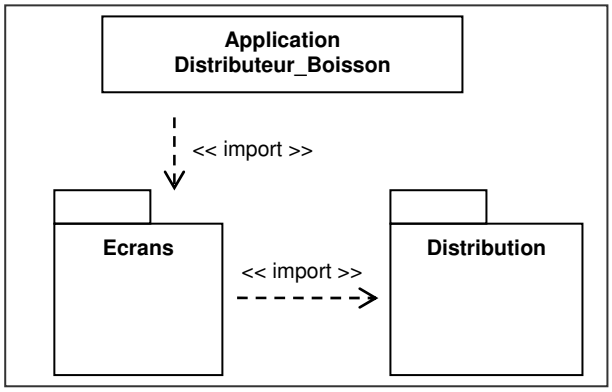


Fig 2- 7 Diagramme de composants du système de distribution de boisson.

Les diagrammes de déploiement ou les diagrammes d'activités permettent de modéliser d'autres aspects du système.

Les diagrammes de déploiement décrivent les ressources de calcul, leurs configurations et le lien entre les exécutables et les ressources de calcul.

Les diagrammes d'activités décrivent l'enchainement des activités, c'est à dire le comportement d'une classe en réponse à des calculs internes. Ils sont similaires aux diagrammes d'état-transitions, mais pour les événements internes.

2.2 Le langage OCL

Une contrainte OCL est toujours définie dans un contexte qui représente une instance d'une classe. OCL permet principalement d'exprimer deux types de contraintes sur l'état d'un objet ou d'un ensemble d'objets :

- Les invariants qui doivent être vérifiés en permanence.
- Les pré-conditions et post-conditions vérifiées respectivement avant et après l'exécution d'une opération.

Nous tenons à signaler q'une expression OCL ne fait que décrire une contrainte et que la vérification nécessite un outil de preuve.

2.2.1 L'invariant

L'invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence.

Dans l'expression OCL suivante :

```
context Boisson
inv: quantite > 0
```

La contrainte impose pour toutes les instances de la classe Boisson que l'attribut **quantite** doit toujours être positif.

2.2.2 Les pré-condition et post-condition

Les pré-condition et post-condition sont des contraintes utilisées pour spécifier une opération :

- **Pré-condition** : état qui doit être respecté avant l'appel d'une opération.
- **Post-condition** : état qui doit être respecté après l'appel d'une opération.

L'expression OCL suivante :

```
context Distributeur::donner_boisson()
pre:(((Boisson.quantite>0)and(gestionnaire_monnaie.argent_insere))
post: Boisson.quantite = Boisson.quantite@pre – 1
```

Cette contrainte exprime le fait que la quantité d'une boisson doit être positive et que la monnaie doit être insérée pour que l'appel de l'opération (**donner_boisson**) soit valide. Après l'exécution de l'opération, l'attribut **quantite** doit avoir pour valeur la différence (**Boisson.quantite@pre)-1** où **Boisson.quantite@pre** désigne la quantité avant l'appel de l'opération.

Dans ce cadre, nous nous limitons à ces deux contraintes OCL, qui seront utilisées dans notre étude.

Chapitre III

UML et techniques formelles : état de l'art

Il est évident que UML devient le langage de modélisation le plus répandu dans la conception et le développement de logiciels industriels. Mais, il demeure néanmoins une méthode semi formelle, le principal reproche réside dans l'absence de base formelle permettant l'application des techniques de vérifications et de preuves formelles.

Dans ce contexte, plusieurs travaux ont été réalisés dont l'objectif est de doter les notations graphiques d'UML de bases formelles et d'outils de vérifications, afin de produire de logiciels cohérents.

La génération de tests [FLE99] [FRÖ00][OFF99], était la technique la plus intuitive pour vérifier des propriétés au sein d'un modèle UML. Une deuxième technique consiste à traduire le modèle UML (ou une partie du modèle) en un langage formel [LED02] [MEY01] [KWO00] [CAR00]. Les réseaux de Pétri sont aussi utilisés pour le même objectif tel que dans [PAL99]. Une autre approche est l'utilisation d'UML lui-même, telle que faite par le groupe précis UML (pUML) dans [CLA01].

Dans ce qui suit, nous allons voir un état de l'art des différentes techniques citées ci-dessus.

3.1 Vérification par génération de tests

Plusieurs travaux se sont intéressés à la vérification par génération de tests, nous pouvons citer :

- **Tests à partir des diagrammes [FLE99] :** Un modèle est représenté par un ensemble de diagrammes en utilisant la notation ROOM et par un ensemble de cas d'utilisation sous formes de diagrammes de séquences, puis les diagrammes décrivant le modèle sont compilés vers un modèle de simulation. Pour un scénario donné, le modèle est stimulé avec les événements d'entrée pour comparer les résultats obtenus avec ceux d'un scénario complet. L'inconvénient de cette approche est d'être limité aux systèmes ayant des réactions complètement déterministes et séquentielles.

- **Tests à partir de cas d'utilisation [Frö00] :** Les cas d'utilisation accompagnés du pré et post-conditions sont transformés en un diagramme d'état-transition, puis un ensemble de contraintes sont écrites en utilisant un langage de planification. Et enfin l'outil graphplan est utilisé pour générer des tests en se basant sur les contraintes de planification.

- **Tests à partir de gardes des transitions [OFF99] :** Une technique permettant de générer des jeux de données pour tester les différentes conditions apparaissant dans les gardes des transitions des automates à états.

- **Tests pour des systèmes concurrents et non déterministes [GUN01] :** L'auteur propose une génération automatique de tests permettant de vérifier la conformité d'une implantation vis-à-vis de sa spécification. L'algorithme de synthèse des cas de tests a été spécialement conçu pour traiter des systèmes fortement concurrents et non déterministes.

3.2 Vérification par traduction en un langage formel

Traduire un sous-ensemble de diagrammes d'UML vers un langage formel permet de donner une sémantique au sous-ensemble UML en question et d'utiliser les techniques formelles offertes par le langage cible. Dans la majorité des cas, les travaux qui ont opté pour cette approche, s'attachent plus particulièrement à la traduction des diagrammes d'état-transitions (appelés aussi statecharts) de UML.

Le reproche majeur de cette approche est l'écart important existant entre le formalisme d'UML et le formalisme du langage cible. Cette écart impose généralement, des restrictions fortes sur les constructions UML qu'il est possible d'utiliser [GUN01].

Les principaux travaux appartenant à cette approche sont :

3.2.1 Traduction vers Promela

C'est le langage de spécification de l'outil SPIN [HOL97]. Le travail présenté dans [PAL99] montre que les diagrammes d'état-transition d'UML peuvent être traduits vers le langage Promela, ce langage permet de représenter des automates communiquant entre eux. L'outil SPIN permet de vérifier des propriétés de la logique temporelle à partir de ces automates. Cette traduction se limite à des spécifications relativement statiques (la version SPIN utilisée n'autorise pas l'usage de constructions dynamiques).

3.2.2 Traduction vers SMV[1]

SMV est un model checking équipé de deux langages de modulation (Extended SMV et synchronous verilog). Le travail présenté dans [KWO00] montre comment SMV permet de faire du model checking à partir des diagrammes d'état-transition d'UML en utilisant différentes formes de spécifications : CTL et LTL. Il est à noté que cette approche ne considère qu'un seul statechart, en isolation, sans envisager de communications inter-objets.

3.2.3 Traduction vers LOTOS

LOTOS [ISO85] est un langage permettant de décrire formellement les systèmes. Il s'agit d'un langage de type algèbre de processus. RT-LOTOS étend le langage LOTOS par des opérateurs temporels (délai, latence et offre limitée dans le temps). Dans [CAR00], il s'agit de la traduction d'un sous ensemble de UML vers le langage de spécification LOTOS. Cependant, les notions de références, de liens dynamiques entre objets, et de création d'objets ne sont pas prises en compte par la traduction, ce qui limite cette approche à des spécifications relativement statiques.

[1] mcmillan@cadence.com, smv-users@cadence.com.

3.2.4 Traduction vers SDL

Les deux langages SDL [CCI87] et UML sont assez proches, ce qui rend possible la traduction de l'un à l'autre. L'ObjectGeode[1] est un ensemble d'outils dédiés à l'analyse, la conception, la vérification et la validation par simulation. L'ObjectGeode permet de traduire les diagrammes d'état-transition d'UML en un processus du langage SDL. Puis, le SDL permet de simuler le comportement de la spécification.

3.2.5 Traduction vers B

B [ABR96] est une méthode de spécification formelle qui permet, à travers un langage adéquat appelé le langage B [CLE03], d'exprimer très précisément des propriétés exigées par le développeur. L'outil ArgoUML+B [LED03] permet la transformation automatique d'un certain nombre de diagrammes UML en B en suivant les définitions proposées dans [MEY01] et [LED02]. Des travaux ont proposé des techniques de validation des propriétés de construction d'un scénario UML/OCL à partir de sa dérivation B [THU04]. Le processus de preuve qui associe la modélisation UML/OCL avec l'outil de preuve de l'Atelier B est donné dans la figure 3.1.

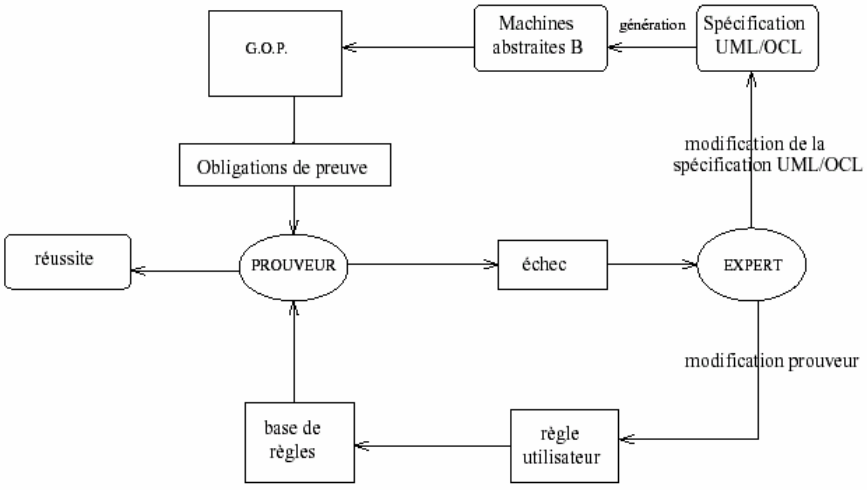


Fig 3-1 Processus de preuve de l'atelier B.

[1] http://www.verilogusa.com/products/geode.htm

3.3 Utilisation des Réseaux de Pétri

Dans [PAL99b], il est prouvé que les aspects dynamiques d'UML peuvent être modélisés par des réseaux de Pétri. Un aspect original de ces travaux est une méthode pour décomposer un sous système en collaboration d'objets plus petits en analysant les "invariants de place" du réseau de Pétri du sous-système.

3.4 Utilisation d'UML lui-même

Dans [CLA01], le groupe *precise UML* (*pUML*) propose certains travaux appartenant à cette catégorie, ainsi que la proposition du groupe de travail sur l'Action Semantics [OMG03] qui travaille à l'élaboration d'une sémantique qui sera officiellement adoptée pour UML par l'Object Management Group. L'Action Semantics représente l'un des efforts le plus important pour donner à UML les bases formelles permettant de lui appliquer des techniques formelles.
Le travail proposé par [GUN01] utilise aussi cette approche en plus de la génération de tests.

3.5 Utilisation du système FoCAL

Dans [ETI05] un algorithme de traduction des spécifications FoCAL vers UML a été proposé. L'objectif est de permettre aux membres du groupe (au sein de Lip6) de comprendre facilement la spécification FoCAL faite dans le cadre du projet intitulé : "*modélisation de la réglementation de l'aviation civile en FoCAL*".
La traduction présentée se limite aux diagrammes de classes. Dans cette traduction, une classe UML est dérivée à partir d'une espèce FoCAL, les relations d'héritages entre espèces sont préservées entre les classes et les opérations de classes sont déduites de celles des espèces. Les diagrammes de classes sont obtenus en suivant le schéma présenté dans la figure ci-dessous :

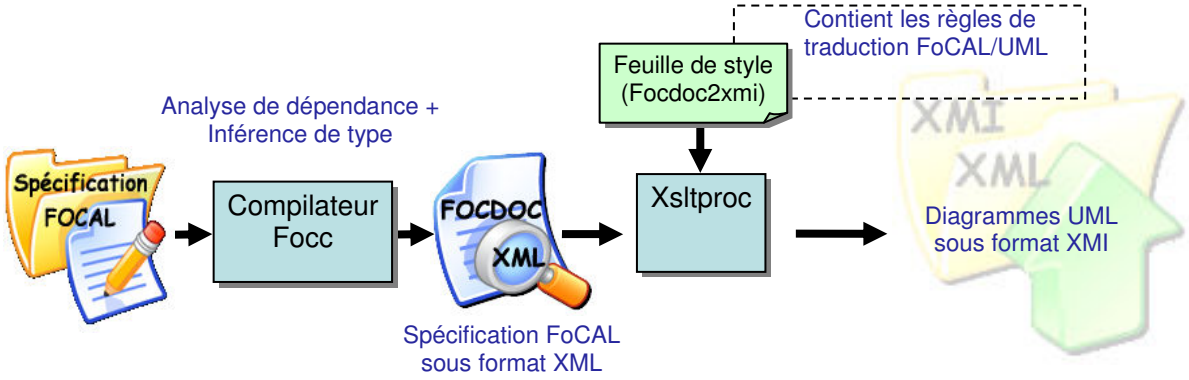


Fig 3- 2 Processus de traduction systématique de FoCAL vers un diagramme de classes UML [ETI05].

On s'intéresse dans notre travail à la démarche inverse à savoir, une transcription d'une modélisation UML/OCL vers une spécification FoCAL. L'intérêt de cette démarche étant de conserver les avantages du langage graphique UML pour représenter des systèmes de manière synthétique et intuitive, tout en disposant d'un cadre plus formel permettant de valider ses propriétés.

Dans le chapitre suivant, une démarche de transcription ainsi qu'une technique de vérification sont présentées.

Chapitre IV

De UML vers FoCAL

Dans ce chapitre nous allons présenter une démarche de transcription des spécifications UML vers le système FoCAL. C'est-à-dire la traduction d'un ensemble de diagrammes UML vers une hiérarchie d'espèces FoCAL. L'objectif de cette démarche est l'utilisation des techniques de preuves formelles offertes par FoCAL pour vérifier des propriétés conditionnant la cohérence du modèle UML.
Notre démarche, influencée par le passage inverse de FoCAL vers UML (*cf.3.5*) s'inspire de la traduction d'UML vers B *(cf.3.2.5).*
Le chapitre introduit dans un premier temps, une démarche globale de transcription, puis dans un second temps, les détails de cette démarche seront illustrés à travers l'exemple classique de "contrôle d'ascenseurs".

4.1 Démarche globale de transcription

Lors de la transformation des spécifications UML vers le système FoCAL, nous considérons un modèle UML/OCL composé de diagrammes UML et de contraintes OCL.
Deux sortes de diagrammes UML seront traitées :

- Les diagrammes de classes : représentant l'aspect statique du modèle. A partir de ces diagrammes, les variables d'instances, les opérations de classes, les dépendances et les associations inter classes seront traduites.
- Les diagrammes d'état-transition : représentant l'aspect dynamique du modèle. A partir de ces diagrammes, des types de données, des contraintes et des propriétés seront générées lors de la transcription.

Les contraintes OCL considérées sont :

- Les invariants de classes.
- Les contraintes de pré-conditions et post-conditions pour les opérations d'une classe.

Les grandes lignes de notre démarche peuvent se résumer comme suit :
Une espèce FoCAL sera dérivée à partir d'une classe UML. Le type support de l'espèce dérivée doit être un enregistrement qui représente l'état de l'objet (ses variables d'instance).

Les méthodes de classes seront reportées vers les espèces dérivées. Et enfin, nous ajoutons une fonction supplémentaire qui doit correspondre à la méthode "*new*" (de l'orienté objet) pour créer un objet.

La collection quant à elle, correspondra à un ensemble d'objets. A chaque fois qu'on souhaite créer un objet on fait appel à la méthode "*new*" de la collection.

Les contraintes de pré-condition et post-condition d'une opération de classe correspondront à des propriétés FoCAL. Ces dernières seront intégrées dans l'espèce dérivée de la classe en question et utilisées lors de la preuve. Des théorèmes seront dérivés automatiquement pour prouver l'absence de toute contradiction entre les diagrammes d'état-transition et les contraintes OCL.

4.2 Dérivation de diagrammes de classes

Un diagramme de classes d'un modèle UML sera transformé en une hiérarchie d'espèces FoCAL, où les espèces correspondront aux classes. Les relations de généralisation/ spécialisation (héritage) entre les classes seront préservées entre les espèces. Les classes concrètes (terminales) correspondront à des collections.

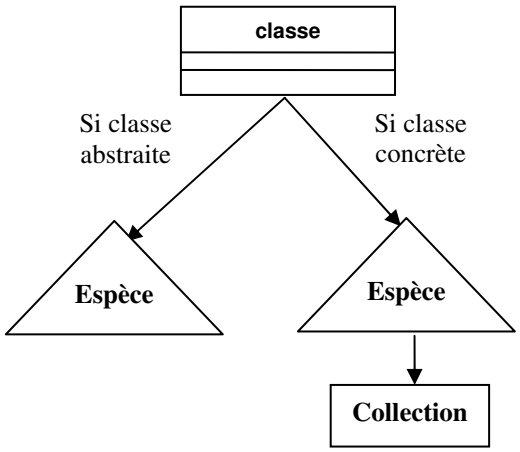


Fig 4- 1 Dérivation des diagrammes de classes

Considérons l'exemple d'un diagramme de classes qui modélise l'aspect structurel du système de contrôle d'ascenseurs.

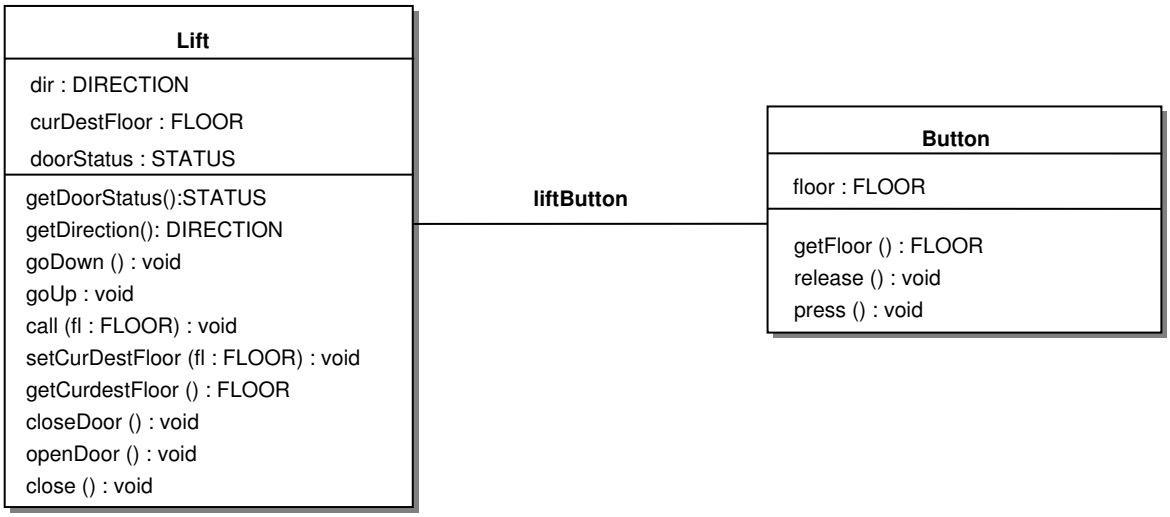


Fig 4- 2 Le diagramme de classes du système de contrôle d'ascenseurs

Le diagramme comporte deux classes : **Lift** pour les ascenseurs et **Button** pour les boutons. Les attributs de la classe **Lift** sont :

Dir la direction courante de l'ascenseur : **{up,down}**
curDest_Floor l'étage de destination courante de l'ascenseur : **ground..top**
doorStatus l'état de la porte de l'ascenseur : **{opened,closed}**

et l'attribut de la classe **Button** est :

floor l'étage associé à chaque bouton : **ground..top**

La hiérarchie des espèces dérivées à partir de ce diagramme (sans tenir compte des associations) est la suivante :

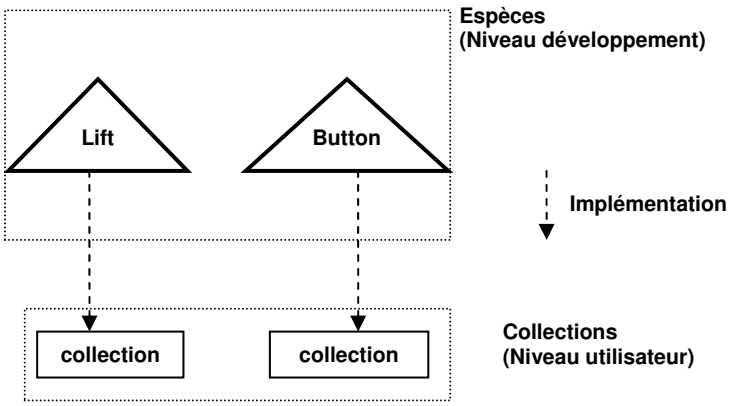


Fig 4- 3 La hiérarchie des espèces du système de contrôle d'ascenseurs

Supposons maintenant que les deux classes Lift et Button héritent d'une même classe (abstraite) appelée Object, le diagramme de classes de l'exemple précédant devient comme suit :

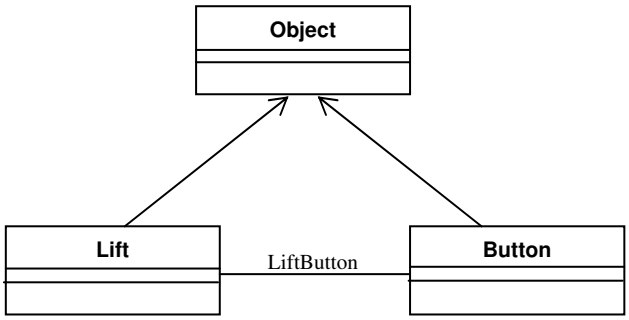


Fig 4- 4 Diagramme de classes du système de contrôle d'ascenseurs en ajoutant la classe Object

La hiérarchie des espèces dérivées à partir de ce diagramme est la suivante :

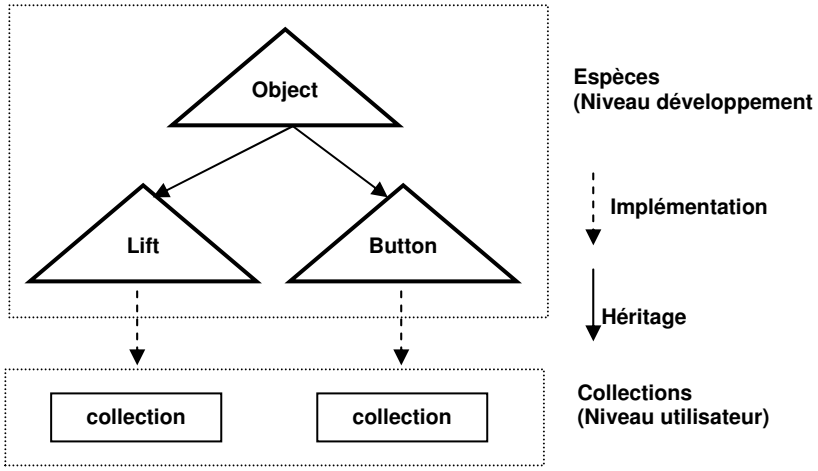
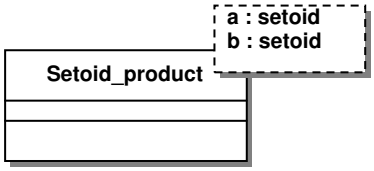


Fig 4- 5 Hiérarchie des espèces du système de contrôle d'ascenseurs en ajoutant l'espèce Object

Une classe paramétrée sera transformée en une espèce paramétrée :

Classe paramétrée	*Espèce paramétrée*
	```***species setoid_produit*** ***(a is setoid, b is setoid)*** ***inherits setoid =*** ***...*** ***end***

Pour transcrire les attributs de classes,

- un type UML est traduit directement vers un type FoCAL, dans le cas où il existe un type de données équivalent en FoCAL. Par exemple le type entier sera exprimé par le type FoCAL *int*.
- Si le type UML n'a pas d'équivalent en FoCAL, il sera exprimé par un type Ocaml, puis ce type sera importé par FoCAL en utilisant l'instruction FoCAL suivante :

```
type type_FoCAL_à_definir = caml_link type_ocaml_importé ;;
```

Par exemple, au type *DIRECTION* = {up, down}, correspond dans FoCAL un type énuméré :

```
type direction =
 Up in direction;
 Down in direction;
```

Dans le cas d'un type UML générique, le type FoCAL correspondant est abstrait. S'agissant d'un type intervalle, les deux bornes de l'intervalle seront considérées comme deux constantes FoCAL et le type intervalle sera interprété par la définition de deux propriétés (<= et >=) garantissant l'appartenance à l'intervalle.

Dans l'exemple proposé, pour le type intervalle *FLOOR = ground .. top*, nous déclarons deux constantes *floor_ground* et *floor_top*, puis nous définissons les propriétés comme suit :

```
sig floor_top in floor;
sig floor_ground in floor;

sig floor_leq in floor -> floor -> bool;
sig floor_geq in floor -> floor -> bool;
...
```

Tous les types de données et les constantes définis peuvent être déclarés au niveau top ou bien stockés dans un fichier *type_file.foc* qui sera référencé et utilisé par les espèces dérivées, auquel cas la spécification FoCAL doit commencer par :

```
Uses  type_file;;
Open  type_file;;
```

4.2.1 Les attributs de classes

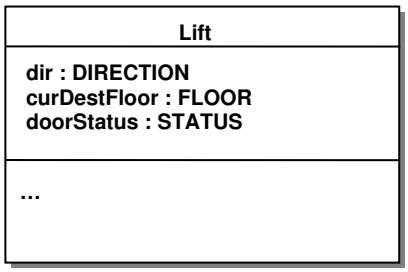
Un type enregistrement est créé à partir des types des variables d'instances d'une classe. Il représente le type support (*rep*) de l'espèce dérivée de la classe. Deux possibilités se présentent pour créer ce type enregistrement :

- soit l'utilisation d'un type enregistrement Ocaml qui sera importé par FoCAL, comme indiqué dans la section précédente,
- soit la création d'un type structure FoCAL par produit cartésien des différents types des variables d'instances.

4.2.1.1 Utilisation d'une structure Ocaml

Pour mieux illustrer le cas de l'utilisation d'une structure Ocaml, reprenons notre classe ascenseur (Lift) :

Les attributs et la méthode new	*Type support et la fonction new*


`P = new Lift(down,2,close);`
	`(* type Ocaml *)`
	`type Lift = {dir:direction ;`
	`curDestFloor : floor ;`
	`doorStatus : status}`
	`(* type importé par FoCAL *)`
	`type Lift = caml_link Lift; ;;`
	
	`species lifts =`
	`rep = Lift;`
	`sig new_lift in direction-> floor -> status`
	`-> self;`
	`...`
	`end`
	`let L = Lift_collection!new_lift(down,2,close);`

où ***type Lift = {dir : direction ; curDestFloor : floor ; doorStatus : status }*** : représente la structure de données Ocaml (qui sera importée par FoCAL).

4.2.1.2 Utilisation du produit cartésien

Pour illustrer le deuxième cas, à savoir l'utilisation du produit cartésien, le type support de l'espèce Lift dérivée de la classe Lift sera représenté par :

```
rep = direction*floor*status
```

tel que les types **direction, floor** et **status** sont définis comme suit :

```
type direction =
 Up in direction;
 Down in direction;
;;

type floor = caml int;
;;

type status =
 Open in status;
 Close in status;
;;
```

On obtient l'espèce Lift dérivée de la classe Lift suivante :

```
species lift =
  rep = direction*floor*status;
  sig floor_top in floor;
  sig floor_ground in floor;

  sig floor_leq in floor -> floor -> bool;
  sig floor_geq in floor -> floor -> bool;
  ...
  sig getDirection in self ->direction;
  sig getDoorStatus  in self ->status;
  sig getCurDestFloor in self -> floor;
  ...
end
```

4.2.2 Relations inter classes

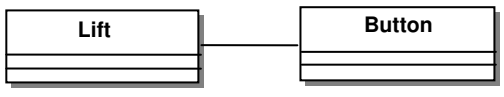
Association et dépendance : les associations et les dépendances inter classes seront réalisées dans un premier temps par la création d'une espèce paramétrée par les espèces dérivées des classes en relation. Puis dans un second temps, par la définition des propriétés entre les instances des espèces correspondant aux classes.

Dans l'exemple du système de contrôle d'ascenseurs, nous créons d'abord l'espèce lift_button paramétrée par les deux espèces Lift et Button. Puis, nous définissons une relation sous forme de couples (Lift , Button) :

```
species lift_button  (l is lift, b is button ) =

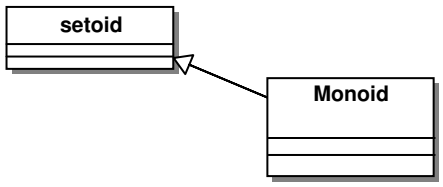
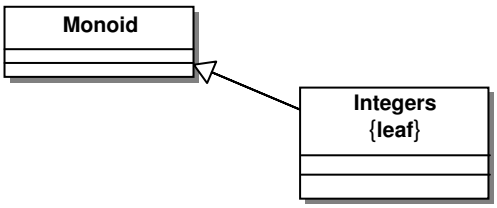
rep = l*b;
sig lift_button_assoc in (list(self));
...
end
```

Association et dépendance	*paramétrage*

	`species lift_button (l is lift, b is button ) =`
	`rep = l*b;`
	`sig lift_button_assoc in (list(self));`
	`...`
	`end`

Héritage : La relation d'héritage entre classes est préservée au niveau de la hiérarchie d'espèces dérivées. Dans le cas où une classe terminale (complète) *CB* hérite d'une classe

CA, la relation d'héritage peut être transcrite directement vers une implémentation (collection).

Relation d'héritage (1)	*La clause **inherits***
	```***species monoid*** ***inherits setoid =*** ***...*** ***End***
Relation d'héritage (2)	*La clause **implements***
	```***collection integers*** ***implements monoid =*** ***...*** ***End***

4.2.3 Les opérations de classes

Les opérations d'une classe seront reportées vers les espèces dérivées. Les opérations virtuelles seront traduites vers des méthodes déclarées en utilisant le type support de l'espèce dérivée. Les opérations définies, quant à elles, seront traduites vers des méthodes définies. Enfin, les opérations récursives seront traduites vers des méthodes récursives. La méthode "new" sera définie dans toutes les collections de la hiérarchie des espèces pour permettre la création des instances des collections correspondantes.

Illustrons d'abord les opérations de l'espèce Lift dans le cas où nous utilisons une structure Ocaml comme type support :

```
Lift = {dir:direction ; curDestFloor : floor ; doorStatus : status}
let getDirection L = L.direction
let getDoorStatus L = L.doorStatus
...
let new_Lift d f s ={dir = d ; curDestFloor = f ; doorStatus = s}
```

La spécification FoCAL derivée est :

```
type Lift = caml_link Lift; ;;
let getDirection ( L in Lift) in direction = caml getDirection;;
let getDoorstatus ( L in Lift) in status = caml getDoorstatus;;
...
species lift =
rep = Lift;
let new_Lift (d in direction, f in floor, s in doorStatus) in Lift =
                                                    caml new_Lift;;
...
```

```
let getDirection ( L in self) in #direction = #getDirection (L);
let getDoorstatus ( L in self) in #status = #getDoorStatus (L);
...
let new_Lift (d in #direction, f in #floor, s in #doorStatus) in self =
                                                    #new_Lift(d,f,s);
...    end
```

Dans le cas de l'utilisation du produit cartésien pour le type support, les opérations seront transcrites comme suit :

```
  species lift =

  rep = direction*floor*status;

  sig floor_top in floor;
  sig floor_ground in floor;
  sig floor_leq in floor -> floor -> bool;
  sig floor_geq in floor -> floor -> bool;
  ...
  sig getDirection in self ->direction;
  sig getDoorStatus  in self ->status;
  sig goDown in self -> unit;
  sig goUp in self -> unit;
  sig call in self -> floor -> unit;
  sig setcurDestFloor in self -> floor-> unit;
  sig getCurDestFloor in self -> floor;
  sig closeDoor in self -> unit;
  sig openDoor in self -> unit;
  sig close in self -> unit;
  sig new_lift in direction -> floor -> status -> lift_status -> self;
...
end
```

Les fonctions: **getDirection**, **getDoorStatus**, **getCurDestFloor...** seront implémentées en utilisant les deux fonctions **basics#first** et **basics#scnd** permettant d'extraire respectivement le premier et le deuxième élément d'une paire. De même, nous utiliserons la fonction **basics#crp** pour composer un tuple à partir des éléments de base.

Classe abstraite et Opérations	*Espèce et Fonctions*

Lift
dir : DIRECTION
curDestFloor : FLOOR
doorStatus : STATUS
getDoorStatus () :STATUS
getDirection (): Direction
goDown () : void
goUp : void
call (fl : FLOOR) : void
setCurDestFloor (fl : FLOOR) : void
getCurDestFloor () : FLOOR
closeDoor () : void
openDoor () : void
close () : void
arrive (fl : FLOOR) : void
	`species lift =`
	`  rep = direction*floor*status;`
	
	`  sig getDirection in self ->direction;`
	`  sig getDoorStatus  in self ->status;`
	`  ...`
	`  sig setCurDestFloor in self -> floor->unit;`
	
	`  sig close in self -> unit;`
	`  sig new_lift in direction -> floor -> status ->`
	`  lift_status -> self;`
	`...`
	`end`

Les deux solutions (type Ocaml importé ou produit cartésien FoCAL) sont réalisables, cependant l'utilisation du produit cartésien convient le mieux du point de vue automatisation de la transcription.

4.3 Dérivation de diagrammes d'état-transition

Dans notre transcription nous nous intéressons principalement aux états, aux transitions entre états dans un diagramme d'état-transition, et aux événements de communications entre les diagrammes d'état-transition. Cette transcription sera illustrée à travers les diagrammes d'état-transition des classes Button et Lift du système de contrôle d'ascenseurs introduit précédemment.

La figure 4.6 présente le diagramme d'état-transition de la classe **`Button`**.

Fig 4- 6 Diagramme d'état-transition de la classe Button

Le diagramme d'état-transition de la classe **Lift** (figure 4.7) a trois états **ready**, **visit** et **movement**.

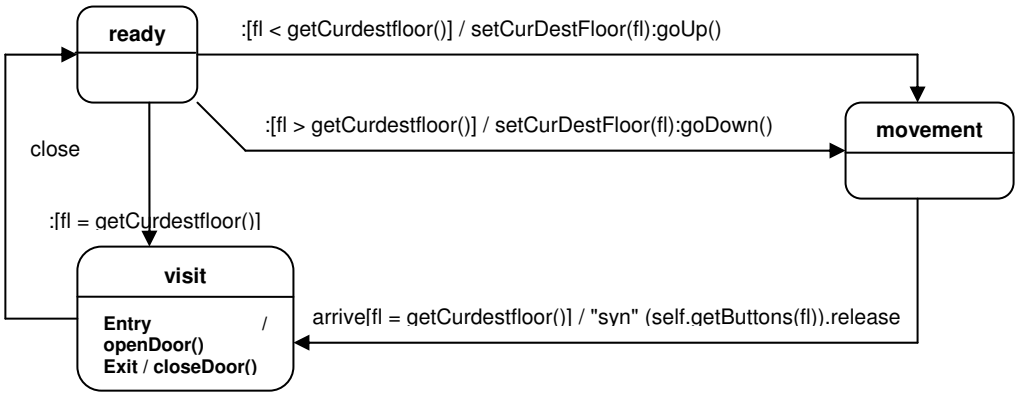


Fig 4- 7 Diagramme d'état-transition de la classe Lift

Un diagramme d'état-transition donne lieu à la création d'un ensemble énuméré modélisant l'espace des états possibles des instances de la classe concernée. L'état courant d'une instance

sera modélisé par l'ajout d'un champ supplémentaire à l'enregistrement modélisant le type support. Ainsi, le champ **lift_status** qui représente l'état courant d'une instance de l'espèce Lift dans son diagramme d'état-transition sera ajouté au type support de l'espèce Lift.

```
rep = direction*floor*status*lift_status;
```

Un ensemble énuméré LIFT_STATE =*{Visit, Ready, Mouvement}* sera créé comme suit :

```
type lift_status =
    Ready in lift_status;
    Movement in lift_status;
    Visit in lift_status; ;;
```

La fonction :

```
sig get_current_status in self -> lift_status;
```

sera ajoutée à l'espèce Lift pour récupérer l'état courant de l'instance dans le diagramme d'état-transition.

De même pour la classe Button l'ensemble correspondant à son diagramme d'état-transition est le suivant

```
type button_state =
   On in button_state;
   Off in button_state;
;;
```

4.3.1 Les transitions

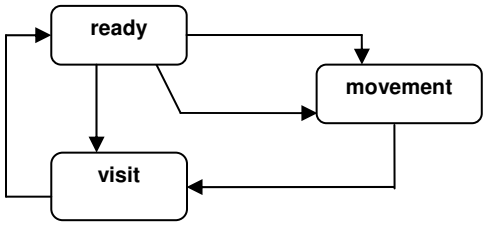
Une transition est dérivée en FoCAL par une méthode définie dont le rôle est de réaliser le changement d'état. Cette méthode est paramétrée par un argument qui modélise l'objet cible (de type self à l'intérieur de l'espèce) de l'événement déclanchant la transition. Cette méthode est définie dans l'espèce dérivée de la classe en question. Par exemple, la méthode définie *Visit_to_Ready* modélise la transition de l'état *Visit* à l'état *Ready* du diagramme d'état-transition de la classe Lift. Cette méthode est définie dans l'espèce Lift dérivée de la classe de même nom. Le type de cette méthode sera : **self -> self**, qui exprime une fonction dont le paramètre est une instance de l'espèce, et retourne la même instance augmentée d'un nouvel état (l'état *Ready*).

sig visit_to_ready in self -> self;

La définition de la méthode représentant la transition, sera préfixée par une condition sur l'état courant de l'instance. Par exemple, avant d'appliquer la transition *Visit_to Ready*, il est nécessaire de s'assurer que l'état courant de l'instance est égal à *"Visit"* :

```
If (current_state = "Visit") then … ;
```

Les états / transitions	*Ensemble énuméré / fonctions*

	`type lift_status =`
	`  Ready in lift_status;`
	`  Movement in lift_status;`
	`  Visit in lift_status;`
	
	`sig get_current_status in self->lift_status;`
	
	`sig visit_to_ready in self -> self;`
	...

4.3.2 Evénements et communications

Chaque événement dans les diagrammes d'état-transition est représenté par la définition d'une méthode qui fait appel aux méthodes représentant les transitions et celles représentant les actions (les opérations de classe).

Par exemple, l'événement *close* du diagramme d'état-transition de la classe Lift, fait appel à la méthode *closeDoor* dérivée de l'opération *closeDoor* de la classe Lift, puis à la méthode *Visit_to_Ready*, representant la transition de l'état *Visit* à l'état *Ready*.

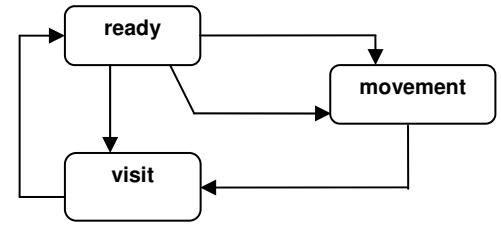
Il y a des événements qui établissent une communication entre deux classes, et par conséquent entre deux espèces : un des événements d'un diagramme d'état-transitions d'une classe peut faire appel aux opérations d'une autre classe. C'est le cas de l'événement *press* dans le diagramme d'état-transition de la classe Button, qui fait appel à l'opération *call* de la classe Lift. Le problème est résolu par la création d'une espèce paramétrée par les classes en communication.

```
species lift_button  (l is lift, b is button ) =
rep = l*b;
...
```

En utilisant cette espèce, nous pouvons définir des opérations, des propriétés et des théorèmes qui communiquent entre les espèces. Nous distinguons deux types de communications entre les diagrammes d'état-transition : synchrone et asynchrone. L'événement qui provoque un message synchrone d'un diagramme d'état-transition à un autre et l'événement résultat de cette communication sont dépendants. C'est-à-dire qu'à la réception du message, l'événement résultat est suivi par le traitement. Il est ainsi tout à fait naturel de considérer que l'effet de l'événement qui initie la communication synchrone, inclut l'effet de l'événement résultat. Donc il est inutile de modéliser l'événement résultat. Par exemple l'événement *arrive* dans le diagramme d'état-transition de la classe Lift

provoque deux événements *release* dans un message synchrone à deux objets Button (un bouton dans l'ascenseur et l'autre dans l'étage). Ainsi, les effets de ces deux événements *release* sont incorporés dans la méthode *arrive* définie dans l'espèce paramétrée par les deux espèces Lift et Button.

Dans le cas d'une communication asynchrone, l'événement qui provoque un message asynchrone et l'événement résultat sont indépendants. Lorsque le message est envoyé, l'événement résultat est enregistré mais le traitement de cet événement peut être différé. Pour cela, nous proposons d'utiliser une file d'événements pour chaque type de message asynchrone pour simuler la communication asynchrone. Dans le système de contrôle d'ascenseurs, il s'agit de l'événement *call* survenu lors d'un message asynchrone envoyé par l'événement *press*. La file d'événements *call* est modélisée par une liste **file_call** dont les éléments sont traités selon la stratégie premier arrivé premier servi (*FIFO*).

Communication entre diagrammes d'état-transition	*Paramétrage, file (sous forme d'une liste)*
Lift  **Press / "asyn" (self.getLift()).call(self.getFloor())** ***Button***  	**species lift_button** **(l is lift, b is button) =** **rep = l*b;** **(** file pour l'événement call **)** **sig file_call in list(floor);** **sig enfiler_call in** **list(floor)->floor->unit;** **sig defiler_call in** **list(floor) -> floor;** **...** **end**

4.4 Dérivation de contraintes OCL

Nous nous intéressons maintenant aux invariants de classes et aux pré-conditions et post-conditions pour les opérations.

Les invariants de classes seront transformés vers des propriétés FoCAL équivalentes définies dans les espèces dérivées. Par exemple, pour exprimer le fait que la variable *floor* de la classe Button doit être supérieure ou égale à *floor_ground* et inférieure ou égale à *floor_top.* La contrainte OCL équivalente sera transformée vers une propriété FoCAL dans l'espèce Button comme l'indique le tableau suivant :

Contraintes OCL (Invariant)	*Propriétés*
context Button ***inv: (floor >= floor_ground) And*** ***(floor <= floor_top).***	***species Button inherits basic_object =*** ***...*** **sig floor_leq in floor -> floor -> bool;** **sig floor_geq in floor -> floor -> bool;** **property invariant1 : all x in self,** **(!floor_leq(!getCurDestFloor(x),** **!floor_top)) and** **(!floor_geq(!getCurDestFloor(x),** **!floor_ground));**

D'une manière générale, pour toute fonction de la forme $f(x_1, x_2, x_3, \ldots)$, dont l'un des arguments est de type *self*, et qui retourne un résultat de type *self*, il faut prouver que l'invariant est préservé (satisfait) après exécution de cette fonction.
Le compilateur FoCAL ne traite pas les invariants. Assurer la satisfaction d'un invariant au sein d'une espèce nécessite l'ajout de certains mécanismes au compilateur FoCAL lui-même.

Quant aux contraintes pré-condition et post-condition, elles seront exprimées par une instruction *if* au sein des opérations :

```
If (pre_condition) then (post_condition) else error.
```

Enfin, une propriété sera déduite et intégrée dans l'espèce concernée pour l'utiliser comme un axiome (assumed) dans la preuve. Le tableau ci-dessous présente la transcription d'une contrainte OCL associée à la fonction " visit_to_ready " du diagramme d'état-transition de la classe Lift :

Contraintes OCL (pré et post-conditions)	*Propriétés*
Context ***Lift:: visit_to_ready ():void*** ***Pre:*** ***self.status = Visit*** ***Post:*** ***self.status = Ready***	***Let visit_to_ready (x) =*** ***if(!egale(x,!Visit))then(!Ready)*** ***else(!foc_error(""));*** ***property visit_to_ready_prop :all x in*** ***self,*** ***(!egale(x,!Visit))->*** ***(!egale((!visit_to_ready (x)),!Ready));***

4.5 Processus de preuves

Une spécification peut être insatisfaisante du point de vue de la preuve pour les deux raisons:

- Soit la spécification est incomplète, c'est-à-dire il y a un manque d'informations (par exemple : "démontrer qu'une variable est toujours paire sans avoir d'information sur ses valeurs potentielles"[MAR97]).
- Soit la spécification est incohérente (par exemple: "démontrer qu'une valeur appartient à l'ensemble vide [THU04]").

Dans les deux cas, l'anomalie conduit à un échec de preuve.

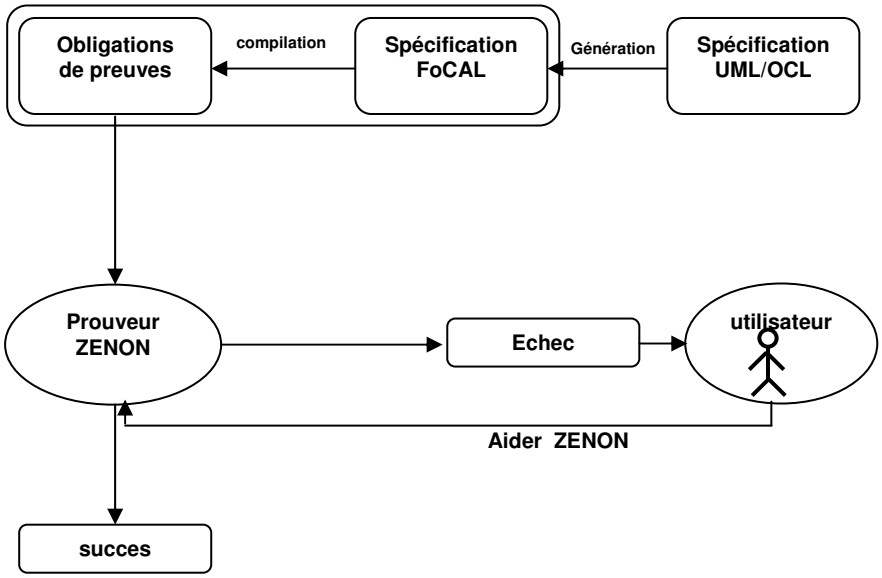


Fig 4- 8 Processus de preuve

La figure 4-8 illustre le processus de preuve qui associe le modèle UML/OCL avec l'outil de preuve Zenon. A partir d'un modèle UML/OCL, une spécification FoCAL est générée. Les obligations de preuves sont obtenues par compilation de la spécification FoCAL générée. Des propriétés et théorèmes sont déduits des diagrammes d'état-transition et des contraintes OCL. Il s'agit là de prouver, en utilisant Zenon, l'absence de toute contradiction entre les diagrammes d'état-transition de chaque classe et les contraintes OCL associées aux opérations de la même classe. Le séquencement d'appel d'opérations au niveau des diagrammes d'état-transition doit respecter les contraintes de pré-conditions et post-conditions correspondant aux mêmes opérations. Si Zenon indique un échec lors de la preuve d'un théorème, il faudra, selon le cas,

- soit corriger les indications de preuves données à Zenon,
- soit apporter des modifications à la modélisation UML/OCL.

En plus des obligations de preuves liées aux contraintes explicites (pré-condition, post-conditions, contraintes des associations, contraintes de diagrammes d'état-transition, …), le

compilateur de FoCAL ajoute des obligations de preuves lors de l'analyse des espèces. En particuliers, "toute construction récursive doit être faite en parallèle avec une preuve de sa terminaison [FOC06]".

4.6 Equivalence entre le modèle UML et le modèle FoCAL

L'équivalence entre la modélisation UML et la spécification FoCAL obtenue, peut être réalisée par une technique de traçabilité entre les deux modèles : Il faut permettre au concepteur de passer entre le modèle UML et le modèle FoCAL dans les deux sens. Cette équivalence permettra d'améliorer la lisibilité et l'évolution de la spécification [LAL02].
La figure 4-9 propose une architecture du passage bidirectionnelle entre une modélisation UML et une spécification FoCAL.

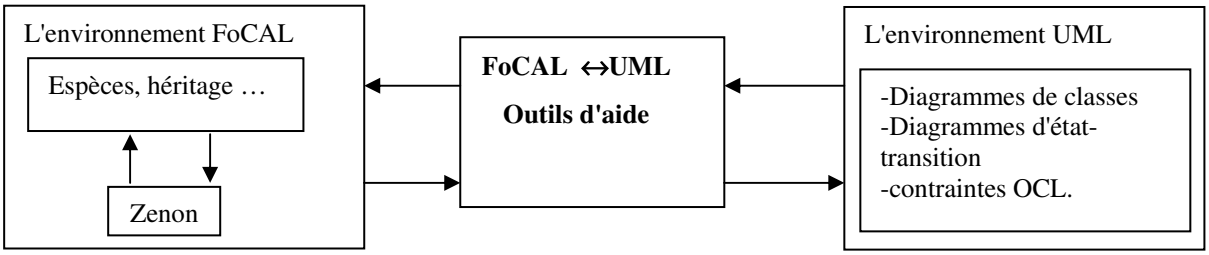
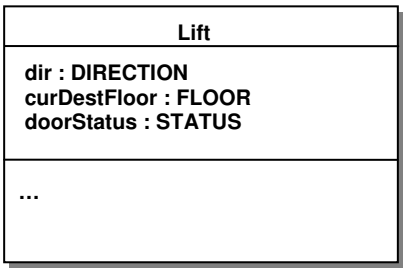


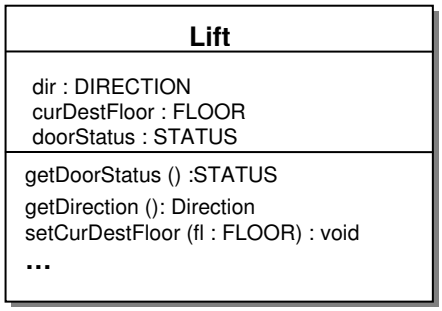
Fig 4- 9 Architecture du passage bidirectionnel entre une modélisation UML et une spécification FoCAL

4.7 Tableau récapitulatif de transcription

UML	FOCAL
Les attributs et la méthode new	*Type support et la fonction new*


`P = new Lift(down,2,close);`
	`(* type Ocaml *)`
	`type Lift = {dir:direction ;`
	`curDestFloor : floor ;`
	`doorStatus : status}`
	`(* type importé par FoCAL *)`
	`type Lift = caml_link Lift; ;;`
	`species lifts =`
	`rep = Lift;`
	`sig new_lift in direction-> floor -> status -> self;`
	`...`
	`end`
	`let L =`
	`Lift_collection!new_lift(down,2,close);`

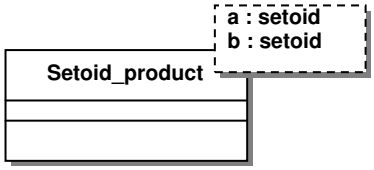
Classe abstraite et Opérations	*Espèce et Fonctions*




```
species lift =
  rep = direction*floor*status;

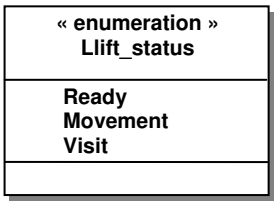
sig getDirection in self ->direction;
sig getDoorStatus  in self ->status;
sig setCurDestFloor in self -> floor->self;
...
sig new_lift in direction -> floor ->
          status -> lift_status -> self;
end
```

Classe paramétrée | *Espèce paramétrée*




```
species setoid_produit
         (a is setoid, b is setoid)
                        inherits setoid =
...
end
```

Classe dotée du stéréotype «enumeration» | *Types définis*




```
type lift_status =
    Ready in lift_status;
    Movement in lift_status;
    Visit in lift_status; ;;
```

Contraintes OCL(Invariant) | *Propriétés*

```
context Button
inv: (floor >= floor_ground) And
(floor <= floor_top ).
```

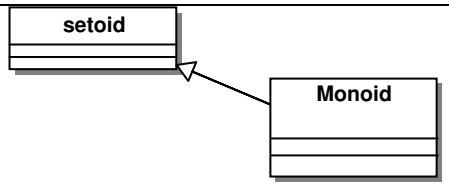
```
species Button inherits basic_object =
...
sig floor_leq in floor -> floor -> bool;
sig floor_geq in floor -> floor -> bool;
Property inv1:all x in self,
basics#and_b((basics#leq(floor_ground,x)),
            (basics#leq(x , floor_top)));
...
end
```

Contraintes OCL(pré et post-conditions) | *Propriétés*

```
Context
Lift:: visit_to_ready ():void
 Pre:
   self.status = Visit
 Post:
   self.status = Ready
```

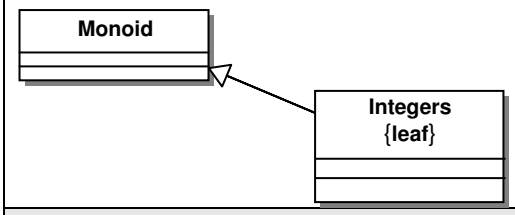
```
Let visit_to_ready (x) =
if(!egale(x,!Visit))then(!Ready)
                    else(!foc_error(""));
property visit_to_ready_prop :all x in
                                        self,
(!egale(x,!Visit))->
(!egale((!visit_to_ready (x)),!Ready));
```

Relation d'héritage (1) | *La clause **inherits***



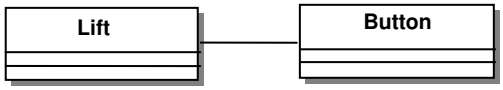

```
species monoid
            inherits setoid =
...
End
```

Relation d'héritage (2) — *La clause **implements***



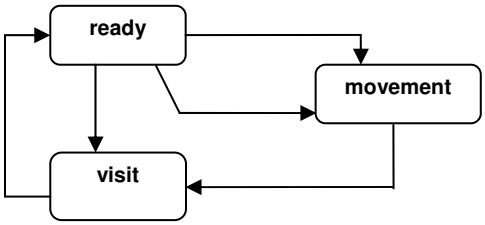

```
collection integers
            implements monoid =
...
End
```

Association — *paramétrage*




```
species lift_button  (l is lift, b is
button ) =
rep = l*b;
sig lift_button_assoc in (list(self));
sig lift_button in l -> b -> bool;
sig get_lift in b -> l;
...
end
```

Les états / transitions — *Ensemble énuméré / fonctions*



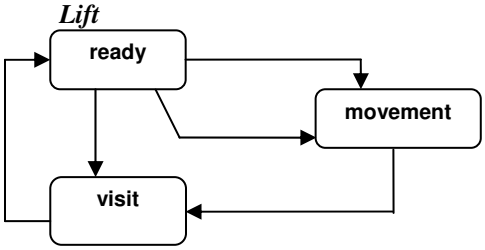

```
type lift_status =
    Ready in lift_status;
    Movement in lift_status;
    Visit in lift_status;

sig get_current_status in
                self->lift_status;

sig visit_to_ready in self -> self;
...
```

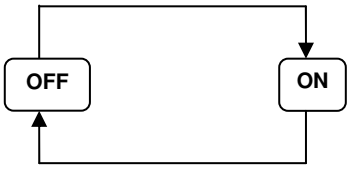
Communication entre diagrammes d'état-transition — *Paramétrage, file (sous forme d'une liste)*

Lift



Press / "asyn" (self.getLift()).call(self.getFloor())

Button




```
species lift_button
   (l is lift, b is button ) =
rep = l*b;

 (** file pour l'événement call **)

sig file_call in list(floor);

sig enfiler_call in
        list(floor)->floor->unit;

sig defiler_call in
        list(floor) -> floor;
...
end
```

Dans le présent chapitre nous avons proposé une démarche de transcription des spécifications UML vers le système FoCAL. La démarche proposée permet d'abord d'obtenir un modèle FoCAL à partir du modèle UML considéré. Puis, de valider la consistance du modèle.

Pour transcrire des spécifications UML vers le système FoCAL, nous nous sommes limités à un modèle UML/OCL composé par des diagrammes de classes et diagrammes d'état-transition. Les contraintes OCL considérées sont les invariants de classes, ainsi que la pré-condition et post-condition pour les opérations de classes. Toutefois, le modèle de transcription proposé peut être enrichi par l'introduction d'autres types de diagrammes UML tel que le diagramme de collaboration, et englobe aisément d'autres types de contraintes OCL telles que les contraintes conditionnelles.

La richesse de FoCAL en termes de spécification, nous a permis de surmonter l'écart existant entre le formalisme introduit par UML et celui du langage FoCAL. La spécification FoCAL obtenue est dynamique, dans la mesure où elle permet la création d'instances et préserve la communication entre des diagrammes d'état-transition.

Chapitre V

Etude de cas : système de contrôle de trains

Dans ce qui suit nous allons présenter un exemple pratique de modélisation UML et sa transcription en FoCAL, en utilisant la proposition présentée dans le chapitre précèdent.
Il s'agit de la transcription de la modélisation UML d'un système de contrôle de trains. Nous avons choisi cet exemple en raison sa simplicité d'un côté, et vu sa richesse en termes de contraintes de l'autre.

5.1 Transcription de la modélisation d'un système de contrôle de trains

Afin d'illustrer notre travail, nous allons traiter un système classique, le contrôle de barrière et de feu nécessaires pour un passage à niveau. Ce système se compose d'un contrôleur, d'un feu de signalisation et d'une barrière. Par souci de simplicité nous éliminons les objets de type véhicule et train de la présentation.
Notre système fonctionne comme suit:

- A l'état normal, le feu est vert, la barrière est ouverte. Quand le train arrive, le contrôleur donne une commande pour arrêter la circulation, le feu passe à l'orange puis au rouge et la barrière se ferme.
- Quand le train est passé, le contrôleur donne une commande pour remettre en route la circulation. La barrière s'ouvre et le feu passe au vert.

Le diagramme de classes du modèle UML associé à ce système se présente comme suit :

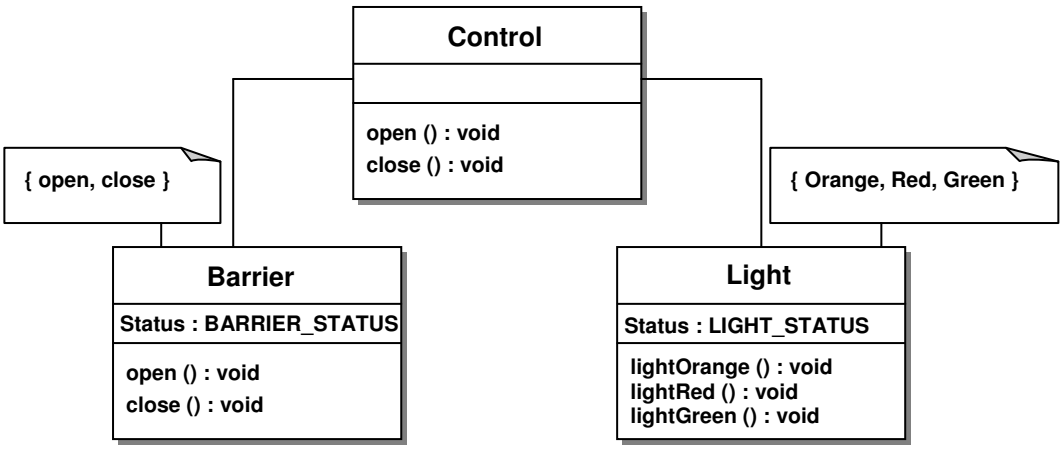


Fig 5-1 Diagramme de classes du système de control de trains

Ce diagramme contient trois classes : *Control*, Barrier et *Light* :

- La classe *Control* n'a pas d'attribut, elle a deux méthodes : *open* pour remettre la circulation et *close* pour arrêter la circulation.
- La classe *Barrier* a un seul attribut Status de type *BARRIER_STATUS* qui est un ensemble énuméré composé de deux valeurs : {*open*, *close*}. Cette classe dispose de deux méthodes : *open* et *close* pour ouvrir et fermer la barrière respectivement.
- La classe *Light* a un seul attribut Status de type *LIGHT_STATUS* qui est un ensemble énuméré composé des valeurs : {*Orange*, *Red*, *Green*}. Cette classe possède trois méthodes *: lightOrange*, *lightRed* et *lightGreen* pour mettre le feu en *Orange, Red* ou *Green* respectivement.

Les figures 5-2a, 5-2b et 5-2c présentent les diagrammes d'état-transition associés aux classes Control, Barrier et Light respectivement.

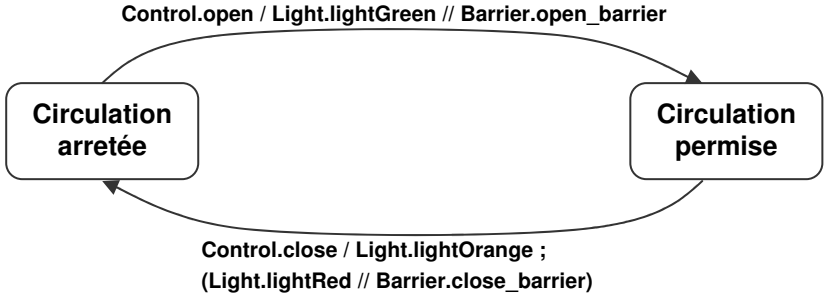


Fig 5- 2a Diagramme d'état-transition de la classe Control

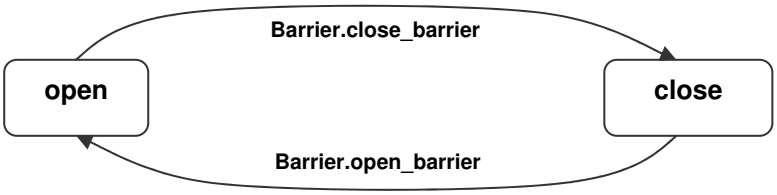


Fig 5- 2b Diagramme d'état-transition de la classe Barrier

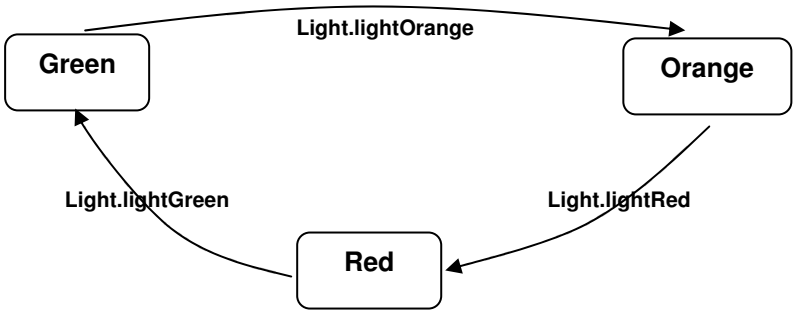


Fig 5- 2c Diagramme d'état-transition de la classe Light

Des contraintes OCL sont associées aux différentes opérations (pré-condition et post-condition) comme suit :

```
Context Control::close():void
 Pre:
   light.status = Green and
   barrier.status = open
 Post:
   Light.status = red and
   Barrier.status = close

Context Light::lightRed():void
 Pre:
   light.status = orange
 Post:
   Light.status = red

Context Light::lightGreen():void
 Pre:
   light.status = red
 Post:
   Light.status = green

Context
Barrier::open_barrier():void
 Pre:
   self.status = close
 Post:
   self.status = open

...
```

5.2 Les types supports (Rep) des espèces

Pour la classe Control il n'y a pas de types de données particuliers (n'a pas d'attributs) alors que les deux classes Barrier et Light ont chacune un attribut de type ensemble énuméré. Ces ensembles seront traduits vers des ensembles énumérés FoCAL comme suit :

```
type barrier_status =
   Opened in barrier_status;
   Closed in barrier_status; ;;

type light_status =
   Green in light_status;
   Red in light_status;
   Orange in light_status; ;;
```

Lors de la dérivation des espèces, le type support des espèces Barrier et Light seront respectivement barrier_status et light_status.

5.3 Hiérarchie des espèces

L'hiérarchie des espèces dérivées est la suivante :

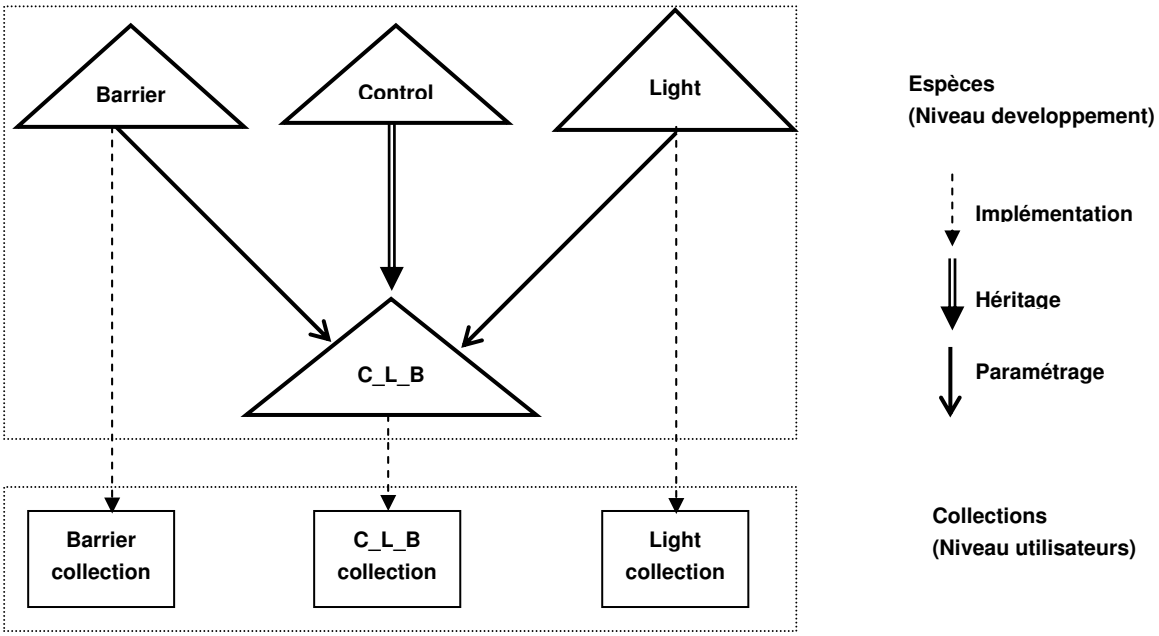


Fig 5- 3 Hiérarchie des espèces dérivées du système de contrôle de trains

Dans cette hiérarchie une espèce Barrier est dérivée à partir de la classe Barrier, une espèce Light est dérivée à partir de la classe Light et une espèce Control est dérivée à partir de la classe Control.

```
species barrier  =
rep = barrier_status;
sig egale in self ->self ->bool;
sig close_barrier in self -> self;
sig open_barrier in self -> self;
let ouverte in self = #Opened;
let ferme in self = #Closed;
sig new_barrier in
        barrier_status -> self ;
...
end
species light =
rep = light_status ;
sig egale in self ->self ->bool;
let vert in self = #Green ;
let orange in self = #Orange ;
let rouge in self = #Red;
sig lightOrange in self -> self;
sig lightRed in  self -> self;
sig lightGreen in self -> self;
let new_light(x in light_status)
                      in self = x;
...
End

species control =
rep;
```

```
sig egale in self ->self ->bool;
sig remettre_circulation in self -> self;
sig arreter_circulation in self -> self;
...
End
```

Les deux espèces Light et Barrier seront implémentées directement par les collections Light_collection et Barrier_collection :

```
collection  barrier_collection implements barrier =
...
let egale = #base_eq;
let new_barrier (x in barrier_status) in self = x;
let close_barrier(x) = if(!egale(x,#Opened)) then ( #Closed) else
(!foc_error(""));
let open_barrier(x) = if(!egale(x,#Closed)) then ( #Opened) else
(!foc_error(""));
...
proof ...
end

collection  light_collection implements light =
...
let egale = #base_eq;
let lightOrange(x)=if(!egale(x,!vert))then(!orange)else(!foc_error(""));
let lightRed(x)=if(!egale(x,!orange))then(!rouge)else(!foc_error(""));
let lightGreen(x)=if(!egale(x,!rouge))then(!vert)else(!foc_error(""));
Proof ...
end
```

L'espèce Control est définie de manière abstraite : le type support de cette espèce sera de la forme **rep.** Comme la classe Control n'a pas de données particulières à manipuler et que les méthodes de cette classe ont besoin d'utiliser les méthodes et les attributs des autres classes (Light et Barrier) il est nécessaire d'introduire une espèce appelée Control_Light_Barrier (C_L_B dans la figure 5-3) pour définir les méthodes et les propriétés de l'espèce Control et les association entre la classe Control et les classes Light et Barrier :

- l'espèce Control_Light_Barrier hérite de l'espèce Control et est paramétrée par les espèces Light et Control.
- Le type support de l'espèce Control_Light_Barrier sera **rep = l*b;** tel que **l : Light** et **b : Barrier**.

```
species control_light_barrier(l is light, b is barrier )inherits control=
rep = l*b;
let first ( x in self) in l = #first(x) ;
let scnd ( x in self) in b = #scnd(x) ;
sig egale in self -> self -> bool;
sig arreter_circulation in self -> self;
sig  remettre_circulation in self -> self;
let new_control(x in l, y in b) in self = #crp(x,y);
...
end
```

Les associations entre la classe Control et les classes Light et **Barrier** sont représentées par la définition de l'espèce Control_Light_Barrier.

Les méthodes **new_barrier**, **new_light** et **new_control** sont définies dans les espèces terminales (Barrier, Light, Control_Light_Barrier) pour permettre la création des instances des collections correspondantes. Par exemple l'instruction FoCAL suivante :

```
let feu = light_collection!new_light(#green)
```

permet de créer une instance de la collection Light_collection appelé "feu" initialisé par l'état "green".

5.4 Dérivation des contraintes OCL et des diagrammes d'état-transition

Les différents états possibles pour une instance de la collection Light_collection sont exactement les éléments de l'ensemble Light_Status. De même, pour les états des instances de la collection Barrier_collection sont les éléments de l'ensemble Barrier_Status. Ainsi il n'est pas nécessaire de créer de nouveaux ensembles pour modéliser les espaces des états possibles pour les deux collections. Ces ensembles sont alors les éléments de :

```
type barrier_status =
   Opened in barrier_status;
   Closed in barrier_status;
   ;;
```

et de :

```
type light_status =
   Green in light_status;
   Red in light_status;
   Orange in light_status; ;;
```

Les transitions de la classe Light sont représentées par les opérations : lightOrange , lightRed et lightGreen. Celles la classe Barrier_collection sont représentées par les opérations open_barrier et close_barrier.

Les états possibles pour les instances de la classe Control sont :

{ circulation_arrêté , circulation_permise}

Ces deux états seront définis dans l'espèce Control_Light_Barrier par les valeurs :

```
let vert_ouverte in self=#crp(l!vert,b!ouverte);(* circulation_permise *)
let rouge_ferme in self =#crp(l!rouge,b!ferme); (* circulation_arretée *)
```

Les transitions d'un état à un autre sont représentées par les opérations : **arreter_circulation** et **remettre_circulation** (voir l'Annexe pour plus de détail).

Les contraintes OCL associées aux opérations (pré-condition et post-condition) seront traduites par l'instruction FoCAL : **if ... then ... else ...**, lors de la définition des opérations des espèces. A titre d'exemple, la contrainte :

```
Context Light::lightOrange():void
 Pre:
   self.status = Green
 Post:
   self.status = orange
```

sera traduite lors de la définition de l'opération **lightOrange** au niveau de la collection Light_collection comme suit :

```
let lightOrange(x)=if(!egale(x,!vert))then(!orange)else(!foc_error(""));
```

puis, la propriété :

```
property lightOrange_prop : all x in self,
(!egale(x,!vert)) -> (!egale((!lightOrange(x)) , !orange));
```

sera rajoutée à l'espèce Light pour être utilisée comme un axiome lors de la preuve des théorèmes (voir annexe).

5.5 L'aspect preuve

Nous illustrons l'aspect preuve à travers deux théorèmes que nous prouverons et une contradiction.

Exemple 1

Nous présentons d'abord la preuve d'un théorème déduit du diagramme d'états-transition de la classe Light. D'après le diagramme d'état-transition (figure5-2c), il faut prouver que pour toute instance *x* de la collection Light_collection :

$(x = green) \leftrightarrow (Light.lightOrange(x) = orange) \leftrightarrow (Light.light\,\mathrm{Re}\,d(Light.lightOrange(x)) = red)$

Pour cela le théorème suivant doit être introduit dans l'espèce Light :

```
theorem light_transition : all x in self,
!egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
!egale(!lightRed (!lightOrange(x)),!rouge)->
!egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
!egale(x,!lightGreen(!lightRed (!lightOrange(x))))
```

Les indications de preuve de ce théorème par Zenon sont les suivantes :

```
proof :

  <1>1 assume x in self
```

```
    H1 : !egale(x,!vert)

    prove !egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
          !egale(!lightRed (!lightOrange(x)),!rouge)->
          !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
          !egale(x,!lightGreen(!lightRed (!lightOrange(x))))

  <2>1 prove !egale(x, !vert) -> !egale( !lightOrange(x),!orange)
       by <1>:H1 , !lightOrange_prop


  <2>2 prove !egale( !lightOrange(x),!orange)->
             !egale(!lightRed (!lightOrange(x)),!rouge)
       by <2>1  ,!lightRed_prop

  <2>3 prove !egale(!lightRed (!lightOrange(x)),!rouge)->
             !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert)
       by <2>2 , !lightGreen_prop

  <2>4 prove !egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
             !egale(!lightRed (!lightOrange(x)),!rouge)->
             !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert)
       by <2>1,<2>2,<2>3

  <2>5 prove  !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
              !egale(!vert,!lightGreen(!lightRed (!lightOrange(x))))
       by !egale_symetrique


  <2>6 prove !egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
             !egale(!lightRed (!lightOrange(x)),!rouge)->
             !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
             !egale(x,!lightGreen(!lightRed (!lightOrange(x))))
       by <2>4, <2>5 , !egale_transitive

  <2>7 qed

  <1>2 qed;

end
```

Après avoir lancé le processus de preuve Zenon, si les indications de preuve sont correctes, Zenon n'indiquera aucun échec.

Exemple 2

Un autre théorème, qui nous semble important, consiste à s'assurer de l'arrêt de circulation lorsque l'opération arreter_circulation est lancée. Par conséquent, nous ajouterons à l'espèce Control_Light_Barrier le théorème suivant :

```
theorem arret_sure: all x in self,
   !egale ( x , !crp(l!vert,b!ouverte)) ->
  !egale ( !arreter_circulation(x),!crp(l!rouge,b!ferme))
```

L'objectif de ce théorème, est de prouver que l'application de l'opération **arreter_circulation** à partir de l'état circulation_permise (Light = vert et Barrier = ouverte), amène à l'état circulation_arrêtée (Light = rouge et Barrier = ferme) ce qui assure l'arrêt sûre de la circulation (la preuve complète de ce théorème est donnée en l'annexe).

Exemple 3

A présent, examinons le cas où il y a une contradiction par rapport au diagramme d'état-transition lors de la modélisation UML des contraintes OCL suivantes :

```
Context Light::lightRed():void
 Pre:
   self.status = Green
 Post:
   self.status = red
```

Dans cette contrainte, la pré-condition de l'opération lightRed **(self.status = Green)** est en contradiction avec le diagramme d'état-transition de la classe Light.

La transcription FoCAL de la contrainte OCL précédente, donne la propriété suivante :

```
property lightRed_prop : all x in self,
(!egale(x,!vert)) ->  !egale((!lightRed(x)) , !rouge);
```

Cette propriété sera utilisée pour prouver le théorème **light_transition** donné précédemment. Dans ce cas, le processus de preuve Zenon provoque un échec et indique la ligne de code Zenon responsable de l'erreur.

Dans ce chapitre, notre proposition de transcription a été appliquée sur la modélisation UML/OCL d'un système de contrôle de trains. La cohérence de la spécification FoCAL obtenue est assurée par la preuve des théorèmes déduits de cette modélisation. Si les indications de preuves sont introduites correctement, Zenon arrive à prouver tous ces théorèmes.

Dans le cas d'une contradiction dans la modélisation UML ou d'une mauvaise indication de preuves, Zenon indiquera un échec.

Conclusion

Dans cette thèse, nous avons présenté une démarche de transcription d'une modélisation UML vers le système FoCAL. Le modèle UML est exprimé par des diagrammes de classes, des diagrammes d'état-transition et des expressions OCL de type invariant et pré et post-condition.
La restriction à ces diagrammes et ces contraintes n'est pas pénalisante du moment où ils restent assez représentatifs des aspects statiques et dynamiques de la plupart des systèmes. Cependant, notre démarche n'exclut pas la prise en compte d'autres types de diagrammes tels que les diagrammes de collaboration, et de contraintes OCL telles que les contraintes conditionnelles.
La dérivation de diagrammes de classes vers FoCAL permet de construire une hiérarchie d'espèces avec les types supports et les signatures des opérations. La dérivation des expressions OCL et des diagrammes d'état-transition en FoCAL fournit le contenu des opérations et établit l'interaction entre les opérations des espèces dans la spécification FoCAL.

Cette transcription répond à notre objectif, qui est de permettre de valider la consistance entre les diagrammes d'état-transition et les contraintes OCL par la preuve des théorèmes déduits de la modélisation elle-même. En effet, la spécification FoCAL obtenue et les obligations de preuves générées par le compilateur FoCAL nous permettent de valider et de vérifier les propriétés dans la modélisation UML en utilisant Zenon.
Le niveau élevé d'abstraction du système FoCAL, permet de minimiser l'écart entre le formalisme UML et celui du langage cible. En effet, notre transcription préserve les propriétés comportementales du modèle UML, à savoir la création dynamique des instances et la communication entre les diagrammes d'états transition.

Afin d'illustrer notre propos, nous avons présenté l'exemple de la transcription de la modélisation UML du système de contrôle de trains vers FoCAL. Dans cet exemple nous

avons appliqué la démarche proposée et montré comment une contradiction dans le modèle UML peut être détectée par Zenon lors de la preuve.

Nous avons concentré notre attention sur l'aspect preuve, mais nous avons vite ressenti le besoin de disposer d'un outil d'aide réalisant le passage d'UML vers FoCAL,

- soit en utilisant le langage Ocaml pour traduire la modélisation UML.
- soit en utilisant un open source UML (inspiré de ArgoUML+B [LED03]) pour dériver systématiquement la hiérarchie des espèces en utilisant le langage XSLT muni d'une feuille de style contenant les règles de traduction FoCAL/UML.

Finalement il faut noter que les invariants ne sont pas traités par le compilateur FoCAL. La transcription des invariants de classes vers FoCAL n'a pas été prise en charge dans ce travail et nécessiterait d'être complétée.

Annexe

Spécifications FoCAL obtenue par transcription de la modélisation UML du système de contrôle de trains

```
uses basics;;
open basics;;

                    (*  structures de données *)

type barrier_status =
   Opened in barrier_status;
   Closed in barrier_status;
   ;;

type light_status =
   Green in light_status;
   Red in light_status;
   Orange in light_status;
;;

                        (*  l'espèce Barrier *)

species barrier  =
rep = barrier_status;

let foc_error(x in string) in self= #foc_error(x);

sig egale in self ->self ->bool;

sig close_barrier in self -> self;
sig open_barrier in self -> self;

let ouverte in self = #Opened;
let ferme in self = #Closed;

sig new_barrier in barrier_status -> self ;

property open_to_close_prop : all x in self,
(!egale(x,!ouverte)) ->  (!egale((!close_barrier(x)) , !ferme));

property close_to_open_prop : all x in self,
(!egale(x,!ferme)) ->  (!egale((!open_barrier(x)) , !ouverte));

property egale_transitive : all x y z in self,
!egale(x,y) -> !egale(y,z) -> !egale(x,z);

theorem barrier_transition : all x in self,
!egale(x,!ferme) -> !egale((!open_barrier(x)) , !ouverte) ->
```

```
!egale((!close_barrier((!open_barrier(x)))) , !ferme)
proof:

   <1>1 assume x in self

      H1:!egale(x,!ferme)

      prove !egale(x,!ferme) -> !egale((!open_barrier(x)) , !ouverte) ->
              !egale((!close_barrier((!open_barrier(x)))) , !ferme)

   <2>1 prove !egale(x,!ferme) -> !egale((!open_barrier(x)) , !ouverte)
          by <1>:H1 , !close_to_open_prop

   <2>2 prove !egale((!open_barrier(x)) , !ouverte) ->
              !egale((!close_barrier((!open_barrier(x)))) , !ferme)
          by <2>1 , !open_to_close_prop

   <2>3 prove !egale(x,!ferme) -> !egale((!open_barrier(x)) , !ouverte) ->
        !egale((!close_barrier((!open_barrier(x)))) , !ferme)
      by <2>1 , <2>2

   <2>4 qed

   <1>2 qed;


end



collection  barrier_collection implements barrier =

let new_barrier (x in barrier_status) in self = x;

let egale = #base_eq;

let close_barrier(x) = if(!egale(x,#Opened)) then ( #Closed) else
(!foc_error(""));
let open_barrier(x) = if(!egale(x,#Closed)) then ( #Opened) else
(!foc_error(""));

proof of egale_transitive = assumed;
proof of open_to_close_prop = assumed;
proof of close_to_open_prop = assumed;

end

                        (*  L'espèce Light *)

species light =

rep = light_status ;

sig egale in self ->self ->bool;

let vert in self = #Green ;
let orange in self = #Orange ;
let rouge in self = #Red;

sig lightOrange in self -> self;
```

```
sig lightRed in  self -> self;

sig lightGreen in self -> self;

let foc_error(x in string) in self= #foc_error(x);



let new_light(x in light_status) in self = x;

property egale_reflexive : all x in self,
(!egale(x,x)) ;

property egale_symetrique : all x y in self,
!egale(x,y) -> !egale(y,x);

property egale_transitive : all x y z in self,
!egale(x,y) -> !egale(y,z) -> !egale(x,z);

property lightRed_prop : all x in self,
(!egale(x,!orange)) ->  !egale((!lightRed(x)) , !rouge)
;

property lightOrange_prop : all x in self,
(!egale(x,!vert)) ->  (!egale((!lightOrange(x)) , !orange))
;


property lightGreen_prop : all x in self,
(!egale(x,!rouge)) ->  (!egale((!lightGreen(x)) , !vert))
;

theorem light_transition : all x in self,
!egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
!egale(!lightRed (!lightOrange(x)),!rouge)->
!egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
!egale(x,!lightGreen(!lightRed (!lightOrange(x))))

proof :

    <1>1 assume x in self
         H1 : !egale(x,!vert)
         prove !egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
                 !egale(!lightRed (!lightOrange(x)),!rouge)->
                 !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
                 !egale(x,!lightGreen(!lightRed (!lightOrange(x))))

    <2>1 prove !egale(x, !vert) -> !egale( !lightOrange(x),!orange)
           by <1>:H1 , !lightOrange_prop

    <2>2 prove !egale( !lightOrange(x),!orange)->
                    !egale(!lightRed (!lightOrange(x)),!rouge)
             by <2>1  ,!lightRed_prop

    <2>3 prove !egale(!lightRed (!lightOrange(x)),!rouge)->
                   !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert)
             by <2>2 , !lightGreen_prop

    <2>4 prove !egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
                   !egale(!lightRed (!lightOrange(x)),!rouge)->
                   !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert)
```

```
            by <2>1,<2>2,<2>3

    <2>5 prove  !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
                !egale(!vert,!lightGreen(!lightRed (!lightOrange(x))))
          by !egale_symetrique



    <2>6 prove !egale( x, !vert) -> !egale( !lightOrange(x),!orange)->
               !egale(!lightRed (!lightOrange(x)),!rouge)->
               !egale(!lightGreen(!lightRed (!lightOrange(x))),!vert) ->
               !egale(x,!lightGreen(!lightRed (!lightOrange(x))))
            by <2>4, <2>5 , !egale_transitive

    <2>7 qed

    <1>2 qed;

end


collection  light_collection implements light =

let egale = #base_eq;
let lightOrange(x) = if(!egale(x,!vert)) then ( !orange) else
(!foc_error(""));
let lightRed(x) = if(!egale(x,!orange)) then ( !rouge) else
(!foc_error(""));
let lightGreen(x) = if(!egale(x,!rouge)) then ( !vert) else
(!foc_error(""));

proof of  egale_reflexive = assumed;
proof of  egale_symetrique = assumed;
proof of  egale_transitive = assumed;

proof of lightOrange_prop = assumed;
proof of lightRed_prop = assumed;
proof of lightGreen_prop = assumed;

end
```

(* l'espèce Control *)

```
species control =


rep;

sig egale in self -> self -> bool;
sig foc_error in string -> self;
sig remettre_circulation in self -> self;
sig arreter_circulation in self -> self;

end
```

(* l'espece control_light_barrier *)

```
species control_light_barrier(l is light, b is barrier)inherits control =
```

```
rep = l*b;
let first ( x in self) in l = #first(x) ;
let scnd ( x in self) in b = #scnd(x) ;
sig egale in self -> self -> bool;



property egale_3: all x in self,(
                  all li in l,(
                  all ba in b ,(

!egale(x, !crp(li,ba)) <-> #and_b( l!egale(!first(x),li)
                                  , b!egale(!scnd(x),ba)) ) ) );
property egale_pair:
             all l1 l2 in l,(
             all b1 b2 in b,(
             ((l!egale(l1,l1)) -> (b!egale(b1,b2)) )  ->
           !egale(!crp(l1,b1),!crp(l2,b2) ) ));

property and_prop1 : all a b in bool,
#and_b(a,b) -> a
;

property and_prop2 : all a b in bool,
#and_b(a,b) -> b;

let  foc_error(x in string) in self = #foc_error(x);


sig arreter_circulation in self -> self;

sig remettre_circulation in self -> self;

let new_control(x in l, y in b) in self = #crp(x,y);

let crp(x in l, y in b) in self = #crp(x,y);

let vert_ouverte in self  = #crp(l!vert,b!ouverte);

let rouge_ferme in self  = #crp(l!rouge,b!ferme);


property arreter_prop : all x in self ,
!egale(x,!crp(l!vert,b!ouverte)) -> !egale( !arreter_circulation(x) ,
                !crp( (l!lightRed(l!lightOrange(!first(x)))) ,
                                  (b!close_barrier(!scnd(x))) )) ;

property egale_transitive: all x in self,(
          all l1 l2 in l,(
          all b1 b2 in b,(
          (!egale(x,!crp(l1,b1)) -> !egale(!crp(l1,b1),!crp(l2,b2)) )->
                  !egale(x,!crp(l2,b2) ) ))) ;

theorem arret_sure: all x in self,
   !egale ( x , !crp(l!vert,b!ouverte)) ->
          !egale ( !arreter_circulation(x),!crp(l!rouge,b!ferme))
proof:
    <1>1 assume x in self
        H1 : !egale ( x , !crp(l!vert,b!ouverte))
```

```
  prove !egale ( x , !crp(l!vert,b!ouverte)) ->
         !egale ( !arreter_circulation(x),!crp(l!rouge,b!ferme))

<2>1 prove !egale(x,!crp(l!vert,b!ouverte)) ->
               !egale( !arreter_circulation(x) ,
                      !crp((l!lightRed(l!lightOrange(!first(x)))) ,
                             (b!close_barrier(!scnd(x))) )  )
     by <1>:H1 , !arreter_prop

<2>2 prove !egale( !arreter_circulation(x) ,
                    !crp( (l!lightRed(l!lightOrange(!first(x)))) ,
                        (b!close_barrier(!scnd(x))) )  ) ->
     #and_b( l!egale( !first(!arreter_circulation(x)),
                   l!lightRed(l!lightOrange(!first(x)))) ,
                      b!egale( ((!scnd(!arreter_circulation(x)))),
                                  (b!close_barrier(!scnd(x)))) )
     by <2>1 , !egale_3


<2>3 prove  #and_b( l!egale( !first(!arreter_circulation(x)),
                    l!lightRed(l!lightOrange(!first(x)))) ,
                      b!egale( ((!scnd(!arreter_circulation(x)))),
                        (b!close_barrier(!scnd(x)))) )->
                          l!egale( !first(!arreter_circulation(x)),
                            l!lightRed(l!lightOrange(!first(x))))
     by <2>2, !and_prop1


<2>4 prove  l!egale(!first(x),l!vert) ->
          l!egale(l!lightOrange(!first(x)) , l!orange)
     by <1>:H1 ,  l!lightOrange_prop

<2>5 prove  l!egale(l!lightOrange(!first(x)) , l!orange) ->
           l!egale(l!lightRed(l!lightOrange(!first(x))) , l!rouge)
       by <2>4 , l!lightRed_prop

<2>6 prove  l!egale( !first(!arreter_circulation(x)),
                         l!lightRed(l!lightOrange(!first(x)))) ->
         l!egale(l!lightRed(l!lightOrange(!first(x))) , l!rouge) ->
                   l!egale( !first(!arreter_circulation(x)),l!rouge )
   by <2>3 , <2>5,l!egale_transitive

<2>7 prove #and_b( l!egale( !first(!arreter_circulation(x)),
                         l!lightRed(l!lightOrange(!first(x)))) ,
                  b!egale( ((!scnd(!arreter_circulation(x)))),
                                (b!close_barrier(!scnd(x)))) )->
           b!egale( ((!scnd(!arreter_circulation(x)))),
                                   (b!close_barrier(!scnd(x))))
     by <2>2 , !and_prop2

<2>8 prove  b!egale(!scnd(x),b!ouverte) ->
          b!egale(b!close_barrier(!scnd(x)) , b!ferme)
     by <1>:H1 ,  b!open_to_close_prop

<2>9 prove  ((l!egale(l!lightRed(l!lightOrange(!first(x))) , l!rouge))
          and  (b!egale(b!close_barrier(!scnd(x)) , b!ferme)) )->
            !egale( !crp( (l!lightRed(l!lightOrange(!first(x)))),
           (b!close_barrier(!scnd(x))) ), !crp( l!rouge, b!ferme ) )
              by <2>5 , <2>8, !egale_pair

 <2>10 prove  (!egale( !arreter_circulation(x) ,
```

```
                    !crp( (l!lightRed(l!lightOrange(!first(x)))) ,
                (b!close_barrier(!scnd(x))) ) ))->
          (!egale( (!crp( (l!lightRed(l!lightOrange(!first(x)))),
       (b!close_barrier(!scnd(x))) )), !crp( l!rouge, b!ferme ) ))->
         !egale( !arreter_circulation(x) , !crp( l!rouge, b!ferme ) )
      by <2>1 , <2>9,!egale_transitive


  <2>11 prove !egale ( x , !crp(l!vert,b!ouverte)) ->
           !egale ( !arreter_circulation(x),!crp(l!rouge,b!ferme))
       by <2>1,<2>10

  <2>12 qed
  <1>2 qed ;

end

collection  control_light_barrier_collection  implements
control_light_barrier(light_collection,barrier_collection) =

let egale (x in self,y in self) in bool =
#and_b(light_collection!egale(!first(x),!first(y)),barrier_collection!egal
e(!scnd(x), !scnd(y)) );


let arreter_circulation(x) =
if !egale(x,#crp(light_collection!vert,barrier_collection!ouverte) ) then
 #crp((light_collection!lightRed(light_collection!lightOrange(!first(x)),
             (barrier_collection!close_barrier(!scnd(x))) )
                   else (!foc_error(" ")) ;

let remettre_circulation(x) =
if !egale(x,#crp(light_collection!rouge,barrier_collection!ferme) )then
          #crp( (light_collection!lightGreen(!first(x))),
                   (barrier_collection!open_barrier(!scnd(x))) )
                  else (!foc_error(" "));

proof of and_prop1 = assumed;
proof of and_prop2 = assumed;
proof of egale_3 = assumed;
proof of arreter_prop = assumed;
proof of egale_pair = assumed;
proof of egale_transitive = assumed;
end
```

Bibliographie

[ABR96] J.R. Abrial. The B-Book. Cambridge University Press, 1996.

[BOO03] G. Booch and J. Rumbaught and I. Jacobson. Le guide de l'utilisateur UML. Edition Eyrolles 2003.

[BOO91] G. Booch. Object Oriented Design with Applications. Benjamin Cummings, 1991.

[BOU00] S. Boulmé. Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel. PhD thesis, Université Paris 6, 2000.

[BOU01] S. Boulmé, T. Hardin and R. Rioboo. Some hints for polynomials in the Foc project. In Calculemus 2001 Proceedings, June 2001.

[CAR00] P. J. F. Carreira and M. E. F. Costa. Automatically verifying an object-oriented specification of the steam-boiler system. Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000), pages 345–360, 2000.

[CCI87] CCITT. SDL, Recommendation Z.100, 1987.

[CHA05] A. Charguéraud. Programmation en Caml pour Débutants, http://www.france-ioi.org/cours_caml/navig.php.2005

[CLA01] Tony Clark, Andy Evans and Stuart Kent. The metamodelling language calculus: Foundation semantics for UML. 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, Springer, 2001.

[CLE03] CLEARSY. Manuel de référence du langage B, version 1.8.5, 2003.

[COQ97] The COQ development team. The Coq Proof Assistant Reference Manual v. 6.1, rapport INRIA N° 0203, 1997.

[DOL04] D. Doligez. Zenon, first-order prover with coq-checkable output. 2nd Workshop on Coq and Rewriting, September 2004.

[DUB01] C. Dubois, notes de cours DEA informatique. Les programmes vus comme des preuves, les types vus comme des propositions, Octobre 2001.

[ETI05] J. F. Etienne, Modélisation de la réglementation de l'aviation civile en FoCAL, 2005.

[FEC05] S. Fechter. Sémantique des traits orientés objet de FoCAL, Thèse de doctorat, Université Paris 6, 2005.

[FLE99] W. Fleisch. Applying use cases for the requirements validation of component-based real-time software. Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99), May 1999.

[FOC03] FoC development team, A brief FoC tutorial, Version 0.0, July 01 2003.

[FOC06] FoC development team. FoCAL Reference Manual, 2006.

[FRÖ00] P. Fröhlich and J. Link. Automated test case generation from dynamic models. In E. Bertino, editor, Proceedings of ECOOP 2000, volume 1850 of LNCS, pages 472–491. Springer, 2000.

[GUN01] A. L. Guennec. Génie Logiciel et Méthodes Formelles avec UML Spécification, Validation et Génération de tests, thèse doctorat de l'université de Rennes I, juin 2001.

[HOL97] J. G. Holzmann. The model checker SPIN. IEEE Transactions on Software Engineering, 23(5):279–295, May 1997.

[ISO85] ISO. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO/ DP 8807, March 1985.

[JAC92] I. Jacobson, M. Christerson, P. Jonsson, and Gunnar Overgaard.Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley/ACM Press, 1992.

[JAU05] M. Jaume and C. Morisset. Formalisation and implementation of access control models. In Information Assurance and Security (IAS'05) International Conference on Information Technology, ITCC, pages 703–708. IEEE CS Press, 2005.

[JAU06] Mathieu Jaume, Présentation de FoCAL, COQ (V8) et Zenon, LIP6 – SPI Université Paris 6: Juin 2006.

[KWO00] G. Kwon. Rewrite rules and operational semantics for model checking UML statecharts. Third International Conference, York, UK, October 2000.

[LAL02] R. Laleau and F. Polack. Coming and going from UML to B, CEDRIC-IIE 2002.

[LED02] H. Ledang.Traduction systématique de spécifications UML en B – Thèse de PhD, université Nancy 2, Novembre 2002.

[LED03] H. Ledang, J. Souquières, and S. Charles. ArgoUML+B : un outil de transformation systématique de spécification UML en B. Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2003.

[MAC84] D. B. MacQueen. Modules for standard ml. Lisp and Functional Programming, 1984

[MAR97] G. Mariano. Évaluation de logiciels critiques développés par la méthode B, Une approche quantitative. Thèse de PhD, Université de Valenciennes et du Hainaut Cambrésis, décembre 1997.

[MEY01] E. Meyer. Développements formels par objets : utilisation conjointe de B et UML – Thèse de PhD, université Nancy 2, mars 2001.

[MOR06] C. Morisset, Tutorial sur FoCAL. Séminaire LACL LIP6, 12 Juin 2006.

[MUL00] P.A. Muller & N. Gaetner. Modélisation Objet avec UML. Edition EYROLLES, 2000.

[OFF99] J. Offutt and A. Abdurazik. Generating tests from UML specifications.. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings, volume 1723 of LNCS, pages 416–429. Springer, 1999.

[OMG03] Updated joint initial submission against the action semantics for UML RFP, available at http://cgi.omg.org/cgi-bin/doc?ad/00-08-03.

[PAL99] I. Paltor and J. Lilius. A tool for verifying UML models. Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99. IEEE, 1999.

[PAL99b] Mario Paludetto and Jérome Delatour. UML et les réseaux de petri : vers une sémantique des modèles dynamiques et une méthodologie de développement des systèmes temps réel. L'objet, 5:443–467, 1999.

[PRE03] Virgile Prevosto. Conception et implantation du langage FoC pour le développement de logiciels certifiés. PhD thesis, Université Paris VI, 2003.

[REM02] D. Rémy. Initiation au langage OCaml, http://pauillac.inria.fr/remy/ocaml/ 2002.

[RUM91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. Object-Oriented Modeling and Design. Prentice Hall, New Jersey, 1991.

[SCH02] P. H. Schmitt: UML and its Meaning, 2003.

[THU04] N. Thuan, T. Souquières, and J. Validation des propriétés de construction d'un scénario UML/OCL à partir de sa dérivation B, 2004.

[WAR03] J. Warmer and A. Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Second Edition 2003.