# Transformation de modèles UML vers des programmes Fiacre

Sotharith Heng

Master 2 Recherche en Informatique

# Rapport

# Transformation de modèles UML vers des programmes Fiacre

HENG Sotharith

heng.sotharith@gmail.com

Encadrants : Philippe Dhaussy, Bruno Aizier

Laboratoire Lab-STICC / ENSTA Bretagne, 2 rue François Verny,

BREST (France), 29806, Cedex 9.

philippe.dhaussy@ensta-bretagne.fr




# Table des matières

1 Introduction ..... 4
1.1 Contexte ..... 4
1.1.1 Présentation et problématiques ..... 4
1.1.2 Etude de l'existant ..... 4
1.2 Objectif du travail ..... 5
1.3 Rhapsody ..... 6
1.4 UML ..... 7
1.4.1 Quels diagrammes choisir ? ..... 7
1.5 Langage FIACRE ..... 7
1.6 Principes de traduction du modèle UML en Fiacre ..... 10
2 Règles de transformation ..... 12
2.1 Traduction de structure du système ..... 12
2.1.1 Traduction de classes inactives ..... 12
2.1.2 Traduction de classes actives ..... 12
2.1.3 Attributs ..... 13
2.1.3.1 Types primitifs ..... 13
2.1.3.2 Types non primitifs ..... 14
2.1.3.3 Accès aux attributs d'une la classe ..... 14
2.1.4 Association entre les classes ..... 17
2.1.4.1 Association simple ..... 17
2.1.4.2 Multiplicité ..... 17
2.1.5 Gestion des concepts d'objet ..... 17
2.1.5.1 Généralisation et Héritage ..... 17
2.1.5.2 Variables temporaires ..... 18
2.1.6 Ports ..... 18
2.1.6.1 Ports Standard ..... 18
2.1.6.2 Flow Port ..... 19
2.2 Traduction de la dynamique du système ..... 20
2.2.1 Traduction de la transition ..... 20
2.2.2 Envoi et réception des évènements ..... 20

2.2.3 Appel et réception des opérations ..... 22
2.2.3.1 Triggered opération ..... 22
2.2.3.2 Opération primitive ..... 26
2.2.4 Traduction de l'action ..... 30
2.2.4.1 Actions dans la transition ..... 31
2.2.4.2 Action en entrée dans un état ..... 32
2.2.4.3 Action en sortie d'un état ..... 32
2.2.5 Traduction de l'état composite ..... 33
2.2.5.1 Etat composite non-orthogonal (sous-état) ..... 33
2.2.5.2 Etat composite orthogonal (AND State) ..... 34
2.2.6 Traduction de sous-machine d'état ..... 35
2.3 Traduction du diagramme d'objets ..... 35
3 Validation des règles de transformation ..... 37
3.1 Validation des règles de transformation ..... 37
3.2 Propriétés à vérifier ..... 38
3.3 CDL ..... 38
3.4 Expression des propriétés ..... 38
4 Conclusion ..... 42
4.1 Travail réalisé ..... 42
4.2 Difficultés ..... 42
4.3 Perspectives ..... 42
Annexes ..... 46

# Table des figures

Figure 1 : objectif du travail....6
Figure 2 : langage Pivot Fiacre....8
Figure 3 : déclaration Fiacre du composant « Elevator »....9
Figure 4 : déclaration Fiacre du processus « LiftPanel »....9
Figure 5 : le schéma de traduction....10
Figure 6 : Outillage utilisé pour les transformations de modèles....11
Figure 7 : traduction de la classe inactive....12
Figure 8 : traduction de la classe active....13
Figure 9 : Automate d'un processus gérant l'accès concurrent à une variable *data* partagée....15
Figure 10 : Exemple d'un processus qui gère l'accès concurrent aux attributs....16
Figure 11 : la traduction du *flow port*....20
Figure 12: scénario général d'envoi d'un évènement entre deux processus Fiacre....21
Figure 13: l'envoi et la réception d'un évènement entre deux objets....22
Figure 14 : scénario d'appel d'une *triggered* opération UML....23
Figure 15 : T*riggered* opération : communication entre le processus appelant et le processus appelé....23
Figure 16 : la réception de l'appel de *triggered* opération « MotOn »....24
Figure 17 : *triggered* opération et garde....25
Figure 18 : la réception d'une *triggered* opération, afin d'éviter un *deadlock*....26
Figure 19 : l'opération primitive qui est traduite par un bloc du code....27
Figure 20 : traduction d'une opération primitive par un bloque du code....28
Figure 21 : l'opération primitive qui est traduite par un processus....28
Figure 22 : le processus qui effectue l'opération primitive....29
Figure 23 : la traduction de l'opération primitive en un processus....30
Figure 24 : l'action dans la transition....32
Figure 25 : les actions entrées dans un état....32
Figure 26 : les actions sortant d'un état....33
Figure 27 : l'état composite non-orthogonal avec l'entrée par défaut....34
Figure 28 : la sortie de l'état composite non-orthogonal....34
Figure 29 : la traduction de sous-machine d'état....35
Figure 30 : validation des règles de transformation....37
Figure 31 : cdl graphe de propriété Pty1....40
Figure 32 : cdl graphe de propriété Pty2.1....41
Figure 33 : cdl graphe de propriété Pty2.2....41
Figure 34 : graphe d'exécution....41

**Liste des tableaux**

Tableau 1 : traduction de types prédéfinis dans UML en Fiacre....14
Tableau 2 : propriétés nécessaires à vérifier....39

# Chapitre 1
# 1 Introduction

## 1.1 Contexte

### 1.1.1 Présentation et problématiques

Aujourd'hui, le domaine de la modélisation et de la validation formelle de logiciels est un enjeu majeur du génie logiciel. De nombreux travaux universitaires ont exploré différents formalismes de modélisation, langages et outils pour concevoir des composants logiciels.

Plus spécifiquement, les architectures logicielles des systèmes embarqués doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. Malgré les progrès techniques, la grande taille de ces systèmes facilite l'introduction d'une plus grande gamme d'erreurs. Actuellement, les industries engagent tous leurs efforts dans le processus de tests et de simulations à des fins de certification. Néanmoins, ces techniques deviennent rapidement inexploitables pour mettre en évidence des erreurs. La couverture des jeux de tests s'amincie au fur et à mesure de la complexification des systèmes et il devient nécessaire d'utiliser de nouvelles méthodes pour garantir la fiabilité des logiciels.

Parmi celles-ci, les méthodes formelles ont contribué, depuis plusieurs années, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de *model-checking* [QUE 1982, CLA 1986] ont été fortement popularisées grâce à leur faculté d'exécuter automatiquement des preuves de propriétés sur des modèles logiciels. De nombreux outils *model-checkers* ont été développés dans ce but [HOL 1997, LAR 1997, [BER 2004], FER 1996, [CIM 2000] ].

Pour accroître la pénétration des formalismes, comme par exemple de type UML, dans les processus industriels d'ingénierie système et logicielle, il est encore nécessaire de pouvoir proposer aux utilisateurs des techniques opérationnels d'analyses de ses modèles. Ceci implique de disposer de transformation de modèles adéquats, paramétrables ou configurables permettant de générer, à partir des modèles de conception UML[1] des codes formels pour différents outils selon le type d'analyse que l'on veut mener. Par exemple, pour la vérification d'exigences fonctionnelles, il est nécessaire de traduire les modèles dans des codes exploitables par un *model-checker*. Mais il est bien connu que la sémantique des modèles peut varier selon l'interprétation donnée par l'utilisateur. Il est donc pertinent de concevoir des règles de transformation qui soient associées à des choix sémantiques complètement définies.

### 1.1.2 Etude de l'existant

De nombreux de travaux de recherches sont menés dans le domaine des techniques de validation formelle des logiciels temps réel. Cependant, le sujet reste délicat. D'un côté, ces techniques impliquent de décrire le logiciel avec un formalisme de haut niveau (SDL[2], UML,

[1] Unified Modeling Language
[2] Specification and Description Language

SysML[3],...). De l'autre, un grand nombre d'outils sont nécessaires pour couvrir les activités exercées dans le processus de développement : modélisation, simulation, vérification, test. De nombreux aspects techniques et scientifiques recouvrent ce domaine : formalisation des exigences, *model-checking*, test. Un certain nombre de travaux ont inspiré cette étude, à savoir :

1. Le laboratoire Lab-STICC s'intéresse lui sur la transformation de modèle UML vers des langages formels et la validation formelle des logiciels temps réel. De nombreux travaux de recherche ont porté sur la transformation de modèle UML vers le langage intermédiaire IF2 [MAR 2002, MAX 2008, JUL 2007, MAX 2009, HAB 2004] et le développement de l'outil OBP Explorer [OBP 2011].

2. Le projet OMEGA[4] s'est intéressé lui à la validation de modèles UML temporisés. Deux approches ont été abordées
    a. La première [IUL 2003, SUS 2003] est basée sur le langage intermédiaire IF2. Un outil a été développé par VERIMAG et a été basé sur une extension du profile SPT[5]. Celle-ci prend en compte les occurrences d'évènements temporisés lors d'une exécution, dans le but de faciliter l'observation des modèles, et de permettre la vérification des propriétés.
    b. La seconde approche [JOS 2004] utilise une technique basée sur la preuve de théorème (*theorem proving).* Ce qui permet la preuve de raisonner sur des modèles exploitant des données de taille infinie comme par exemple des files d'attentes. Dans notre étude, nous ne ciblons pas ce type de technique mais plutôt les techniques basées sur le *model-checking*.

3. Le projet TOPCASE[6] a donné lieu au développement du langage formel intermédiaire Fiacre [FAR 2008, BER 2007, BER 2008] qui est exploitable par l'outil *modèle-checker* OBP Explorer ou TINA [BER 2004]. Le langage pivot Fiacre est un langage formel de description qui s'inspire des nombreux travaux de recherche, à savoir V-Cotre [BER 2003], NTIF [HUB 2002], ainsi que les décennies de recherche sur la théorie de la concurrence et la théorie du système en temps réel. Dans notre étude, nous utilisons Fiacre comme langage cible de nos transformations de modèles.

## 1.2 Objectif du travail

L'objectif de mon travail est d'établir, à partir de la sémantique d'UML, des règles de transformation vers Fiacre. Afin de formaliser des règles de transformation, il faut assurer que la sémantique du système est préservée et que toutes les propriétés qui sont vérifiées dans le monde UML sont aussi préservées dans le monde Fiacre. Ensuite, le programme cible Fiacre sera utilisé pour vérifier formellement (*model-checking*) des propriétés avec l'outil OBP Explorer [DBR 2011, OBP 2011]. Pour cela, les propriétés du système, encodées par des prédicats et des automates d'observateurs sont décrits avec le langage CDL (*Context Description language*) [CDL 2012].

Dans notre travail, nous ne prenons pas en compte les aspects temporisés des modèles. Ceci fait l'objet d'un autre travail, réalisé est en parallèle avec mon travail, et qui porte sur la « Formalisation des règles de traduction des Méta-modèles du sous profile Time de MARTE

[3] Systems Modeling Language
[4] http ://www-Omega.imag.fr
[5] Schedulability Performance and Time
[6] http ://www.topcased.org

en Fiacre ». L'objectif est d'intégrer ensemble ces deux travaux. Mon travail est donc consacrée uniquement sur la partie non-temporisée de modèles UML.

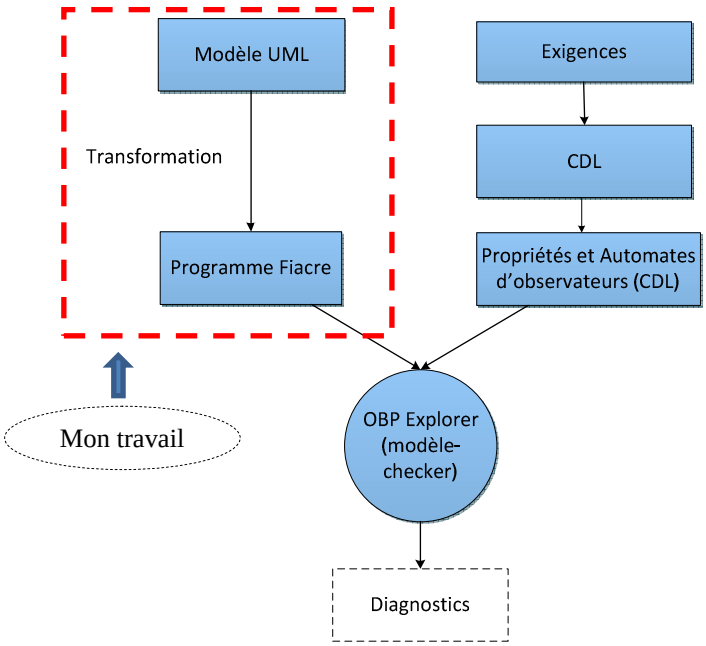



**Figure 1 : objectif du travail**

## 1.3 Rhapsody

*Rhapsody* [RHA 2008, REL 2009, TEL, HAR 2001, HAR 1998] est un outil de modélisation de systèmes temps réel. Il propose des fonctions de génération de tests et de code pour appuyer le processus de développement de systèmes temps réel orientés objets. De ce fait, il utilise une représentation opérationnelle des modèles.

Certains concepts nécessaires à la modélisation de logiciels temps réel ne sont pas pris en compte par *Rhapsody*. En particulier pour la spécification des aspects temporels. En effet, la seule notion temporelle disponible dans cet outil est le *timer*. Or cette notion est insuffisante pour modéliser un système temps réel, la valeur d'un *timer* ne pouvant pas être, par exemple, testée dans une garde. Compte tenu de l'état de développement des techniques de la validation des logiciels distribués, il n'est pas encore possible d'utiliser un langage de modélisation comme langage d'implémentation d'un logiciel industriel conjointement à une technique de validation formelle. Afin de progresser sur cette problématique, il est nécessaire d'étudier des solutions permettant :

1. D'utiliser un outil qui assure un processus de modélisation complet des systèmes et de génération de code exécutable.
2. De valider les modèles de manière complémentaire par des techniques de test pour supporter des logiciels de taille importante.

Dans *Rhapsody*, seuls les diagrammes de classes, les diagrammes d'états et le diagramme d'objets sont utilisés pour générer le code. Dans les diagrammes de classes, les collaborations permettent de déclarer des *pseudos variables*. Une classe voit une autre classe à travers une collaboration par l'intermédiaire de ces variables. Elles peuvent être utilisées pour envoyer des messages entre les instances du système.

## 1.4 UML

Le langage modélisation UML[7] [OMG 2010, LAU 2009] est maintenant largement utilisé dans le processus de développement de logiciels orientés d'objet. Un modèle UML contient à la fois des aspects dynamiques et statiques. En générale, les aspects statiques sont introduits par les diagrammes de classes liés à des contraintes écrites en OCL[8] et les diagrammes d'objets. Les aspects dynamiques sont introduits par les diagrammes d'états ou les diagrammes d'activités.

### 1.4.1 Quels diagrammes choisir ?

Un modèle UML contient beaucoup d'informations qui ne sont pas toutes nécessaires pour la validation formelle du modèle. Par exemple, UML 2.0 possède treize types de diagrammes. Quelques-uns sont trop informels pour être utilisés dans une phase de validation, comme par exemple les *Use Cases*. Pour d'autres diagrammes, leur relation avec l'environnement du modèle UML n'est pas clairement définie (comme par exemple, les diagrammes de de déploiement). Alors, nous nous limiterons ici aux concepts UML qui spécifient une vue opérationnelle du système modélisé, c'est à dire la structure et la dynamique du système, complétée par les données statiques nécessaires (diagrammes de classes incorporant la spécification des structures de données, diagrammes d'objets).

Dans un système, les objets réactifs communiquent entre eux via des envois asynchrones d'évènement et des appels synchrones d'opération. Les aspects dynamiques peuvent être modélisés par des diagrammes dynamiques UML, comme les diagrammes d'états. Lors des transformations vers Fiacre, nous prendrons en compte uniquement trois types de diagrammes d'UML qui sont donc [MAX 2008, JUL 2007, MAX 2009, HAB 2004] :

1. Les diagrammes de classes qui spécifient la structure interne des objets (données, méthodes de traitement) du système.
2. Les diagrammes d'états qui représentent le comportement des objets actifs du système et qui seront traduits par des automates de comportement Fiacre.
3. Le diagramme d'objets qui représente l'architecture du système et qui sera traduit par un réseau de processus à instancier.

## 1.5 Langage FIACRE

Le langage Fiacre (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) [FAR 2008, BER 2007, BER 2008] a été développé dans le cadre du projet TOPCASED[9] comme langage pivot entre les formalismes de haut niveau comme UML, AADL[10], SDL et les outils d'analyse formelle. L'utilisation d'un langage formel intermédiaire apporte l'avantage de réduire le gap sémantique entre les formalismes de haut niveau et les descriptions manipulées en interne par les outils de vérification comme les réseaux de Petri, les algèbres de processus ou les automates temporisés. Fiacre peut être considéré comme un langage disposant d'une sémantique formelle et servant de langage d'entrée à différents outils de vérification (Figure 2).

[7] Unified Modeling Language
[8] Object Constraint Language
[9] www.topcased.org
[10] Architecture Analysis & Design Language

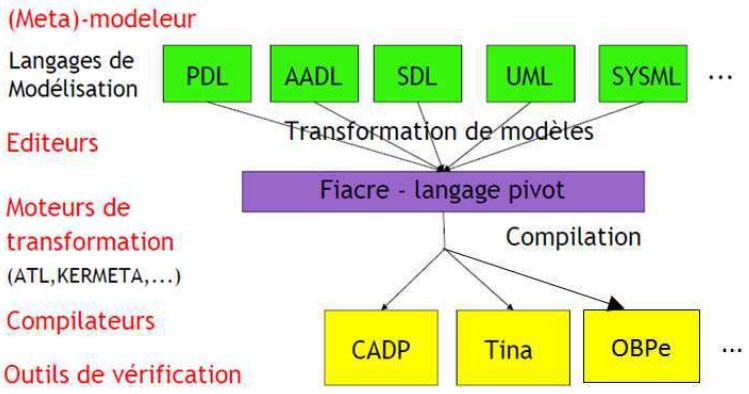



**Figure 2 : langage Pivot Fiacre**

### Structure et entités d'un modèle Fiacre

Fiacre est un langage formel de spécification pour décrire les aspects comportementaux et temporisés des systèmes temps réels. Il intègre les notions de processus et de composants :

- Les automates (processus) sont décrits par un ensemble d'états, une liste de transitions entre ces états avec des constructions classiques (affectations des variables, *if-elsif-else*, *while*, compositions séquentielles), des constructions non déterministes et des communications par ports et par variables partagées.
- Les composants (component) décrivent les compositions de processus. Un système est construit comme une composition parallèle de composants et/ou processus qui peuvent communiquer entre eux par des ports ou variables partagées. Les priorités et les contraintes de temps sont associées aux opérations de communication.

Les processus Fiacre peuvent se synchroniser de manière synchrone avec ou sans passage de valeur via les ports. Ils peuvent également s'échanger des données via des files de communications asynchrones implantées par des variables partagées. Ce sont ces modes de communication que nous utilisons dans le modèle Fiacre. L'expression du temps dans le langage Fiacre est basée sur la sémantique des systèmes de transitions Temporisés (TTS) [HEN 1991]. Toute transition est associée à des contraintes de temps (temps minimum et maximum). Ces contraintes assurent que les transitions sont tirables dans des intervalles de temps définies (ni trop tôt, ni trop tard).

Les instances des processus Fiacre communiquent soit par des variables partagées. Elles permettent alors de modéliser des registres partagés ou des files de messages pour une communication asynchrone en mode *FIFO*[11]. Pour les communications synchrones, les ports sont utilisés.

L'exemple de modèle Fiacre au-dessous comprend un composant « Elevator » (Figure 3) qui regroupe les instances des processus qui s'exécutent en parallèle (opérateur ||) : « *LiftPanel* », « LiftControl » et « HoistControl ». Ces instances communiquent par les variables partagées (LiftPanel_1, events_To_itsLiftControl, events_To_itsHoistControl).

[11] First In, First Out.

```
component Elevator is

var
        LiftPanel_1 : FIFO_Events_LiftPanel,
        events_To_itsLiftControl : FIFO_Events_LiftControl,
        events_To_itsHoistControl : FIFO_Events_HoistControl

init    LiftPanel_1 := {||};
        events_To_itsLiftControl := {||};
        events_To_itsHoistControl := {||}

par
        LiftPanel (&LiftPanel_1, &events_To_itsLiftControl)
        || LiftControl  (&events_To_itsLiftControl, &events_To_itsHoistControl)
        || HoistControl     (&events_To_itsHoistControl)

end

Elevator
```

**Figure 3 : déclaration Fiacre du composant « Elevator »**

Le comportement de chaque processus est modélisé par un automate. Les états et transitions dans les machines d'états UML sont traduits par des états et transitions Fiacre. La Figure 4 illustre partiellement la programmation du processus « LiftPanel ».

```
process LiftPanel (&events_itsLiftPanel : read write FIFO_Events_LiftPanel,
&events_itsLiftControl : FIFO_Events_LiftControl) is
states Off, On, tmp_state_1
var
        event_Received : Type_Events_LiftPanel,
        attributes : Type_LiftPanel,
        arg_LiftPanel_Go : Type_LiftPanel_Arg_Go

init
        event_Received := LiftPanel_NUL;
        to Off

from Off
        if( empty(events_itsLiftPanel)) then loop end;
        event_Received := first (events_itsLiftPanel);
        events_itsLiftPanel := dequeue (events_itsLiftPanel);
        case event_Received of
                LiftPanel_Go arg_LiftPanel_Go->
                        attributes.floor := arg_LiftPanel_Go.floor;
                        to tmp_state_1
        end case;
        loop

from tmp_state_1
        if( full(events_itsLiftControl)) then loop end;
        events_itsLiftControl := enqueue(events_itsLiftControl, LiftControl_Dest({floor =
attributes.floor}));
        to On

from On
        wait[3, 3];
        to Off
```

**Figure 4 : déclaration Fiacre du processus « LiftPanel »**

Pour l'implantation des communications asynchrones, les opérations *first*, *dequeue* et *enqueue* permettent respectivement de lire un message en tête d'une file, de supprimer un message en tête de file et d'écrire un message en queue de file. Chaque processus se voit attribuer une file d'entrée unique.

L'écriture d'un message dans cette file (*enqueue*) correspond à un envoi. Pour cette traduction, le modèle UML est interprété avec la sémantique suivante : l'ordonnancement des processus concurrents (i.e., s'exécutant en parallèle) repose sur l'entrelacement atomique et non déterministe des transitions de chaque processus (atomique signifiant qu'une transition ne peut pas être suspendue au milieu de son exécution).

C'est-à-dire, pour tous les processus possédant des transitions tirables, une transition est susceptible d'être choisie de manière indéterministe et exécutée de manière atomique. Ensuite une autre transition, parmi l'ensemble des transitions tirables, est à nouveau choisie. Cette politique d'ordonnancement est conforme aux hypothèses sémantiques d'exécution d'UML *Rhapsody* [HAR 2001, HAR 1998] et correspond à la sémantique d'exécution des programmes Fiacre.

## 1.6 Principes de traduction du modèle UML en Fiacre

Les principes de traduction du modèle UML en Fiacre sont les suivants. Les classes non actives sont exploitées pour générer des structures de données Fiacre de type *record*. Les classes actives et leur machine d'états sont exploitées pour générer les automates des processus Fiacre. Les objets actifs du modèle UML sont traduits par des instances de processus Fiacre. Le ou les diagrammes d'objets sont exploités pour générer un réseau de processus Fiacre (*component*). Les processus communiquent soit par des ports (pour une communication synchrone), soit par des fifos (pour une communication asynchrone) implantées par des variables partagées (*queues*).

**Schéma de traduction**

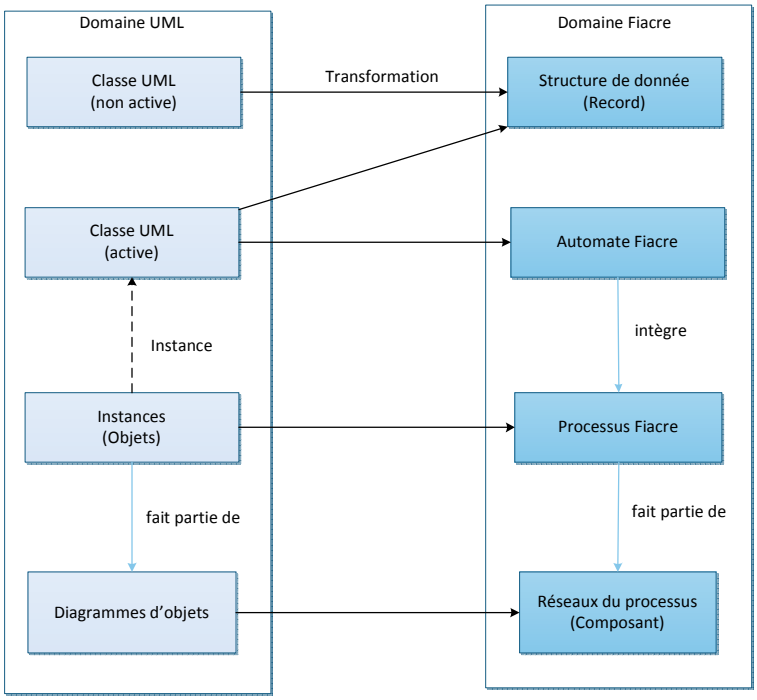



**Figure 5 : le schéma de traduction**

Le schéma au-dessous montre la transformation de modèles UML vers programmes Fiacre avec l'outil *MDWorkbench*[12] *ou Rhapsody Rules Composer* [SOU 2008, RAJ 2010]. Le modèle d'entrée est le modèle UML ou UML *Rhapsody* qui est le format XMI[13] et qui conforme respectivement au méta-modèle MOF[14] ou méta-modèle de *Rhapsody*. Le modèle Fiacre sortie conforme au méta-modèle Fiacre qui est utilisé dans l'outil modèle-checker OBP Explorer [OBP 2011].

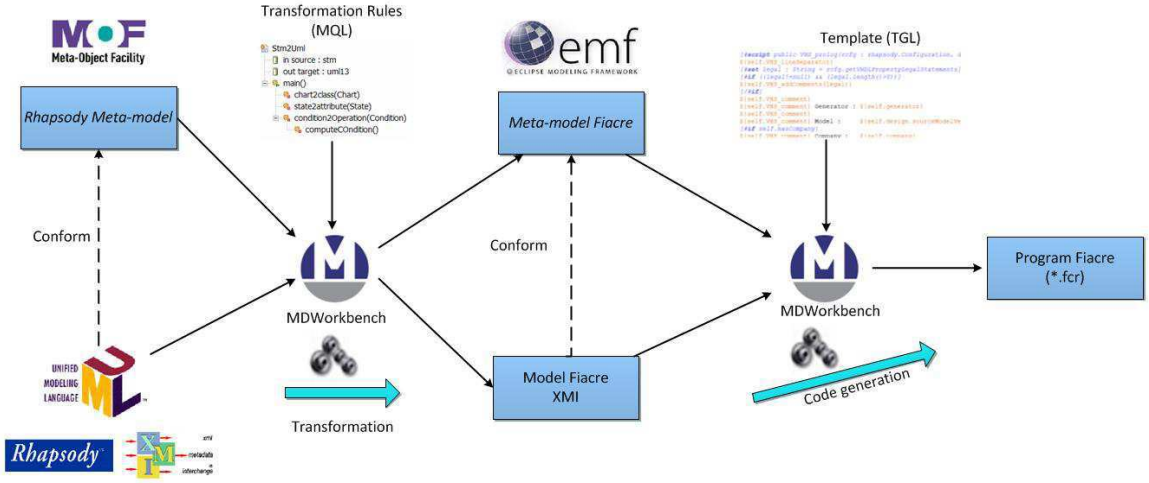



**Figure 6 : Outillage utilisé pour les transformations de modèles**

[12] Un outil qui permet de transformer le modèle UML en un autre modèle, et la génération de code cible.
[13] Le XMI est un format permettant d'échanger des modèles UML d'un outil à un autre.
[14] Meta-Object Facility.

# Chapitre 2

# 2 Règles de transformation

L'objective de ce chapitre est de définir des règles de transformation de modèles UML vers des programmes Fiacre. Les diagrammes à prendre en compte sont les diagrammes de classes, les diagrammes d'états, et le diagramme d'objets. Afin de formaliser les règles de transformation, il faut assurer que les sémantiques du système Fiacre ciblé sont préservées.

## 2.1 Traduction de structure du système

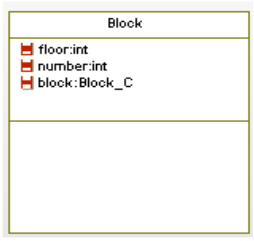
Les diagrammes de classes UML représentent la structure interne du système [OMG 2010, LAU 2009]. Alors, ils seront exploités pour construire la structure du processus Fiacre. Le diagramme de classe UML est composé par les classes, les attributs, les opérations, les différents types d'associations entre les classes et les concepts d'objets. En général, on peut différencier les classes UML en deux catégories, ce sont les classes actives, qui possèdent d'une machine d'états et les classes inactives qui n'en possèdent pas.

### 2.1.1 Traduction de classes inactives

Une classe inactive UML ne possède pas de machine d'états, mais uniquement la spécification de données (variables d'instances). Elle est donc traduite par un type *record* en Fiacre. Chaque attribut de cette classe est traduit par un champ de ce record de type correspondant (voir la partie 2.1.3).

| *Règle 1 :* On traduit la classe inactive par un type de record donc ses champs sont les attributs de la classe. |
|---|

Exemple :




```
        type Type_Block is recordfloor : int,
        number: int,
        block: Block_C
end
```

**Figure 7 : traduction de la classe inactive**

### 2.1.2 Traduction de classes actives

Une classe active UML possède une machine d'états. Il est donc nécessaire de traduire sa machine d'état directement en un automate Fiacre qui sera exécuté par un processus Fiacre. Si la classe active est un singleton (règle 2.a), c'est-à-dire si elle n'est instanciée qu'une seule fois dans le diagramme d'objet, un processus Fiacre est généré. La machine d'états de cette classe est traduite en un automate (règle 2.c) pour construire le comportement du processus (les états, les transitions, etc…). Les attributs de la classe (variables de classe) seront traduits (règle 6.a) par un type de record et encapsulés par un processus Fiacre pour gérer l'accès

concurrent aux variables (voir la partie **Error! Reference source not found.**). Si la classe est instanciée plusieurs fois (règle 2.b), les objets, instances de cette classe, sont traduites par des instances du processus Fiacre correspondant.

*Règle 2.a :*
Pour chaque classe active instanciée une seule fois, un processus en Fiacre est créé, dont le nom est le nom de classe.

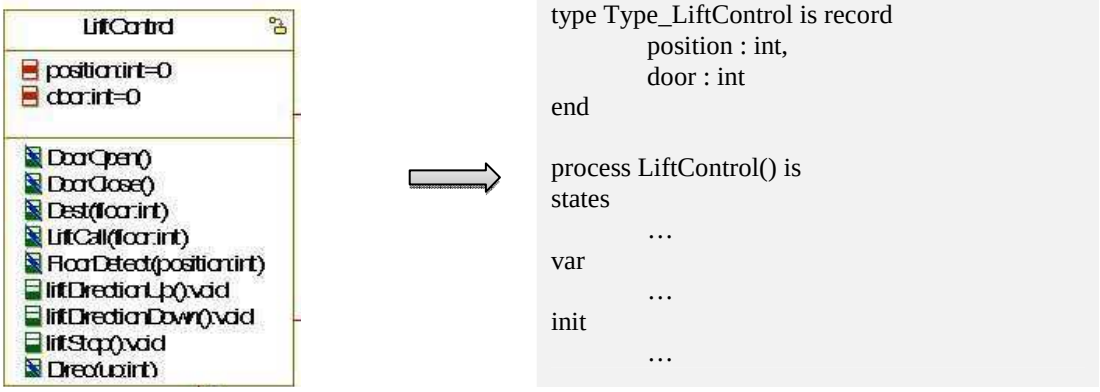



**Figure 8 : traduction de la classe active**

*Règle 2.b :*
Pour chaque classe active instanciée plusieurs fois, une instance du processus Fiacre est créée pour chaque instance, dont le nom est le nom de classe post fixé par le nom de l'instance. (voir Annexe pour la traduction de diagramme d'objets)

*Règle 2.c :*
Chaque machine d'état de la classe d'un objet réactif est traduite par un automate Fiacre.

### 2.1.3 Attributs

Les attributs de classe et d'instances mémorisent des informations que détient un objet, instance de cette classe. . Chacune de ces informations est définie par un nom, un type de données, un propriété de visibilité (privé ou public) et peut être initialisée. Le nom de l'attribut ou variable doit être unique dans une classe. On peut différencier les types d'attributs de la classe en deux catégories importants, ce sont les attributs de types primitifs et les attributs qui sont les références vers les autres classes.

#### 2.1.3.1 Types primitifs

Chaque attribut de la classe qui est de type prédéfini dans UMLest traduit par un champ de type prédéfini dans Fiacre. Le tableau au-dessous présente la traduction de type primitive dans UML vers le type de base Fiacre.

*Règle 3 :*
Chaque type prédéfini en UML sera traduit par un type primitif en Fiacre.

| UML | Fiacre |
|---|---|
| Short, Int, long, float, double, double long | int |
| Char, unsigned char, unsigned shor, unsigned int, unsigned long, unsigned int | nat |
| Bool | bool |
| Void | Int |

**Tableau 1 : traduction de types prédéfinis dans UML en Fiacre**

2.1.3.2 Types non primitifs

Chaque attribut non primitif d'une classe qui est une référence vers une autre classe inactive est traduit par un champ de record de type de record portant le nom la classe inactive correspondant.

*Règle 4 :*
Chaque attribut qui est une référence vers une autre classe inactive est traduit par un champ de record de type correspondant (Figure 7).

Chaque attribut qui est une référence vers une classe active est traduit par une instance du processus Fiacre correspondant à cette classe. Ce processus va permettre d'exécuter les opérations primitives intégrées dans cette classe. De plus il est à noter qu'une classe inactive ne contient que références à des classes inactives, pas à des classes actives. Et, une classe active peut contenir des références à des classes actives ou inactives.

*Règle 5 :*
Chaque attribut qui est une référence vers une autre classe active est traduit par une instance du processus correspondant.

2.1.3.3 Accès aux attributs d'une la classe

La valeur d'un attribut d'une classe peut être accédée en lecture ou en écriture selon les propriétés de visibilité de l'attribut. Lors la propriété est de type privé, la variable ne peut être que lue par un objet d'une autre classe. Si la propriété est de type public, la variable ne peut être lue ou modifiée par un objet d'une autre classe. (Par la suite, nous étudierons une optimisation de ce schéma en regroupant les variables (d'instances ou de classe) dans un seul processus).

L'accès en écriture à un attribut de type public doit être associé à un mécanisme de protection pour éviter les incohérences sur les valeurs de l'attribut. L'accès à une variable de ce type sera géré par un processus spécifique à cette variable (vue comme une variable partagée) excluant deux écritures simultanées par deux objets qui veulent modifier la valeur.

Nous illustrons ce mécanisme par la figure 9. Un processus gère l'accès en lecture et écriture de la valeur de l'attribut. Toute demande d'écriture et lecture sur la variable partagé est permise à un seul processus à la fois. La demande d'accès aux attributs s'effectue par l'envoi d'un message synchrone (communication par port). Une demande de lecture ( ?get) est suivie par un envoi (synchrone) de la valeur ( !data). Une demande d'écriture ( ?set) est

suivie par la modification de la variable (data := value) et un envoi (synchrone) d'un acquittement (ack).

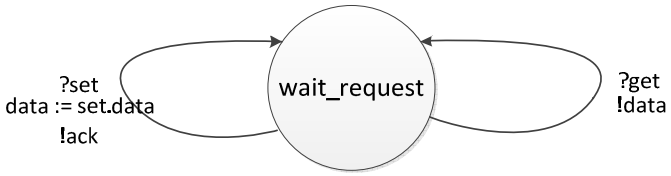



**Figure 9 : Automate d'un processus gérant l'accès concurrent à une variable *data* partagée**

Une variable (de classe ou d'instance) *v* d'une classe *C* doit être encapsulée par un processus spécifique *C_v*. Ce processus gère l'accès concurrent à *v*. Si un objet *o* d'une classe *C'* a besoin d'accéder (en lecture ou écriture) à la valeur de *v*, le processus implantant *o* envoi une demande d'accès à *v* au processus *C_v*. *C_v* répond en modifiant la valeur de *v* (en cas d'écriture par *o*) ou en envoyant la valeur *v* au processus *o*. Les demandes sont implantées via les ports et sont synchrones.

*Règle 6.a :*
Chaque attribut d'une classe est traduit par un processus spécifique qui encapsule une variable partagée. Ce processus gère l'accès concurrent en lecture et écriture à cette variable.

Ci-dessous, nous illustrons par un exemple de code Fiacre un processus qui gère l'accès concurrent aux attributs de la classe « LiftControl ». Lorsqu'il y a une instance de la classe « LiftControl », on crée une instance du processus « Mutex_LiftControl » pour contrôler l'accès à ses attributs.

Exemple :

```
type Type_Acknowledgement is union
        ACK
        | NACK
end union
type Type_GET_SET is union
        GET
        | SET
end union
type Request_Attr_LiftControl is record
        operation : Type_GET_SET,    // on pourrait utiliser ce champ pour spécifier si on fait une lecture ou
                                     // une écriture
        data: Type_LiftControl
end
type Return_Attr_LiftControl is record
        acknowledge: Type_Acknowledgement, //on pourrait utiliser ce champ pour spécifier l'acquittement
        data: Type_LiftControl
end

process Mutex_LiftControl [
                    from_itsLiftControl: in Request_Attr_LiftControl,
                    to_itsLiftControl: out Return_Attr_LiftControl,
                    from_itsHoistControl: in Request_Attr_LiftControl,
                    to_itsHoistControl: out Return_Attr_LiftControl] is
states wait_request, process_request, send_data, tmp_state_1, tmp_state_2, send_ack, tmp_state_3,
tmp_state_4
var
        localData: Type_LiftControl,
```

```
        return : Return_Attr_LiftControl,
        request : Request_Attr_LiftControl,
        object_port : int
init
        object_port := 0;
        to wait_request

from wait_request
        select
                from_itsLiftControl?request;
                object_port := 1;
                to process_request
        []
                from_itsHoistControl?request;
                object_port := 2;
                to process_request
end
from process_request
        case request.operation of
                GET ->
                        to send_data

                | SET ->
                        localData := request.data;
                        to send_ack
        end case
from send_data
        if (object_port = 1) then
                to tmp_state_1
        elsif (object_port = 2) then
                to tmp_state_2
        end
from tmp_state_1
        return.acknowledge := ACK;
        return.data := localData;
        to_itsLiftControl!return;
        to wait_request
from tmp_state_2
        return.acknowledge := ACK;
        return.data := localData;
        to_itsHoistControl!return;
        to wait_request
from send_ack
        if (object_port = 1) then
                to tmp_state_3
        elsif (object_port = 2) then
                to tmp_state_4
        end
from tmp_state_3
        return.acknowledge := ACK;
        to_itsLiftControl!return;
        to wait_request
from tmp_state_4
        return.acknowledge := ACK;
        to_itsHoistControl!return;
        to wait_request
```

**Figure 10 : Exemple d'un processus qui gère l'accès concurrent aux attributs**

*Règle 6.b :*

Pour chaque classe UML, on crée un processus qui joue le rôle comme le coordinateur pour gérer l'accès concurrent à ses attributs variables de classe.

*Règle 6.c :*

Pour chaque instance d'une classe UML, on crée un processus qui joue le rôle comme le coordinateur pour gérer l'accès concurrent à ses attributs variables d'instance.

### 2.1.4 Association entre les classes

2.1.4.1 Association simple

Lorsque deux classes sont reliées par une association, leurs instances peuvent échanger des messages entre eux [OMG 2010, LAU 2009]. Lors de transformation en Fiacre, il faut créer des chemins de communication entre le processus correspondant et le processus de la classe reliée. L'association entre les classes est précisée par des chemins de communications entre les processus Fiacre correspondants.

Normalement, on crée trois chemins de communication pour l'association avec la multiplicité « une », l'association unidirectionnelle. Un chemin de communication asynchrone est créé du processus émettant au processus recevant pour des envois d'évènements. Un chemin de communication synchrone est créé du processus appelant au processus recevant pour des appels de *triggered* opérations et un autre chemin de communication synchrone est créé pour des envois de messages de retour de *triggered* opérations.

*Règle 7 :*

Pour chaque association, des chemins de communication sont créés entre le processus correspondant et le processus de la classe reliée. Ces chemins véhiculent les envois des évènements et les invocations des opérations. (voir Annexes pour la classe « Door »)

2.1.4.2 Multiplicité

*Règle 8 :*

La multiplicité entre les classes est identifiée par le nombre de chemins des communications entre le processus correspondant et le processus de la classe reliée. (voir Annexes pour la classe « Door »)

### 2.1.5 Gestion des concepts d'objet

Les concepts d'objets sont importants dans le langage modélisation orienté objet UML, il faut donc les prendre en compte lors de transformation en Fiacre. Cependant, la richesse des concepts objets UML ne permet pas dans tous les cas une traduction correcte en Fiacre. Nous choisissons donc de limiter des concepts supportés par les transformations. Par exemple, l'héritage des machines d'états et la surcharge des opérations ne sont pas considérés.

2.1.5.1 Généralisation et Héritage

La généralisation décrit une relation entre une classe générale (classe de base ou classe parent) et une classe spécialisée (sous-classe). La classe spécialisée est intégralement

cohérente avec la classe de base, mais comporte des informations supplémentaires (attributs, opérations, associations). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé. Dans le langage UML, ainsi que dans la plupart des langages objet, cette relation de généralisation se traduit par le concept d'héritage. L'héritage de la machine d'états pose des problèmes sémantiques. Leur héritage n'est donc pas pris en compte dans la transformation en Fiacre.

**Attributs**

Lors de la transformation en Fiacre, on copie tous les attributs avec la visibilité *public* ou *protected* dans la classe mère et pour les incorporer dans la classe fille. Nous ne prenons pas en compte la surcharge des variables. Dans la hiérarchie d'une classe, chaque variable a un nom unique.

*Règle 9.a:*
Pour la classe fille, on réplique les attributs de la classe mère.

**Opérations et Evènements**

Le processus de la classe fille peut recevoir les appels d'opérations, l'envoi et la réception d'évènements supportés par les classes dont elle hérite [OMG 2010, LAU 2009]. Etant donné qu'on ne peut pas distinguer deux processus par le nom de paramètre et son type dans le programme Fiacre, alors nous ne prenons pas en compte la traduction de la surcharge de l'opération.

*Règle 9.b:*
Pour la classe fille, on réplique la capacité de recevoir les appels d'opérations et les évènements pouvant être reçus par la classe mère.

2.1.5.2 Variables temporaires

Lorsqu'un processus Fiacre reçoit un message, il stocke les valeurs des paramètres dans ses variables. De ce fait, une variable temporaire est créée pour chaque type de paramètre susceptible d'être reçu.

*Règle 10:*
Pour chaque message susceptible d'être reçu, des variables temporaires typées distinctes capables de stocker les valeurs des paramètres sont générées.

### 2.1.6 Ports

2.1.6.1 Ports Standard

Un port est une propriété de classe qui spécifie le point d'interaction entre l'instance de la classe et l'environnement, l'instance d'une autre classe ou un élément interne de la classe [OMG 2010]. Il spécifie aussi les services fournit ou requis par l'instance de la classe. Deux ports sont connectés par un connecteur standard.

*Règle 11:*
Lors de transformation en Fiacre, on traduit le port standard par les envois et les réceptions des messages synchrones et asynchrones entre les processus (voir la partie 2.2.2 et 2.2.3).

2.1.6.2 Flow Port

Le *Flow Port* nous permet de présenter le flux de donnée entre les objets dans le système sans utiliser des évènements et des opérations [TEL]. Il peut être ajouté à un objet dans le diagramme d'objets du système. Il nous permet de modifier un attribut dans un objet automatiquement quand un attribut du même type dans un autre objet change sa valeur.

**Hypothèse 1 :**

Pour chaque variable qui sera envoyé dans le *flow port*, alors il ne peut pas porter la visibilité *public*. De cette façon, on peut simplifier le problème de l'accès au variable de ne pas créer le processus complémentaire pour générer l'accès concurrent.

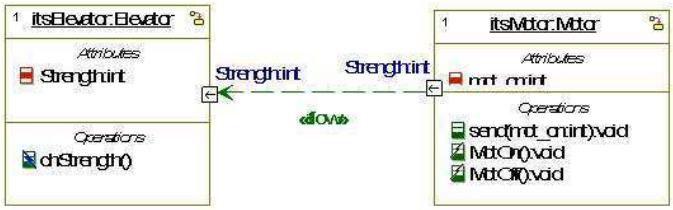
*Règle 12.a:*

On traduit le *flow port* par l'envoi d'un message dans un port qui connecte les deux instances du processus. Ce message est l'attribut envoyé dans le *flow port*. Lorsque la valeur de la variable dans le processus représenté l'objet source est changé par la méthode *setAttribut()*, il est envoyé dans le port au processus destinataire.

*Règle 12.b:*

Dans le cas où la valeur de l'attribut est envoyée dans le *flow port* à *n* objets, on synchronise les processus représentés ces objets par un seul port. Fiacre nous permet de synchroniser les processus par un seul port.

Exemple :




```
process Elevator [flow_Strenght_From_itsMotor : in int] is
states waitStrenght, applyStrenght…
var
        Strenght : int
        …
init
        …
...
from waitStrenght
        flow_Strenght_From_itsMotor?Strenght;
        …
        to applyStrenght
...

process Motor [flow_Strenght_To_itsElevator : out int] is
states Stop, tmp_state_4, Running,…
var
        Strenght : int
        …
```

```
init
        …
...
from tmp_state_4
        Strenght := newValue;  // modification de la variable dans la meme transition
                               // que l'écriture.
        flow_Strenght_To_itsElevator!Strenght
        to Running

…
```

**Figure 11 : la traduction du *flow port***

## 2.2 Traduction de la dynamique du système

Le diagramme d'états UML modélise les comportements internes d'un objet réactif à l'aide d'un automate à états fini. Ils représentent les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode, etc.). Alors, diagramme d'états est traduit pour relater la partie comportement du processus de la classe active correspondante. Lors de la transformation en Fiacre, les éléments suivants sont pris en compte: les états, les transitions, les actions, les sous-états et les sous-machines d'états.

### 2.2.1 Traduction de la transition

Une transition définit la réponse d'un objet à l'occurrence d'un événement [OMG 2010, HAR 1998, HAR 2001]. Elle lie généralement, deux états *E*1 et *E*2 et indique qu'un objet dans un état *E*1 peut entrer dans l'état *E*2 et exécuter certaines activités, si un événement déclencheur se produit et que la condition de garde est vérifiée.

*Règle 13 :*
Une transition UML peut être traduite par plusieurs états et transitions dans le processus Fiacre. Il dépend des actions dans le corps de la transition.

### 2.2.2 Envoi et réception des évènements

L'événement UML permet une communication asynchrone entre 2 objets ou un objet et l'environnement du le système [OMG 2010, HAR 1998, HAR 2001]. La réception d'un évènement peut changer l'état de l'objet recevant. Et, l'objet qui fournit l'évènement continue son exécution sans attendre le message de retour. Normalement, chaque classe UML définit un ensemble des évènements qu'il peut recevoir. L'ajout d'une réception d'un événement à un objet définit la capacité d'un objet pour recevoir ce type d'événement.

Tous les évènements reçus sont placés dans la *FIFO* des évènements de l'objet recevant. Ils sont envoyés plus tard, un à la fois, à la machine d'états. L'évènement qui est envoyé à la machine d'état est supprimé de la *FIFO*. Un évènement peut déclencher la transition dans la machine d'états. Un objet peut envoyer aussi un évènement vers lui-même. Au-dessous, des exemples de syntaxe pour envoyer un évènement sous *Rhapsody*.

*Envoi d'un évènement à un autre objet:*

$$OUT\_PORT(port) \rightarrow GEN(new\ events(p1, p2, p3, \dots, pn))\ ;$$

*Ou*

$$nom\_objet \rightarrow GEN(new\ events(p1, p2, p3, \dots, pn));$$

*Envoi d'un évènement vers lui-même :*

$$GEN(new\ events(p1, p2, p3, \dots, pn));$$

On traduit l'envoi d'un évènement UML par l'envoi d'un message asynchrone en Fiacre. L'envoi de message asynchrone entre les processus Fiacre est réalisé par un *FIFO* partagée. Alors, pour chaque processus qui peut recevoir des évènements on crée une *FIFO* correspond aux évènements susceptibles d'être reçus. Le processus qui envoie des évènements et le processus qui reçoit des évènements communiquent par cette *FIFO* partagée (Figure 13). Cependant, le message reçu est supprimé lorsqu'il est consumé par le processus (Figure 12). La réception d'un évènement non prévu par la machine d'états du processus destinataire est supprimée automatiquement de la *FIFO* du processus recevant.

*Règle 14 :*
On traduit l'envoi d'un évènement UML par l'envoi d'un message asynchrone en Fiacre. La réception d'un évènement non prévu par la machine d'états du processus destinataire est supprimée automatiquement de la *FIFO* (voir Annexes pour la classe « LiftControl »).

**Scénario d'envoi et réception d'un évènement**

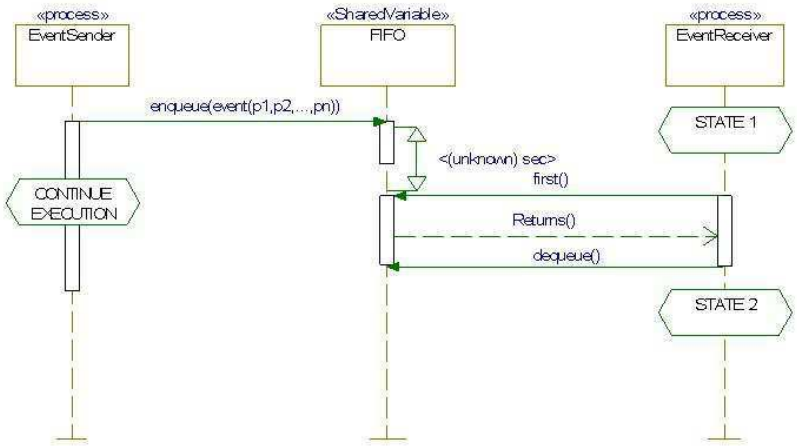



**Figure 12: scénario général d'envoi d'un évènement entre deux processus Fiacre**

Exemple :

Cet exemple présente l'envoi et la réception des évènements entre l'objet « itsLiftControl » et l'objet « itsHoistControl » par un *FIFO* partagé (voir Annexes).

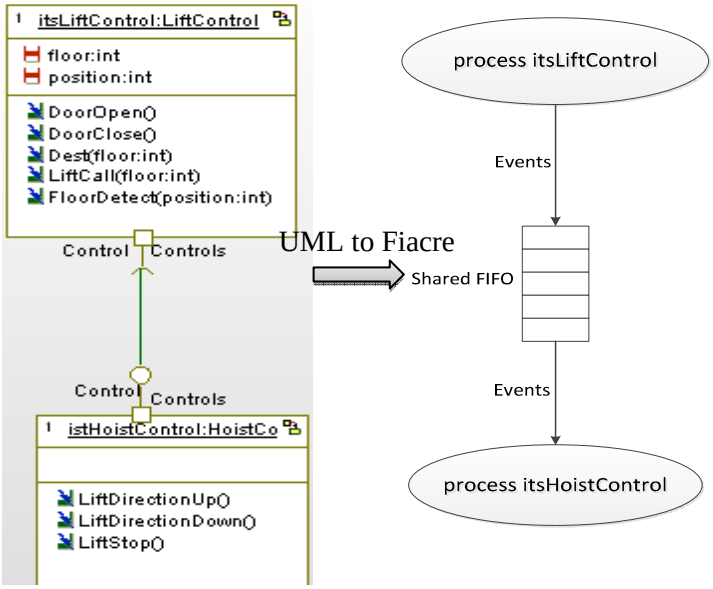



**Figure 13: l'envoi et la réception d'un évènement entre deux objets**

### 2.2.3 Appel et réception des opérations

Dans le modèle UML *Rhapsody*, on peut différencier les opérations en deux catégories importantes, qui sont les *triggered* opérations et les opérations primitives. L'exécution de l'objet qui fait l'appel à ce type d'opération est bloquée jusqu'à ce que l'exécution de l'opération soit finie et jusqu'à ce qu'il reçoit le signal d'acquittement.

#### 2.2.3.1 Triggered opération

La *triggered* opération implique la communication d'un évènement synchrone qui peut retourner une valeur. Elle fournit la communication synchrone entre les objets réactifs dans le système. Elle peut être appelée par un objet pour changer l'état d'un autre objet [HAR 1998, HAR 2001].

Le corps de la *triggered* opération est décrit par le langage d'action qui se trouve dans la transition où l'évènement est attendu. Alors, l'exécution de *triggered* opération peut être différente selon l'état de l'objet quand l'opération est invoquée. L'exécution de l'objet qui fait l'appel de la *triggered* opération est bloquée jusqu'à l'exécution de l'opération par l'objet appelé soit finie et que l'objet appelé lui envoie le message de retour. S'il y a plusieurs objets qui appellent une *triggered* opération sur la même machine d'états, les évènements sont envoyés à la machine d'états un à la fois. Alors, un seul appel d'une opération sur une machine d'états est autorisé à la fois et l'envoi de message d'acquittement se fait dans la transition entrant dans un état stable. L'état stable est un état dont la transition sortante ne peut pas être tirée sans la réception d'un appel d'évènement ou un appel d'une *triggered* opération.

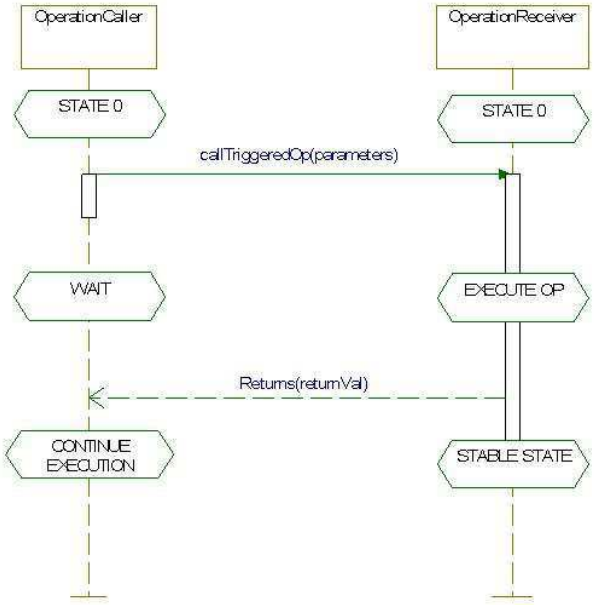



**Figure 14 : scénario d'appel d'une *triggered* opération UML**

On traduit l'appel d'une *triggered* opération par l'envoi d'un message synchrone en Fiacre (la communication par port). Pour chaque opération, on crée deux envois de message : un message correspondant à l'appel de l'opération et un autre message correspondant à la valeur de retour de l'opération. Les deux messages sont envoyés par deux ports connectés au processus appelant et au processus appelé. Les codes de l'opération sont exécutés directement dans la transition du processus appelé.

*Règle 15 :*
Un appel de *triggered* opération dans UML est traduit par un envoi d'un signal synchrone en Fiacre. Et le processus appelant est bloqué jusqu'à il reçoit le message de retour du processus appelé (voir Annexe pour la classe « Door »).

### Communication entre le processus appelant et celui appelé

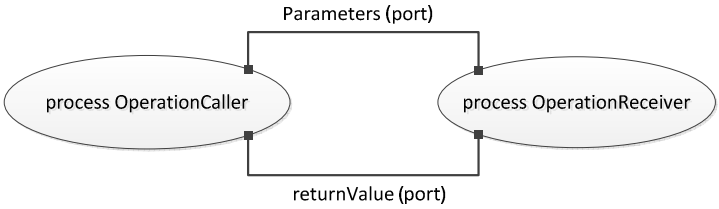



**Figure 15 : T*riggered* opération : communication entre le processus appelant et le processus appelé**

### Envoi d'une *triggered* opération

Nous faisons l'hypothèse qu'il n'y a pas d'appel de plusieurs *triggered* opérations dans une même transition. L'objet appelant est bloqué jusqu'il reçoit le message de retour de l'objet appelé.

*Règle 16 :*
En général, on doit créer deux états dans le processus appelant pour chaque appel de *triggered* opération. Dans un état, on fait l'appel de *triggered* opération et dans un autre état on attend pour le message de retour.

### Réception d'un appel de *triggered* opération

La réception d'un appel de *triggered* opération peut changer l'état de l'objet recevant. Ensuite, l'objet appelé envoie le message de retourne à l'objet appelant dans un état qui correspond à l'entrée dans un état stable dans la machine d'état UML (Figure 14).

*Règle 17 :*
En général, on doit créer deux états dans le processus appelé pour chaque réception de *triggered* opération. Dans un état on attend la réception de *triggered* opération et dans un autre état on envoie le message de retour au processus appelant.

Selon les cas, on a besoin de créer plusieurs états pour implanter les actions correspondant au corps (*body*) de l'opération

Exemple :

Dans cet exemple, la machine d'état de l'objet « itsMotor » attend l'appel de *triggered* opération « MotOn » avant de passer dans l'état « Running ». L'état « Running » est un état stable.

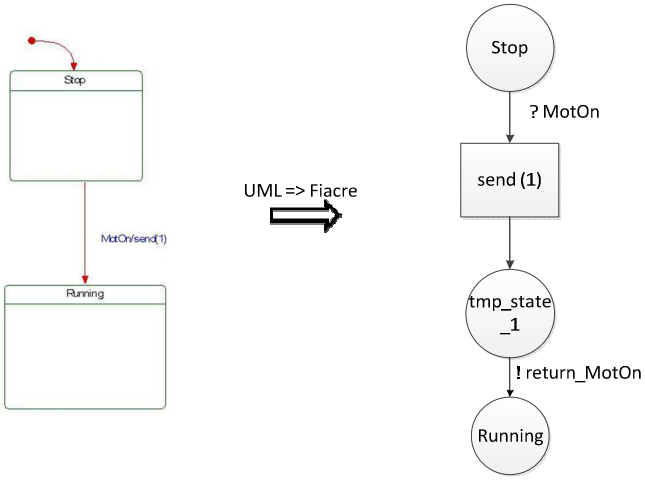



**Figure 16 : la réception de l'appel de *triggered* opération « MotOn »**

### *Triggered* opération et garde

Pour les transitions de sortie d'un état, la transition avec une garde doit être évaluée avant celle avec le *triggered* [HAR 1998, HAR 2001]. De ce fait, on crée un état qui permet d'évaluer la garde avant la réception de *triggered* opération et l'envoi de message de retour se fait dans les transitions de sortie de cet état (Figure 17).

Exemple :

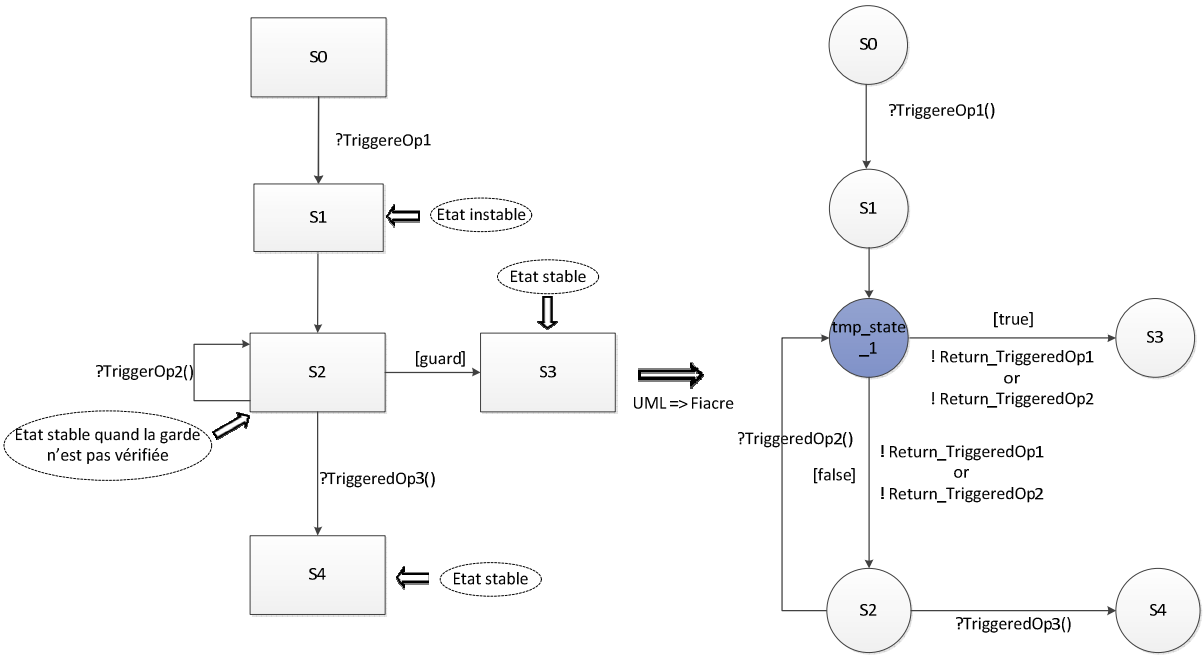



**Figure 17 : *triggered* opération et garde**

### Réception d'une triggered opération non prévue

Dans notre modèle Fiacre, il faut prendre en compte aussi la réception d'appel de *triggered* opération non prévu qui peut provoquer le *deadlock* dans le système. Prenons l'exemple dans la Figure 16, à l'état « Stop », le processus « Motor » peut aussi recevoir l'appel de *triggered* opération « MotOff » qui n'est pas présenté dans la machine d'état. Dans ce cas, le processus qui a fait l'appel la *triggered* opération « MotOff » est bloqué parce qu'il n'y a pas le message de retour du processus « Motor ». Rhapsody, lui, gère ce cas de figure automatiquement. La *triggered* « MotOff » ne permettant la prise d'aucune transition, l' « acquittement » est automatiquement envoyé.

*Règle 18 :*

Dans chaque état stable, pour chaque *triggered* opération non déjà utilisée dans une transition dans le modèle UML, on crée une nouvelle transition qui boucle sur l'état courant. Cette transition consomme la *triggered* opération et envoie directement un message d'acquittement au processus appelant.

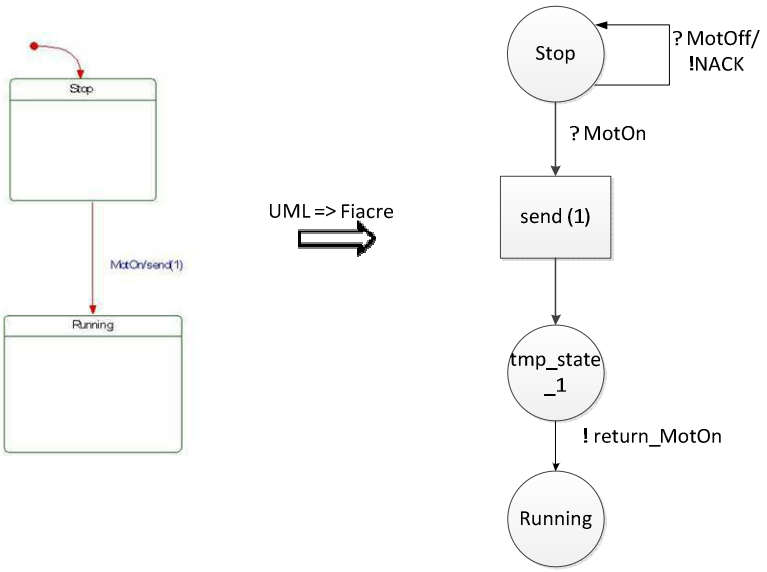



**Figure 18 : la réception d'une *triggered* opération, afin d'éviter un *deadlock***

On peut aussi prévoir une transition générique qui permet de se synchroniser avec l'appelant quelque soit l'opération évoquée. Un test peut être fait sur le type d'appel fait. La réponse de l'appelé dépend alors du type d'appel. Si un appel n'est pas prévu, l'appelé renvoie NACK. Si l'appel est prévu, l'appelé exécute l'opération et renvoie la réponse adéquate.

2.2.3.2 Opération primitive

Les objets peuvent se communiquer entre eux en appelant les opérations primitives. Quand un objet fait un appel d'oppération primitive, il se met dans l'état d'attente jusqu'à ce que l'exécution de l'opération soit finie et qu'il reçoive le message de retour. L'exécution de l'opération est indépendante de l'état du processus appelé. Quand un objet reçoit un appel de l'opération primitive, il crée un thread pour traiter cette opération [OMG 2010, HAR 2001].

En général, l'appel de l'opération primitive est effectué dans le body d'une autre opération ou dans les actions de machine d'états. L'invocation des opérations primitives sur un même objet en parallèle peut provoquer le problème d'accès concurrence aux variables partagés (voir la partie **Error! Reference source not found.**).

*Règle 19 :*
On va traduire un appel de l'opération primitive soit par un bloc de code qui s'exécute dans processus appelant soit par une instance du processus qui effectue cette opération.

**Réponse à une opération primitive par bloc de code**

La création d'une instance du processus dans le système consomme de la ressource de calcul. Alors, pour éviter la création d'une instance d'un processus pour chaque appel de l'opération, on va le traduire par un bloc du code qui s'exécute directement dans le processus appelant.

*Règle 20 :*

Pour chaque appel de l'opération primitive qui ne contient pas d'autres appels, l'opération primitive (Figure 19) sera traduite par un bloc de code qui s'exécute directement dans le processus appelant. Les états et les transitions complémentaires peuvent être créés selon les instructions dans cette opération.

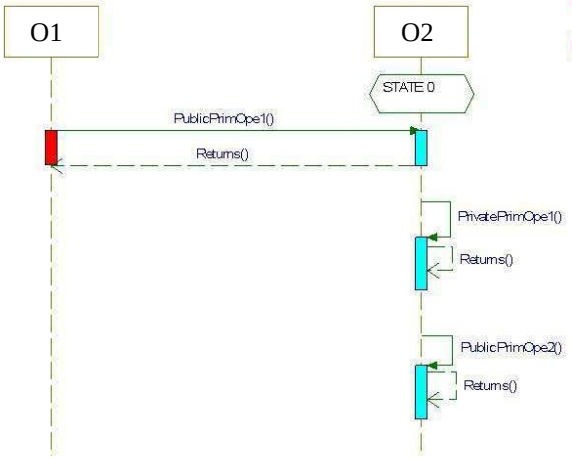


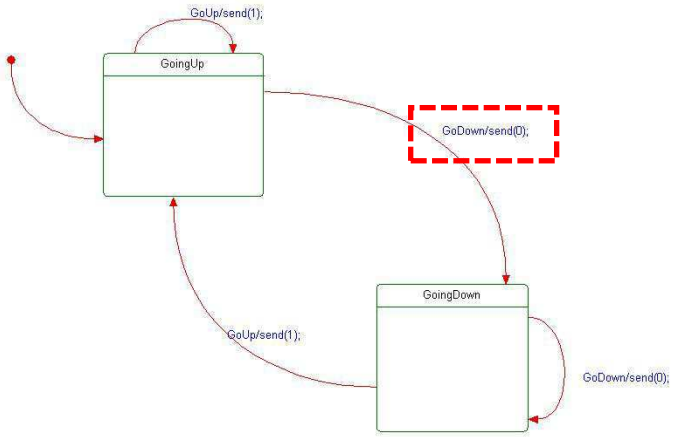

**Figure 19 : l'opération primitive qui est traduite par un bloc du code**

Etant donné que l'exécution du processus appelant est bloquée quand il fait un appel de l'opération primitive, alors on va effectuer l'exécution de l'opération directement dans le processus appelant pendant cette période.

Exemple :

Dans cet exemple, l'instance de la classe « Direction » fait l'appel d'une opération primitive « send(0) » à l'instance de la classe « ShaftSensor ». L'opération « send (up) » est une opération primitive de la classe « Direction » et est utilisé pour envoyer un évènement « Direc (up) » à la classe « ShaftSensor ».

*Le body de l'opération « send (up) »: OUT_PORT (direc) -> GEN (Direc (up));*



```
process Direction [call_from_itsHoistControl : in Type_Call_TrigOps_Direction,
return_to_itsHoistControl : out Type_Return_TrigOps_Direction]
(&events_To_itsShaftSensor : read write FIFO_Events_ShaftSensor, constructor :
Type_Direction) is
states GoingUp, GoingDown, tmp_state_3, …
var
        arg_itsShaftSensor_Direc : Type_ShaftSensor_Arg_Direc
        …
init
        …
        to GoingUp
from GoingUp
        …
from tmp_state_3
        if( full(events_To_itsShaftSensor)) then loop end;
        arg_itsShaftSensor_Direc.up := 0;
        events_To_itsShaftSensor := enqueue(events_To_itsShaftSensor, ShaftSensor_Direc
(arg_itsShaftSensor_Direc));
        to tmp_state_4
from tmp_state_4
        …
…
```

**Figure 20 : traduction d'une opération primitive par un bloque du code**

### Réponse à une opération primitive par la création d'un processus

Une opération primitive peut faire des appels à d'autres opérations primitives dans son body. Normalement, il peut avoir jusqu'à quatre niveaux d'appel d'opération consécutif dans un modèle des systèmes d'industriels que nous ciblons. Alors, pour réduire la complexité de transformation on va traduire ce type d'opération par un processus indépendant.

*Règle 21 :*
Pour chaque opération primitive qui fait des appels à d'autres opérations primitives, on va la traduire par un processus qui effectué cette opération.

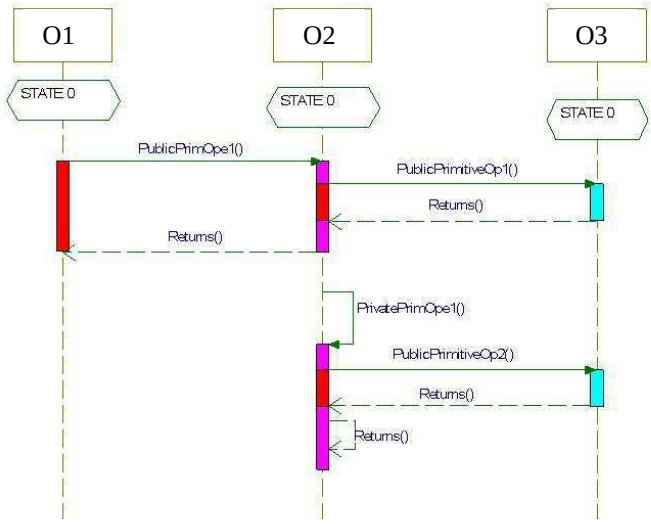



**Figure 21 : l'opération primitive qui est traduite par un processus**

*Processus de l'opération :*

Le processus qui effectue l'opération primitive peut avoir trois états principaux, ce sont « wait_request », « execute_operation », « send_return ». Les états et les transitions complémentaires peuvent être créés selon les instructions dans cette opération.

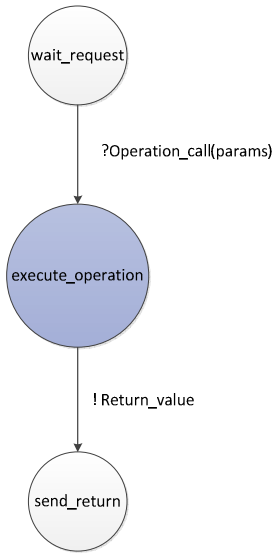


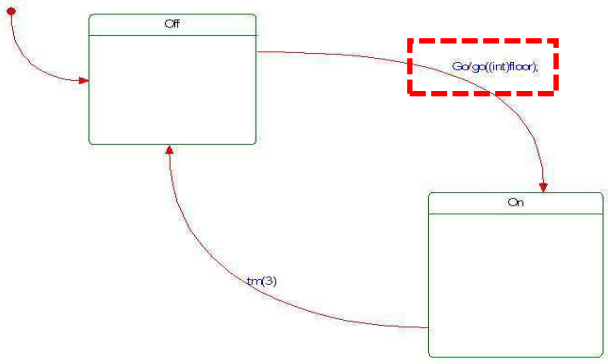

**Figure 22 : le processus qui effectue l'opération primitive**

Exemple :

Dans cet exemple, l'instance de la classe « LiftPanel » fait l'appel d'une opération primitive « go (floor) » à l'instance de la classe « LiftControl». Dans le body de l'opération « go (floor)», il y a un appel de l'opération « send (floor) ». L'opération « go (floor) » et « send (floor) » sont des opérations primitives de la classe « LiftPanel ».

*Le body de l'opération « send (floor) »: OUT_PORT (RequestFloor) -> GEN (Dest (floor));*

*Le body de l'opération « go (floor) »: send (floor);*




```
type Type_Call_PrimOps_go_LiftPanel is record
        go : int,
        floor : int
end

process PrimOps_go_LiftPanel [operation_call : in Type_Call_PrimOps_go_LiftPanel, return_value
: out int] (&events_itsLiftControl : FIFO_Events_LiftControl) is
state wait_request, execute_operation, send_return, Fin
var
```

```
        param : Type_Call_PrimOps_go_LiftPanel,
        return : int
        …
init
        …
        to wait_request
from wait_request
        operation_call?param;
        to execute_operation
from execute_operation
        if( full(events_itsLiftControl)) then loop end;
        events_itsLiftControl:=enqueue(events_itsLiftControl, LiftControl_Dest({floor =
        param.floor}));
        return := 0;
        to send_return
from send_return
        return_value!return;
        to Fin
from Fin
        loop

process LiftPanel [call_go_To_itsLiftControl : out in Type_Call_PrimOps_go_LiftPanel,
return_go_from_itsLiftControl: in int]
(&events_itsLiftPanel : read write FIFO_Events_LiftPanel) is
states Off, On, tmp_state_1
var
        event_Received : Type_Events_LiftPanel,
        arg_LiftPanel_Go : Type_LiftPanel_Arg_Go,
        return_itsLiftPanel_Go : int
        …
init
        event_Received := LiftPanel_NUL;
        …
        to Off
from Off
        …
from tmp_state_1
        call_go_To_itsLiftControl!arg_itsLiftPanel_Go;
        to tmp_state_2
from tmp_state_2
        return_value?return_itsLiftPanel_Go;
        to On
from On
        …
…
```

**Figure 23 : la traduction de l'opération primitive en un processus**

### 2.2.4 Traduction de l'action

Il peut y avoir une action à exécuter dans la transition ou à l'entrée et à la sortie d'un état. Cette action est composée par les instructions qui sont décrites en langage action de *Rhapsody*, par exemple C, C++, Java. Chaque instruction dans une action doit être invoquée séquentiellement et la machine d'états ne peut que continuer son exécution quand cette action est finie. Souvent, il peut y avoir des invocations d'opérations ou d'envois d'évènements dans l'action.

Etant donné que les actions dans la transition Rhapsody sont écrites dans le langage d'action, nous ne prenons pas en compte dans cette étude sa traduction pour l'instant.

*Règle 22:*

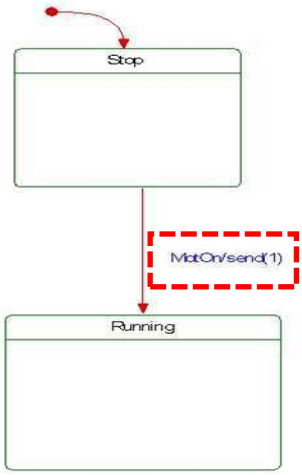
On traduit une action UML par une séquence d'actions, d'états, et de transitions dans le processus Fiacre. Cette séquence dépend du type d'instructions utilisées dans cette action.

2.2.4.1 Actions dans la transition

Une transition d'UML peut invoquer une action. Cette action est traitée dès que la transition est déclenchée. La machine d'état ne peut entrer dans l'état suivant qu'après que cette action soit finie.

Exemple :

Il y a un appel d'une opération primitive « send(1) » dans la transition sortant de l'état « Stop ». L'état « tmp_state_1 » et une transition sont créés pour réaliser cette action.




```
process Motor [call_from_itsHoistControl : in Type_Call_TrigOps_Motor,
return_to_itsHoistControl : out Type_Return_TrigOps_Motor]
(&events_To_itsShaftSensor : read write FIFO_Events_ShaftSensor) is
states Stop, tmp_state_1, Running, tmp_state_2, …
var
        receive_from_itsHoistControl: Type_Call_TrigOps_Motor,
        arg_itsShaftSensor_Mot : Type_ShaftSensor_Arg_Mot,
init
        receive_from_itsHoistControl := Motor_Call_NUL;
        to Stop
from Stop
        call_from_itsHoistControl?receive_from_itsHoistControl;
        case receive_from_itsHoistControl of
                Motor_Call_MotOn ->
                        to tmp_state_1
                any ->
                        to tmp_state_2
        end case
from tmp_state_1
        if( full(events_To_itsShaftSensor)) then loop end;
        arg_itsShaftSensor_Mot.mot_on := 1;
        events_To_itsShaftSensor := enqueue(events_To_itsShaftSensor,
```

```
ShaftSensor_Mot(arg_itsShaftSensor_Mot));
        to tmp_state_2
from tmp_state_2
        …
…
```

**Figure 24 : l'action dans la transition**

### 2.2.4.2 Action en entrée dans un état

Il peut y avoir des actions à invoquer à l'entrée dans un état UML. Ces actions sont exécutées dans l'étape quand l'état est entré, alors toute la transition entrée effectuée les mêmes actions. De ce fait, on doit alors traiter les actions dans chaque transition entrée indépendamment.

*Règle 23.a:*

S'il y a une action à l'entrée dans un état UML, on crée un état simple dont l'action est exécutée dans la transition sortante de cet état. Des états et les transitions complémentaires peuvent être crées selon les instructions dans cette action.

*Règle 23.b:*

S'il y a plusieurs actions potentielles à l'entrée dans un état UML, on crée un ensemble d'états pour chaque action à exécuter. Chaque action est traitée séparément et conformément à la règle 23.a.

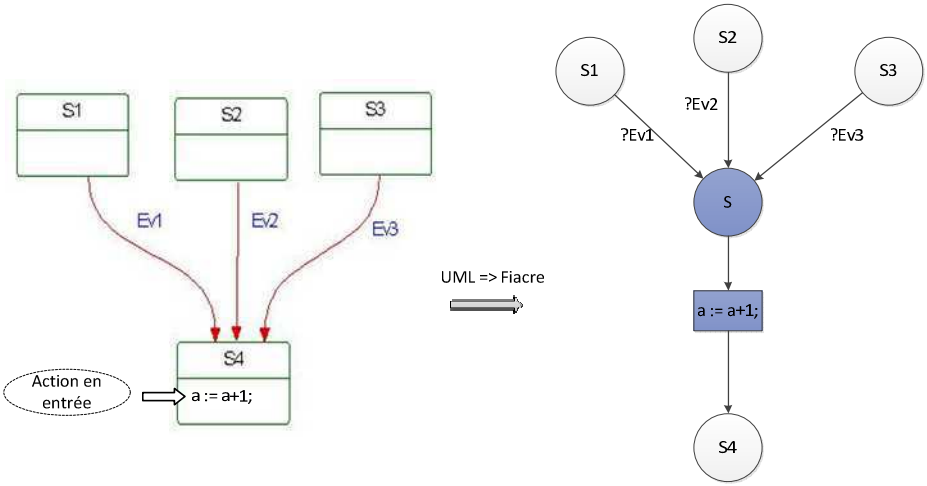



**Figure 25 : les actions entrées dans un état**

### 2.2.4.3 Action en sortie d'un état

Normalement, il peut avoir une action à invoquer à la sortie d'un état UML. En effet, cette action est exécutée dans l'étape quand l'état est sorti. Alors, une action sortante de l'état est traitée dépendamment à chaque transition sortie de cet état. S'il y a plusieurs transitions sorties, celles-ci ont forcément des facteurs déclenchant différents et peuvent avoir des états de destination différents. C'est-à dire, une seule transition est déclenchée à la fois. Les actions en sortie d'un état est effectué seulement s'il y a une des transitions est déclenché.

On ne peut pas adapter la même transformation que l'action à l'entrée dans un état parce que pour chaque transition sortie est déclenché par le facteur différent et peut avoir l'état destinataire différent. Alors l'action sortie de l'état est effectuée dans chaque transition sortant de l'état.

*Règle 24:*

On traduit l'action sortante d'un état UML par une action dans chaque transition sortant de cet état. Les états et les transitions complémentaires peuvent être crées selon les instructions dans cette action.

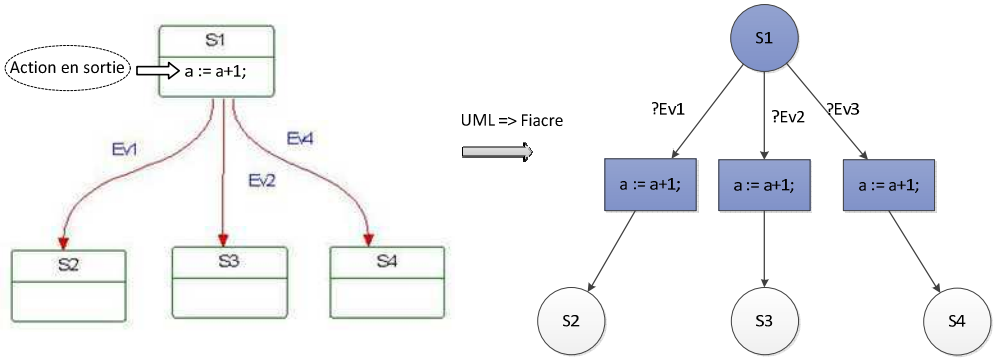



**Figure 26 : les actions sortant d'un état**

### 2.2.5 Traduction de l'état composite

L'état composite peut contenir une ou plusieurs régions qui contiennent un ensemble de sous-états. Dans chaque région, il peut y avoir un seul sous-état qui est actif à la fois. L'état composite avec une seul région est appelé l'état composite non-orthogonal (sous-état) et l'état composite avec plusieurs régions est appelé l'état composite orthogonal (AND state).

2.2.5.1 Etat composite non-orthogonal (sous-état)

Quand la machine d'état arrive dans un état qui est un état composite non-orthogonal, il entre dans cet état et en sort quand l'exécution des sous états est terminée. Dans l'état composite non-orthogonal, il peut y avoir un ensemble de sous états qui peut être activé un à la fois. L'entrée dans l'état composite peut se faire par une entrée par défaut, l'entrée dans un sous-état spécifique, ou un connecteur initial. Il faut absolument avoir une entrée par défaut dans le sous-état, si on veut éviter un problème de choix non-déterministe. La sortie de l'état composite non-orthogonal peut se faire par le pseudo état « final » ou par la réception d'un évènement ou d'une *triggered* opération qui peut déclencher la transition sortante de l'état composite et qui n'est pas prévue par le sous-état active. Quand l'état composite est sorti, les sous-états actifs sont sortis récursivement [OMG 2010, HAR 1998, HAR 2001].

**Entrée dans un état composite non-orthogonal :**

*Règle 25:*

Lors de transformation en Fiacre, on crée deux états pour chaque l'état composite non-orthogonal. Un état correspond à l'entrée dans l'état composite et un autre correspond à la sortie de l'état composite. L'exécution de l'état composite se fait directement dans l'exécution du processus principal.

Exemple :

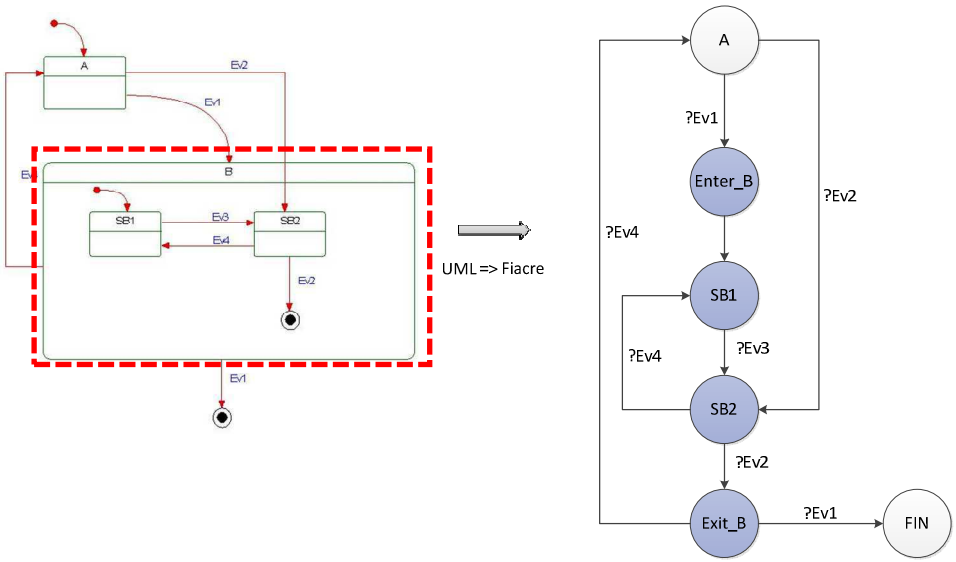



**Figure 27 : l'état composite non-orthogonal avec l'entrée par défaut**

### Sortie de l'état composite non-orthogonal :

*Règle 26:*

Pour chaque sous-état, on crée une transition vers l'état sortie de l'état composite. Cette transition peut être déclenchée par la réception d'un évènement ou d'une *triggered* opération qui peut trigger la transition sortie de l'état composite et qui n'est pas prévue par le sous-état.

Exemple :

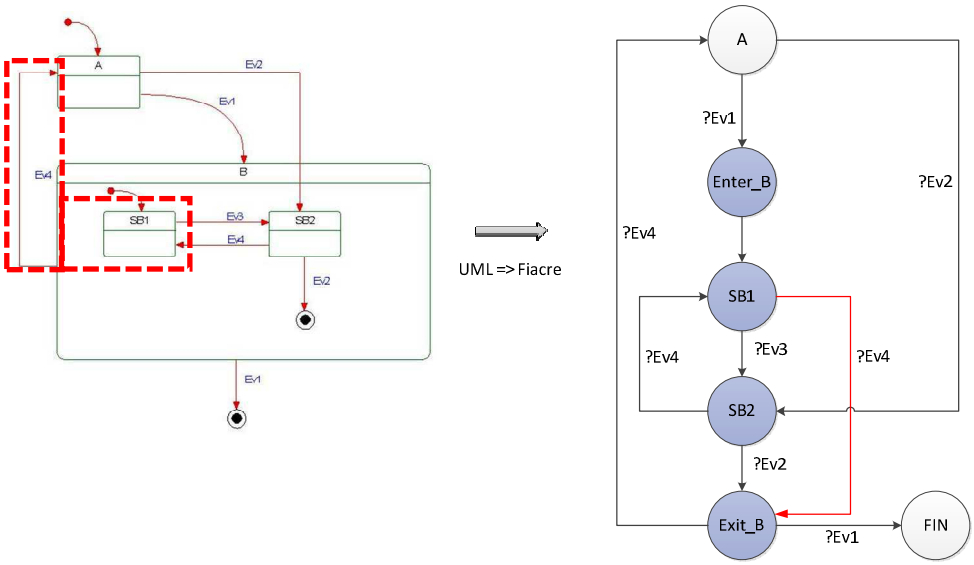



**Figure 28 : la sortie de l'état composite non-orthogonal**

2.2.5.2 Etat composite orthogonal (AND State)

Une classe ou un état peut posséder plusieurs machines d'états de même niveau qui sont exécutés en parallèle [OMG 2010, HAR 1998, HAR 2001]. Chaque machine état est encapsulée dans une région. La machine d'état de chaque région peut communiquer entre eux en envoyant des évènements, des appels des *triggered* opération, ou par le *FIFO* partagé.

*Règle 27:*

Lors de transformation en Fiacre, on crée un processus pour la machine d'état dans chaque région. Ces processus peuvent communiquer entre eux et les processus externes (les processus représentés des autres objets) par les envois des messages synchrones et asynchrones (voir la partie 2.2.2 et 2.2.3).

Cependant, le cas des transitions est un peu plus délicat car si deux machines d'états présentent la même condition au même moment, alors seule celle qui a la priorité la plus importante sera tirée. Pour gérer ce cas, un AND state ne doit pas posséder de transitions ayant une condition de tirage se retrouvant dans plusieurs machines d'états.

### 2.2.6 Traduction de sous-machine d'état

Normalement, il est possible d'avoir des états qui contiennent des sous-machines dans le système. Quand une machine d'états arrive dans un état dans lequel il possède une sous-machine d'état à invoquer, alors cette machine reste en attente jusqu'à la sous-machine d'états termine ou la transition sortant de état contenant est déclenchée. La sémantique de sous-machine d'état est équivalemment à celle de l'état composite. Mais, l'entrée et la sortie de sous-machine d'état se fait respectivement par le pseudo état « initial » et « final » [OMG 2010, HAR 1998, HAR 2001].

*Règle 28:*

Lors de transformation en Fiacre, on crée deux états pour chaque état qui possède une sous-machine d'état. Un état correspond à l'entrée dans la sous-machine et un autre correspond à la sortie de la sous-machine. L'exécution de la sous-machine d'état se fait directement dans l'exécution du processus principal.

Exemple :

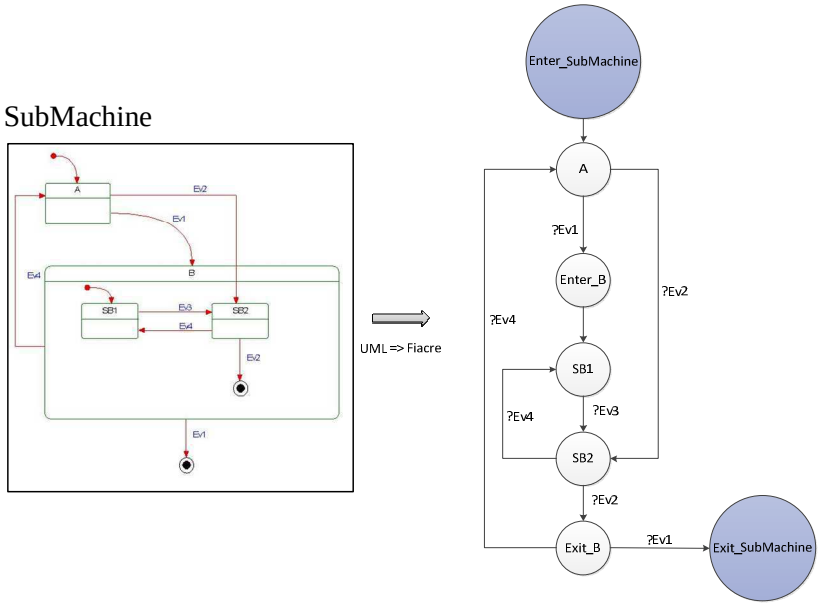



**Figure 29 : la traduction de sous-machine d'état**

## 2.3 Traduction du diagramme d'objets

Le diagramme d’état UML représente la configuration initiale du système. Il est alors traduit pour construire la partie composant du système. La classe d’un objet réactif et sa machine d’états sont traduites pour construire le processus Fiacre et son comportement.

Les objets sont utilisés pour identifier les instances du processus à instancier dans la partie composant du programme Fiacre. La relation entre les objets sera traduite par sa collaboration qui est implantée par des chemins de communication entre les processus Fiacre. Au-dessous, c’est un exemple du diagramme d’objets du système de l’ascenseur et sa traduction en Fiacre.

*Règle 29:*

Le diagramme d’objet UML est traduit pour construire la partie composant du système. La classe d’un objet réactif et sa machine d’états sera traduite pour construire le processus Fiacre et son comportement. La relation entre les objets sera traduite par sa collaboration qui est implantée par des chemins de communication entre les processus Fiacre. (voir Annexes pour la traduction du diagramme d’objet)

# Chapitre 3

# 3 Validation des règles de transformation

## 3.1 Validation des règles de transformation

Maintenant qu'on a défini les règles de transformation de modèles UML vers programmes Fiacre, il faut vérifier la correction de ces règles. Nous proposons deux approches dans le but de valider les règles de transformation:

1. Montrer la préservation des propriétés lors des transformations en Fiacre.
2. Montrer l'équivalence des sémantiques d'exécution entre le monde UML et le monde Fiacre.

On divise le travail de validation en deux étapes comme suit :

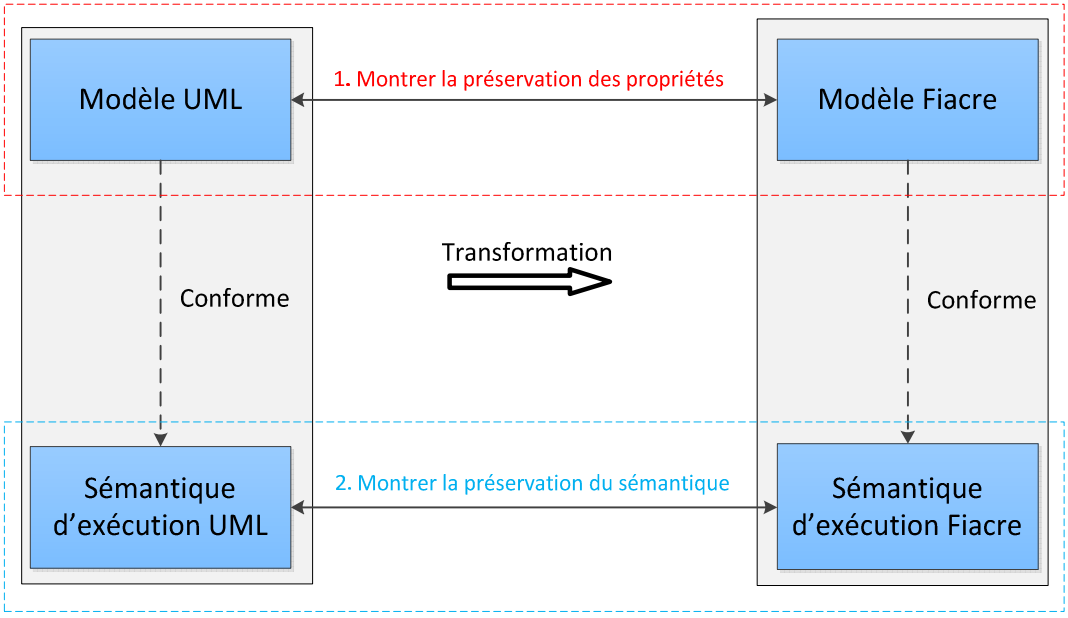



**Figure 30 : validation des règles de transformation**

Dans cette étude, nous nous concentrons sur la première approche. Pour cela, nous devons identifier les propriétés significatives à vérifier dans le modèle UML. Ensuite, ces propriétés sont interprétées et traduites dans le langage CDL [CDL 2012]. L'outil OBP acceptant le langage CDL permet d'explorer le modèle Fiacre généré et de vérifier les propriétés. La préservation des propriétés s'exprime par la règle suivante :

Soit *Sys : le système décrit en UML*

*Sys' : le système décrit en Fiacre*

*$\phi$: les propriétés du système UML*

*$\phi$': les propriétés du système Fiacre*

Si $\phi \mid= Sys \Rightarrow \phi' \mid= Sys'$

$\neg \phi \mid= Sys \Rightarrow \neg '\phi \mid= Sys'$

## 3.2 Propriétés à vérifier

Pour valider les règles de transformation, on va s'intéresser sur des propriétés suivantes qui sont beaucoup utilisées pour la validation du logiciel :

**Accessibilité :** Dire qu'un état est accessible, consiste à affirmer qu'il existe une exécution qui passe par cet état.

**Sûreté :** Les propriétés de sureté indiquent qu'un évènement ne se produira jamais. Ce type de propriété est donc simple à vérifier et par conséquent particulièrement adapté à la validation des logiciels de taille industrielle.

**Vivacité :** Une propriété de vivacité exprime qu'un évènement finira par avoir lieu. Elle est qualifiée de bornée si l'évènement attendu se produit dans un intervalle de temps borné.

L'outil *model-checker* OBP Explorer nous permet de définir des invariants et des automates d'observateurs dans le langage CDL. Avec des invariants, on peut vérifier les propriétés accessibilités et sûretés. Cependant des automates d'observateurs nous permettent de vérifier les propriétés vivacités et sûretés.

## 3.3 CDL

Un modèle CDL [CDL 2012] permet à l'utilisateur de décrire le comportement de l'environnement du modèle à valider et les propriétés devant être vérifiées. Le comportement est considéré comme des enchainements de scénarios qui décrivent les interactions entre le modèle soumis à validation et des entités composant l'environnement de ce modèle.

Ce DSL permet, d'une part, de décrire le comportement de plusieurs entités (nommés acteurs) contribuant à l'environnement et pouvant s'exécuter en parallèle. D'autre part, il intègre un langage de description de propriétés reposant sur la notion de patron et des automates observateurs.

Le langage est compilé par l'outil OBP (Observer-Based Prover) pour générer des graphes de comportements exploitables soit par l'outil OBP Explorer, soit par l'outil TINA SELT [BER 2004] via le langage Fiacre [FAR 2008].

## 3.4 Expression des propriétés

Il est difficile à identifier des bonnes propriétés UML à vérifier pour chaque règle de transformation. En fait, on peut limiter des propriétés à vérifier en éliminant celles qui sont triviales.

Exemple : Quand la transformation d'un pattern UML se fait par génération d'un pattern Fiacre, un pas de calcul UML pour un pas de calcul Fiacre, comme par exemple l'affectation :

x (uml) := e (uml) → x (fiacre) := e (fiacre), c'est trivial.

Alors, on peut identifier des propriétés importantes à vérifier dans le tableau au-dessous. Ensuite, on va les traduire en des invariants et des automates d'observateurs en langage CDL.

| Eléments UML | Propriétés à vérifier |
|---|---|
| échange d'évènement | • L'échange d'évènement est asynchrone. (est vérifié par la construction)<br>• Si un évènement est reçu et il n'est pas prévu par le processus recevant, il est supprimé de la *FIFO*. (est vérifié par la construction)<br>• L'évènement qui est envoyé dans la *FIFO* est toujours consommé par le processus destinataire.<br>• La réception d'un évènement peut faire changer l'état du processus recevant.<br>• L'évènement qui est consumé par le processus recevant est supprimé de la *FIFO*. (est vérifié par la construction) |
| appel de *triggered* opération | • L'appel de *triggered* opération est synchrone. (est vérifié par la construction)<br>• Si un appel de *triggered* opération est reçu et il n'est pas prévu par le processus appelé, un message NACK est envoyé au processus appelant. Le processus appelé ne change pas son état. (est vérifié par la construction)<br>• Le processus qui fait l'appel de *triggered* opération n'est pas toujours bloqué, néanmoins, il reçoit le message (NACK) pour débloquer son exécution.<br>• Il faut avoir un message de retourne pour chaque appel de l'opération. |

**Tableau 2 : propriétés nécessaires à vérifier**

Exemple :

On reprend l'exemple du système de l'ascenseur pour faire le test. Dans cet exemple, on veut tester des propriétés 1 et 2 dans le tableau au-dessus.

Dans le système le processus « LiftControl » et « Door » communique par l'envoie des évènements et l'appel des *triggered* opérations. On va prendre des propriétés intéressantes pour les écrire en CDL.

**Expression en CDL**

- ***Pty1*** : L'évènement qui est envoyé dans la *FIFO* est toujours consommé par le processus destinataire. La réception d'un évènement peut faire changer l'état du processus recevant.

```
predicate check_event is {{LiftControl}1:event_Received = DoorOpen}
predicate  ch_state is {{LiftControl}1:event_Received = DoorOpen  and {LiftControl}1@DoorOpened}
predicate not_change_state is {{LiftControl}1:event_Received = DoorOpen  and {LiftControl}1@idle}

event evt_to_LiftControl is {send DoorOpen from {Door}1 to {LiftControl}1}
event evt_from_Door1 is { receive DoorOpen from {Door}1 to {LiftControl}1}
event evt_check_event is {check_event becomes true}
event evt_change_state is {ch_state becomes true}
event evt_not_change_state is {not_change_state becomes true}

property pte_evt_from_Door1 is {
        start -- / / evt_to_LiftControl / -> send_event;
        send_event -- / / evt_from_Door1 / ->  rec_event;
        rec_event --/ / evt_check_event  / -> change_state;
        change_state -- / / evt_not_change_state / -> reject;
        change_state -- / / evt_change_state / -> success
}
```

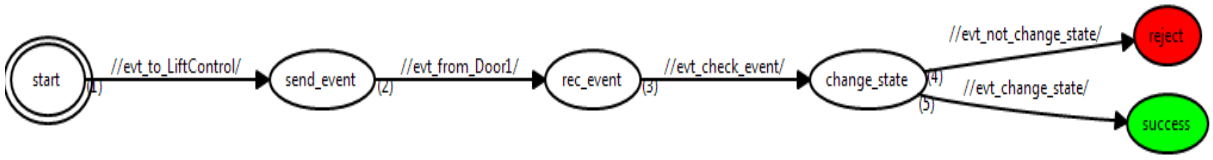



**Figure 31 : cdl graphe de propriété Pty1**

- ***Pty2*** : Le processus qui fait l'appel de *triggered* opération n'est pas toujours bloqué, néanmoins, il reçoit le message (NACK) pour débloquer son exécution. Il faut avoir un message de retourne pour chaque appel de l'opération.

```
predicate check_trigOps is {{Door}1:receive_trigOps_itsDoor = Open}
predicate door_change_state is {{Door}1@Opened}
predicate not_door_change_state is {{Door}1@Closed}

event send_Open_to_Door1 is { sync call_itsLiftControl_to_itsDoor1 (Open) from {LiftControl}1 to {Door}1}
event return_Open_to_LiftControl is { sync call_itsLiftControl_to_itsDoor1 (Open) from {LiftControl}1 to
{Door}1}
event evt_check_trigOps is {check_trigOps becomes true}
event evt_door_change_state is {door_change_state becomes true}
event evt_not_door_change_state is {not_change_state becomes true}

//Pty2.1
property pte_call_trigOps_to_Door1 is {
        start -- / / send_Open_to_Door1 / -> wait_return;
        wait_return -- / / return_Open_to_LiftControl / -> success

}

//Pty2.2
property pte_trigOp_to_Door1 is {
        start --/ / send_Open_to_Door1 / -> rec_trigOps;
        rec_trigOps --/ / evt_check_trigOps / -> send_return;
```

```
        send_return --/ / return_Open_to_LiftControl / -> change_state;
        send_return --/ / evt_not_change_state / -> reject;
        change_state --/ / evt_door_change_state / -> success
}
```

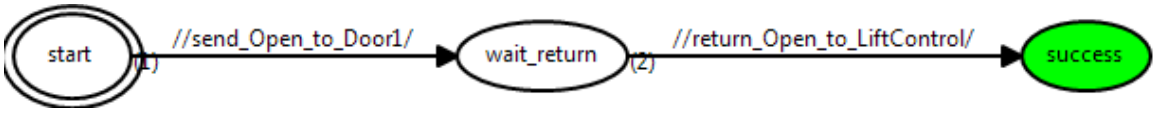



**Figure 32 : cdl graphe de propriété Pty2.1**

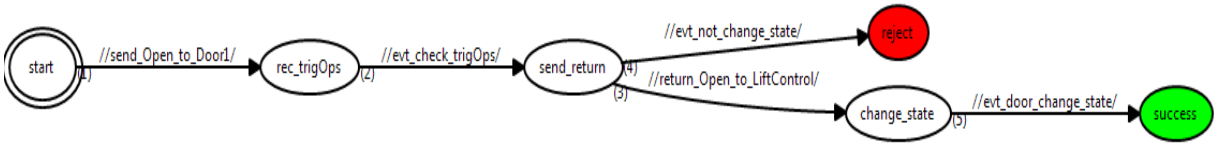



**Figure 33 : cdl graphe de propriété Pty2.2**

La figure au-dessous montre le scénario d'échange des messages synchrones et asynchrone entre le processus « LiftControl » et « Door ».

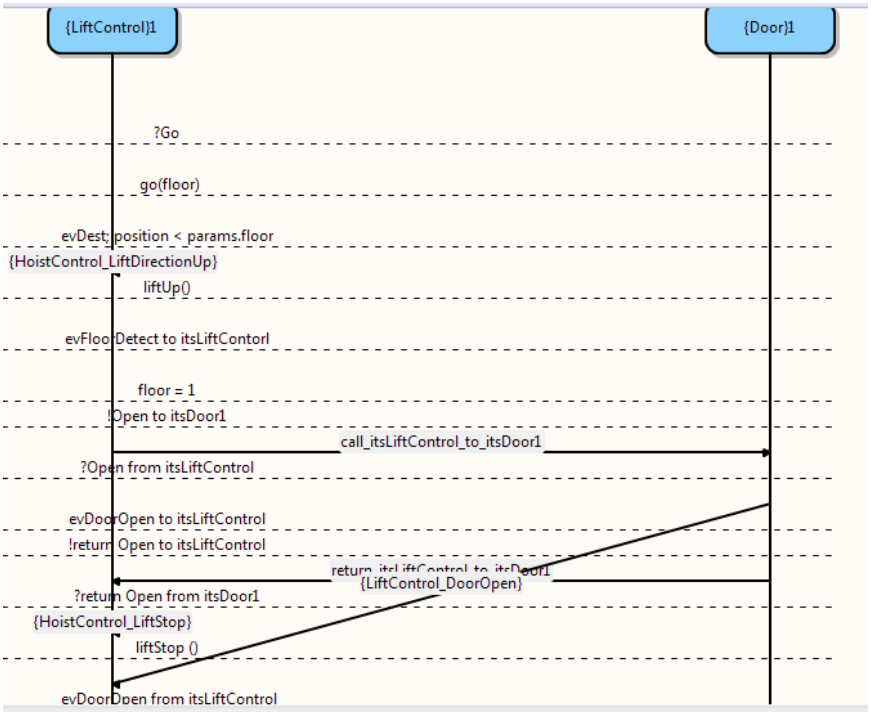



**Figure 34 : graphe d'exécution**

# Chapitre 4

# 4 Conclusion

## 4.1 Travail réalisé

Dans l'état actuel du travail, nous avons identifié des règles de transformation des éléments importants dans le modèle UML qui sont utilisés dans la modélisation des systèmes industriels ciblés. Ensuite, nous avons implanté ces règles de transformation dans l'outil *MDWorkbench* ou *Rhapsody Rules Composer*. Le langage d'action utilisée dans le modèle est en C++. Le programme Fiacre généré et les propriétés du système décrites en langage CDL sont exploités pour exécuter des preuves formelles avec l'outil OBP Explorer.

Nous avons commencé à étudier la validation formelle des règles de transformation mais ce travail est encore en cours et sera poursuivi durant le reste du stage de Master. Pour valider des règles de transformation, on propose deux approches différentes (voir la partie 3.1). On vérifie premièrement la préservation des propriétés du programme Fiacre généré. Nous avons défini des propriétés significatives (voir le **Error! Reference source not found.**) et les avons exprimés en langage CDL. Nous avons illustré notre travail sur un exemple de modèle de système d'un ascenseur. Nous avons vérifié quelques propriétés montrant ainsi leur préservation.(voir la partie 3.4).

## 4.2 Difficultés

En résumé, les points durs de notre travail ont été :

1. La difficulté à identifier les éléments de modèles UML à traduire.
2. La prise en compte de la sémantique UML lors des transformations.
3. L'identification des propriétés pour vérifier que les règles de préservation lors des transformations.

## 4.3 Perspectives

Pour la suite du travail, il est prévu de finir la partie formalisation des règles de transformation et vérifier la préservation de la sémantique du système généré. De plus, je vais améliorer le parseur pour qu'il puisse analyser des expressions plus complexes. En même temps, je vais essayer d'illustrer les règles de transformation sur des différents exemples de systèmes. Ceci permettra d'étudier la préservation d'autres propriétés.

En fin, mon travail sera intégré avec le travail de mon collègue sur « Formalisation des règles de traduction des Méta-modèles du sous profile Time de MARTE en Fiacre » pour pouvoir générer tous les éléments UML intégrant des aspects temporisés et non-temporisés en Fiacre.

# Références

[BER 2004] Berthomieu B., Ribet P.-O., Verdanat F., « The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », International Journal of Production Research, 2004.

[CIM 2000] Cimatti A., Clarke E., Giunchiglia F., Roveri M., « NuSMV : a new symbolic model checker », Int. J. on Software Tools for Technology Transfer, vol. 2, n° 4, p. 410425, 2000.

[CLA 1986] Clarke E., Emerson E., Sistla A., « Automatic verification of finite-state concurrent systems using temporal logic specifications », ACM Trans. Program. Lang. Syst., vol. 8, n° 2, p. 244- 263, 1986.

[DBR 2011] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, « Reducing State Explosion with Context Modeling for Model-Checking ». Hase'11, oct. 2011.

[FAR 2008] Farail P., Gaufillet P., Peres F., Bodeveix J.-P., Filali M., Berthomieu B., Rodrigo S., Vernadat F., Garavel H., Lang F., « FIACRE : an intermediate language for model verification in the TOPCASED environment », European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008, SEE, janvier, 2008.

[BER 2007] Bernard Berthomieu, Frédéric Lang, « Le langage pivot asynchrone Fiacre », réunion WP3 Topcased - LAAS - Toulouse, 13 novembre 2007.

[BER 2008] B. Berthomieu1, JP. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, F. Vernadat, « Fiacre: an Intermediate Language for Model Verification in the TOPCASED Environment », 4th European Congress ERTS Embedded Real Time Software, Toulouse (France), 29 Janvier - 1 février 2008.

[CDL 2012] Philippe Dhaussy, Jean-Charles Roger, « Spécification du langage CDL », version 2.0 Syntaxe et sémantique, 17 janvier 2012.

[FER 1996] Fernandez J.-C., Garavel H., Kerbrat A., Mounier L., Mateescu R., Sighireanu M., « CADP : A Protocol Validation and Verification Toolbox », CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification, Springer-Verlag, London, UK, p. 437-440, 1996.

[HOL 1997] Holzmann G., « The Model Checker SPIN », Software Engineering, vol. 23, n° 5, p. 279-295, 1997.

[LAR 1997] Larsen K. G., Pettersson P., Yi W., « UPPAAL in a Nutshell », International Journal on Software Tools for Technology Transfer, vol. 1, n° 1-2, p. 134-152, 1997.

[QUE 1982] Queille J.-P., Sifakis J., « Specification and verification of concurrent systems in CESAR », Proceedings of the 5th Colloquium on International Symposium on Programming, Springer- Verlag, London, UK, p. 337-351, 1982.

[HEN 1991] HENZINGER T., MANNA Z., PNUELI A., « Timed Transition Systems », Proceedings of the 1991 REX Workshop, 1991.

[SUS 2003] Susanne GRAPH, et Ileana OBER. « A Real-Time Profile for UML and how to adapt it to SDL ». Proceedings of the 11th international conference on System design, page 55-77, Springer-Verlag Berlin, Heidelberg 2003.

[IUL 2003] Iulian OBER, Susanne GRAPH, et Ileana OBER. « Validating UML models by simulation and verification ». laboratoire VERIMAG 2003.

[JOS 2004] Joseph HOOMAN, et Mark B. VAN DER ZWAAG. « A semantics of communicating Reactive Objects with Timing ». université de Nijmegen (pays bas), Janvier 2004.

[MAR 2002] Marius Bozga et Yassine Lakhnech. « IF-2.0 Common Language Operational Semantics », laboratoire VERIMAG, 29 septembre 2002.

[MAX 2008] Maxime FROMENTIN, « Identification des mécanismes de transformation entre UML et IF (V1) », 7 novembre 2008.

[JUL 2007] Julien AUVRAY, Mémoire de projet de fin d'études « Validation formelle de modèles UML », promotion 2007.

[MAX 2009] Maxime FROMENTIN, « Identification des mécanismes de transformation entre UML et IF (V4) », version du 26 février 2009.

[HAB 2004] Habart OLIVIER, Rapport de PFE « Modélisation et validation formelles de propriétés en environnement », 23 juillet 2004.

[OBP 2011] Philippe Dhaussy, Jean-Charles Roger. « OBPe (OBP Explorer) V. 1.2 Documentation », http://www.obpcdl.org, 28 décembre 2011

[BER 2003] B. Berthomieu, P.-O. Ribet, F. Vernadat, J. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Fi-lali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert, «Towards the verification of real-time systems in avionics: the Cotre approach », volume 80 of Electronic Notes in Theoretical Computer Science, pages 201–216. Elsevier, June 2003.

[HUB 2002] Hubert Garavel et Frédéric Lang, «NTIF: A general symbolic model for communicating sequential processes with data », volume 2529 of Lecture Notes in Computer Science, pages 276–291. Springer Verlag, November 2002.

[RHA 2008] « Rhapsody systems Engineering Tutorial », Copyright IBM Corporation 1997, 2008.

[REL 2009] « Relational Rhapsody User Guide », Copyright IBM Corporation 2000, 2009.

[TEL] « Telelogic Academy , Advanced Rhapsody, Data flow – Event Parameters ».

[OMG 2010] « OMG Unified Modeling Language™ (OMG UML), Superstructure », Version 2.3 without change bars, http://www.omg.org/spec/UML/2.3/Superstructure, 05 mai 2010.

[LAU 2009] Laurent AUDIBERT, « UML 2 », http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML.html#htoc184, 01 décembre 2009.

[HAR 2001] David Harel et Hillel Kugler, « The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) », Preliminary Version 2001.

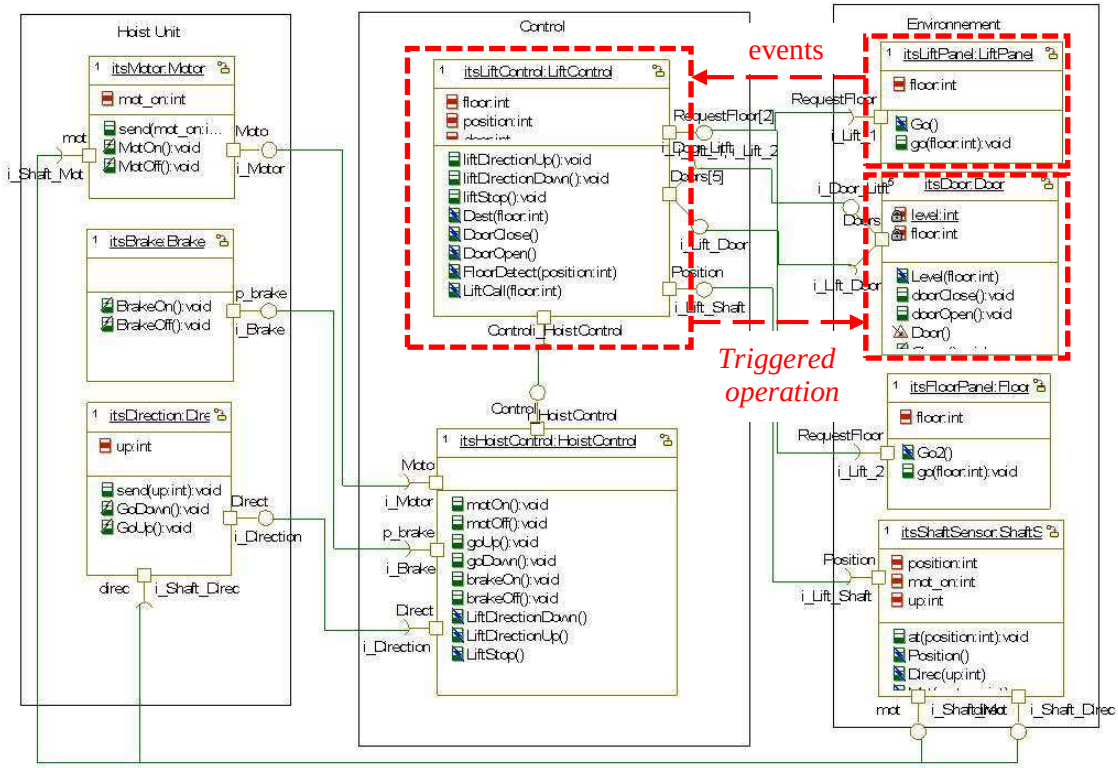
[HAR 1998] David Harel et Michal Politi, « Modeling Reative Systems with Statecharts, The Statemate Approach », Copyright 1998 by The McGraw-Hill Companies.

[RAJ 2010] Rajesh Singh , « Working with Rules Composer in Rational Rhapsody, Rules-based code generation engine for Rational Rhapsody », August 20, 2010.

[SOU 2008] Philippe SOULARD, « MDWorkbench, A Model-Driven Engineering Tool », GDR GPL – Toulouse – ENSEEIHT, 27 janvier 2008.

# Annexes

- *Diagramme d'objet de l'ascenseur :*




- *Processus de la classe LifPanel:*

```
process LiftPanel (&events_itsLiftPanel : read write FIFO_Events_LiftPanel, &events_itsLiftControl :
FIFO_Events_LiftControl) is
states Off, On, tmp_state_1
var
        event_Received : Type_Events_LiftPanel,
        attributes : Type_LiftPanel,
        arg_LiftPanel_Go : Type_LiftPanel_Arg_Go
init
        event_Received := LiftPanel_NUL;
        to Off
from Off
        if( empty(events_itsLiftPanel)) then loop end;
        event_Received := first (events_itsLiftPanel);
        events_itsLiftPanel := dequeue (events_itsLiftPanel);
        case event_Received of
                LiftPanel_Go arg_LiftPanel_Go->
                        attributes.floor := arg_LiftPanel_Go.floor;
                        to tmp_state_1
        end case;
        loop
from tmp_state_1
        if( full(events_itsLiftControl)) then loop end;
        events_itsLiftControl := enqueue(events_itsLiftControl, LiftControl_Dest({floor =
attributes.floor}));
        to On
from On
```

```
        wait[3, 3];
        to Off
```

- *Processus de la classe « Door »:*

```
process Door [call_from_itsLiftControl : in Type_Call_TrigOps_Door, return_to_itsLiftControl : out
Type_Return_TrigOps_Door] (&events_To_itsLiftControl : read write FIFO_Events_LiftControl) is
states Closed, Opened, tmp_state_2_1, tmp_state_3, ..
var
        receive_trigOps_itsDoor : Type_Call_TrigOps_Door,
        attributes : Type_Door,
        …
init
        receive_trigOps_itsDoor := Door_Call_NUL;
        …
        to Closed

from Closed
        call_from_itsLiftControl?receive_trigOps_itsDoor;
        case receive_trigOps_itsDoor of
                Door_Call_Open ->
                        to tmp_state_2
                any ->
                        to tmp_state_3
        end case;
from tmp_state_2
        if( full(events_To_itsLiftControl)) then loop end;
        events_To_itsLiftControl := enqueue(events_To_itsLiftControl, LiftControl_DoorOpen);
        to tmp_state_2_1
from tmp_state_2_1
        return_to_itsLiftControl!Door_Return_Open;
        to Opened
from tmp_state_3
        return_to_itsLiftControl!Door_Return_NACK;
        to Closed
…
```

- *Processus de la classe « LiftControl »:*

```
process LiftControl [call_to_itsDoor1 : out Type_Call_TrigOps_Door, return_from_itsDoor1 : in
Type_Return_TrigOps_Door, call_to_itsDoor2 : out Type_Call_TrigOps_Door, return_from_itsDoor2 : in
Type_Return_TrigOps_Door, call_to_itsDoor3 : out Type_Call_TrigOps_Door, return_from_itsDoor3 : in
Type_Return_TrigOps_Door, call_to_itsDoor4 : out Type_Call_TrigOps_Door, return_from_itsDoor4 : in
Type_Return_TrigOps_Door, call_to_itsDoor5 : out Type_Call_TrigOps_Door, return_from_itsDoor5 : in
Type_Return_TrigOps_Door] (&events_To_itsLiftControl : read write FIFO_Events_LiftControl) is

states idle, Going_Down, Going_Up, DoorOpened, tmp_state_1, tmp_state_2, tmp_state_3, tmp_state_4,
tmp_state_4_1, tmp_state_5,
tmp_state_5_1, tmp_state_6, tmp_state_6_1, tmp_state_7, tmp_state_7_1, tmp_state_8, tmp_state_8_1,
tmp_state_9

var
        event_Received : Type_Events_LiftControl,
        attributes : Type_LiftControl,
        arg_LiftControl_FloorDetect : Type_LiftControl_Arg_FloorDetect,
        arg_LiftControl_LiftCall : Type_LiftControl_Arg_LiftCall,
        arg_LiftControl_Dest : Type_LiftControl_Arg_Dest,
        return_from_itsDoor : Type_Return_TrigOps_Door
```

```
init
        event_Received := LiftControl_NUL;
        return_from_itsDoor := Door_Return_NACK;
        to idle

from idle
        if( empty(events_To_itsLiftControl)) then loop end;
        event_Received := first (events_To_itsLiftControl);
        events_To_itsLiftControl := dequeue (events_To_itsLiftControl);
        case event_Received of
                LiftControl_Dest arg_LiftControl_Dest ->
                        attributes.floor := arg_LiftControl_Dest.floor;
                        if((attributes.position > arg_LiftControl_Dest.floor) and (attributes.door = 0))
                        then
                                to tmp_state_1
                        elsif((attributes.position < arg_LiftControl_Dest.floor) and (attributes.door = 0))
                        then
                                to tmp_state_2
                        end
        |       LiftControl_DoorOpen ->
                        attributes.door := 1;
                        to DoorOpened
        end case;
        loop
from tmp_state_1
        …
        to Going_Down
from tmp_state_2
        …
        to Going_Up

from Going_Down
        if( empty(events_To_itsLiftControl)) then loop end;
        event_Received := first (events_To_itsLiftControl);
        events_To_itsLiftControl := dequeue (events_To_itsLiftControl);
        case event_Received of
                LiftControl_FloorDetect arg_LiftControl_FloorDetect->
                        if(arg_LiftControl_FloorDetect.position > attributes.floor) then
                                attributes.position := arg_LiftControl_FloorDetect.position;
                                to tmp_state_1
                        elsif (arg_LiftControl_FloorDetect.position = attributes.floor) then
                                attributes.position := arg_LiftControl_FloorDetect.position;
                                to tmp_state_3
                        end
        end case;
        loop

from tmp_state_3
        if(attributes.floor = 1) then
                to tmp_state_4
        elsif(attributes.floor = 2) then
                to tmp_state_5
        elsif(attributes.floor = 3) then
                to tmp_state_6
        elsif(attributes.floor = 4) then
                to tmp_state_7
        elsif(attributes.floor = 5) then
                to tmp_state_8
        end
from tmp_state_4
```

```
        call_to_itsDoor1!Door_Call_Open;
        to tmp_state_4_1
from tmp_state_4_1
        return_from_itsDoor1?return_from_itsDoor;
        to tmp_state_9
…
from tmp_state_9
        …
        to idle
…
```

- *Traduction de diagramme d'objet:*

```
component Elevator is

var
        LiftPanel_1 : FIFO_Events_LiftPanel,
        toContext : FIFO_Events_LiftPanel,
        events_To_itsLiftControl : FIFO_Events_LiftControl
port
        call_itsLiftControl_to_itsDoor1 : Type_Call_TrigOps_Door in [0, 0],
        return_itsLiftControl_to_itsDoor1 : Type_Return_TrigOps_Door in [0, 0],
        call_itsLiftControl_to_itsDoor2 : Type_Call_TrigOps_Door in [0, 0],
        return_itsLiftControl_to_itsDoor2 : Type_Return_TrigOps_Door in [0, 0],
        call_itsLiftControl_to_itsDoor3 : Type_Call_TrigOps_Door in [0, 0],
        return_itsLiftControl_to_itsDoor3 : Type_Return_TrigOps_Door in [0, 0],
        call_itsLiftControl_to_itsDoor4 : Type_Call_TrigOps_Door in [0, 0],
        return_itsLiftControl_to_itsDoor4 : Type_Return_TrigOps_Door in [0, 0],
        call_itsLiftControl_to_itsDoor5 : Type_Call_TrigOps_Door in [0, 0],
        return_itsLiftControl_to_itsDoor5 : Type_Return_TrigOps_Door in [0, 0]
init
        toContext := {||};
        LiftPanel_1 := {||};
        events_To_itsLiftControl := {||}

par
        LiftPanel (&LiftPanel_1, &events_To_itsLiftControl)
        || LiftControl [call_itsLiftControl_to_itsDoor1, return_itsLiftControl_to_itsDoor1,
                call_itsLiftControl_to_itsDoor2, return_itsLiftControl_to_itsDoor2,
                call_itsLiftControl_to_itsDoor3, return_itsLiftControl_to_itsDoor3,
                call_itsLiftControl_to_itsDoor4, return_itsLiftControl_to_itsDoor4,
                call_itsLiftControl_to_itsDoor5, return_itsLiftControl_to_itsDoor5]
                (&events_To_itsLiftControl)
        || Door [call_itsLiftControl_to_itsDoor1, return_itsLiftControl_to_itsDoor1]
                (&events_To_itsLiftControl)
        || Door [call_itsLiftControl_to_itsDoor2, return_itsLiftControl_to_itsDoor2]
                (&events_To_itsLiftControl)
        || Door [call_itsLiftControl_to_itsDoor3, return_itsLiftControl_to_itsDoor3]
                (&events_To_itsLiftControl)
        || Door [call_itsLiftControl_to_itsDoor4, return_itsLiftControl_to_itsDoor4]
                (&events_To_itsLiftControl)
        || Door [call_itsLiftControl_to_itsDoor5, return_itsLiftControl_to_itsDoor5]
                (&events_To_itsLiftControl)
end

Elevator
```