N° d'ordre :

N° de série :

République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Superieur et de la Recherche Scientifique

---

UNIVERSITÉ ECHAHID HAMMA LAKHDAR D'EL OUED

FACULTÉ DESSCIENCES EXACTES

DEPARTEMENT D'INFORMATIQUE

Mémoire de fin d'étude

MASTER ACADEMIQUE

Domaine: Mathématiques et Informatique
Filière: Informatique
Spécialité: Systèmes Distribués et Intelligence Artificiel (SDIA)

Thème

Transformation de diagramme d'activité UML vers FoCaLiZe

Présenté par

FERHAT HAMIDA Imane SALHI Noura

Soutenu devant le jury composé de

M.	ABBAS Messaoud	Rapporteur	Univ. d'El Oued
M.		Présidente	Univ. d'El Oued
M.		Examinatrice	Univ. d'El Oued

Année universitaire 2015/2016

Remerciements

Nous remercions tout d'abord notre Dieu qui nous a donné la force et la volonté pour élaborer ce travail.

Nous tenons à exprimer toutes nos gratitudes à notre encadreur

Mr ABBAS Messaoud *pour nous avoir proposé ce travail, pour son encadrement, son écoute, ses élucidations, ses conseils, ses directives et encouragements qu'il nous afflué.*

Nous remercions vivement les membres du jury qui nous ont fait l'honneur de présider le jury de notre soutenance.

*Notre reconnaissance va aussi à tous ceux qui ont collaboré à notre formation en particulier les enseignants du département d'**Informatique**, université d'**El-Oued**.*

*Aussi à nos collègues de la promotion **2015-2016***

Master Informatique,

Nous remercions également tous ceux qui ont participé de près ou de loin à élaborer ce travail.

Résumé

UML est un langage de modélisation objet possédant une popularité incomparable à la fois dans le monde industriel qu'académique. Mais, il demeure néanmoins une méthode semi-formel parce qu'il n'offre pas la possibilité pour une vérification et preuve formelle. D'autre part, les méthodes formelles permettent de s'assurer de la fiabilité des logiciels par des procédés mathématiques. Un exemple de ces outils est le système FoCaLiZe, un environnement de développement orienté objet et de la programmation certifiée.

Dans ce mémoire, nous proposons une approche MDE ("Model Driven Engineering") pour de transformation automatique des diagrammes d'activité d'UML en spécifications FoCaLiZe. Pour atteindre cet objectif, nous avons opté pour une approche directe (transformationnelle), en définissant une règle de transformation pour chaque constituant de diagramme d'activité. Pour mettre en œuvre notre démarche de transformation, nous utilisons le langage XSLT (Extensible Stylesheet Language Transformation), que nous permettra de générer une spécification FoCaLiZe à partir d'un document XMI (XML Meta data Interchange).

Mots clés : diagramme d'activité d'UML, FoCaLiZe, MDE, XSLT, XMI.

Abstract

UML is an object modeling language with the unparalleled popularity in both the industrial and academic world. But it still remains a semi-formal method because it does not offer the possibility for verification and formal proof. On the other hand, formal methods to ensure software reliability through mathematical processes. An example of these tools is the Focalize system, an object-oriented development environment and certified programming.

In this paper, we propose an MDE ("Model Driven Engineering") approach for automatic conversion activity diagrams of UML in Focalize specifications. To achieve this, we opted for a direct approach (transformational), defining a transformation rule for each activity diagram component. To implement our transformation process, we use XSLT (Extensible Stylesheet Language Transformation), we will generate; FoCaLiZe a specification from a document XMI (XML Metadata Interchange).

Keywords: activity diagrams of UML, FoCaLiZe, MDE, XSLT, XMI.

ملخص

UML هو لغة نمذجة (تخطيطية) كائنية تمتلك شعبية لا مثيل لها بالمرة في العالم الصناعي والأكاديمي. ولكنه مع ذلك لا يزال وسيلة معتمدة من طرف OMG ,إلا انها لا تتيح إمكانية التحقق والإثبات المنهجي. من ناحية أخرى، الأدوات الصورية تسمح بضمان موثوقية البرمجيات من خلال العمليات الحسابية. مثال على هذه الأدوات هو نظام FoCaLiZe، بيئة تطوير للبرمجة كائنية التوجُّه والبرمجة المعتمدة.

في هذه المذكرة، نقترح طريقة MDE ("Model Driven Engineering") للتحويل التلقائي للرسم البياني الحركي الخاص بـ UML وفق مواصفات FoCaLiZe. ولتحقيق ذلك، اخترنا طريقة مباشرة (تحويلية)، عن طريق تعريف قاعدة التحول لكل مكون للرسم البياني الحركي. لتشغيل عملية التحوي ، نستخدم لغة برمجة XSLT ، التي سوف تسمح بإنتاج مواصفة FoCaLiZe انطلاقا من ملف XMI (XML Meta data Interchange) .

الكلمات المفتاحية: الرسم البياني الحركي التابع لـUML, FoCaLiZe , XSLT, MDE , XMI.

Sommaire

Chapitre I: Les diagrammes d'activité

Chapitre II : L'environnement FoCaLiZe

Chapitre III : Formalisation des diagrammes d'activité

Chapitre IV : La transformation des diagrammes d'activité vers FoCaLiZe

Chapitre V: L'implémentation

Listes des figures

Liste des tableaux

Introduction générale

La maîtrise du logiciel passe par une spécification rigoureuse dès les premières étapes du cycle de vie. Malgré les progrès enregistrés dans ce domaine, nous continuons à souffrir du manque d'un cadre adéquat de spécification. D'un côté nous avons des langages à base de notations graphiques, tels qu'UML qui permettent de représenter des systèmes de manière synthétique et intuitive, mais auxquels il manque les bases formelles nécessaires pour faire de la vérification.

De l'autre côté, nous avons des langages formels, tels que FoCaLiZe, qui fournit des notations mathématiques permettant de spécifier très précisément les propriétés du système à construire. Le modèle est alors moins intuitif, mais peut être validé grâce à des techniques de preuve formelle.

Dans ce contexte, Plusieurs travaux se sont intéressés à l'application des techniques de preuves formelles sur un modèle UML. Parmi ces travaux, nous pouvons citer essentiellement ceux qui se sont intéressés à la traduction d'un modèle UML vers un langage formel tels que B [**AHM+07**], Alloy [**YOA15**], CSP [**HOU10**] et LOTOS [**LOT02**].

Cependant, la traduction d'UML vers ces langages impose généralement des restrictions fortes sur les constructions UML qu'il est possible d'utiliser. En effet, la plupart de ces travaux sont attachés aux diagrammes d'activités en isolation (modèles statiques) sans tenir compte de la communication entre eux.

L'environnement FoCaLiZe, est un atelier intégré de construction modulaire de logiciels certifiés, initiée à la fin des années 90 par Thérèse Hardin et Renaud Rioboo au sein des laboratoires LIP6, CEDRIC et INRIA. FoCaLiZe, issue de l'assistant d'aide à la preuve Coq, est un environnement complet de développement comportant un langage de programmation (fonctionnelle), un langage de spécification des besoins et un langage de preuve.

L'objectif visé dans cette mémoire consiste à proposer une démarche de transformation automatique des diagrammes d'activité d'UML en spécifications FoCaLiZe.

Pour cela, nous commençons par comprendre la transformation de diagramme d'une classe (composée des attributs et des opérations) telle quelle traitée dans les travaux [**Abb+14**, **Abb14**, **kha+15**]. Ensuite, nous proposons une démarche de transformation des diagrammes d'activité UML en spécifications FoCaLiZe, basée sur la transformation de classes citée ci-dessus. Pour atteindre cet objectif, nous avons opté pour une approche directe (transformationnelle), en définissant une règle de transformation pour chaque constituant de diagramme d'activité.

Enfin, pour mettre en œuvre notre démarche de transformation, nous utilisons le langage XSLT (Extensible Stylesheet Language Transformation). Nous avons défini une feuille de style XSLT, qui met en œuvre les règles de transformation à partir d'un document XMI (XML Meta data Interchange) et permettre la génération du code FoCaLiZe.

Cette thèse est composée de cinq chapitres :

Chapitre I: Il est consacré à l'étude des diagrammes d'activité UML. En particulier, nous détaillons les composants que nous supportons dans notre démarche de transformation.

Chapitre II: Ce chapitre présente les concepts de base de l'environnement FoCaLiZe.

Chapitre III: Dans ce chapitre, nous présenterons un état d'art des différentes approches de formalisation de modèles UML (diagramme d'activité).

Chapitre IV: Ce chapitre est le cœur de notre travail, il décrit notre démarche de transformation des diagrammes d'activité vers FoCaLiZe.

Chapitre V: Ce chapitre présente l'implémentation de notre démarche de transformation.

Chapitre I

Les diagrammes d'activité d'UML2.0

I.1 Introduction

Le langage de modélisation objet unifié UML (*Unified Modeling Language*), est né de la fusion des trois méthodes objet : OMT (*Object Modeling Technique*) de James Rumbaugh [**RBL+90**], OOSE (*Object Oriented Software Engineering*) d'Ivar Jacobson [**JCC94**], et la méthode de Grady Booch [**BOO91**]. Puis il est normalisé par l'OMG en 1997 dans sa version 1.1 comme langage de modélisation des systèmes d'information à objets. UML est rapidement devenu un standard incontournable [**HOU10**].

La figure ci-dessous [**MOU13**], présente l'évolution des versions d'UML.

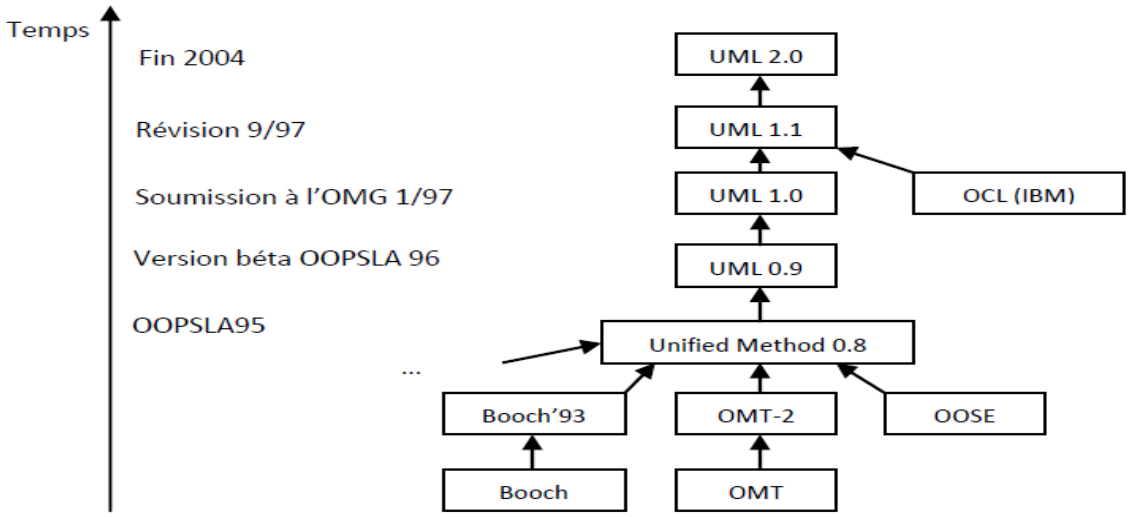


Figure I. 1: Evolution des versions d'UML

Au niveau d'UML, deux éléments importants sont à noter. Le terme *unified* et le terme *langage*. Le premier terme signifie que les auteurs ont essayé de regrouper les éléments importants des concepts objets, alors que le deuxième montre qu'il s'agit d'un langage de modélisation et non d'une méthode.

UML est un langage qui permet de modéliser non seulement des applications informatiques ou des structures de données, mais également les activités d'un domaine : mécanique, biologie, processus métier... [**Perochon09**].

Plus précisément, UML permet d'offrir des outils d'analyse, de conception et d'implémentation des systèmes logiciels, ainsi que pour la modélisation d'entreprise et des systèmes non logiciels [**OMG09**].

Le but de ce chapitre est de donner une description des diagrammes fondamentaux d'UML, mais nous nous intéresserons ici qu'aux diagrammes de classes et les diagrammes d'activité, qui sont les seuls pris en compte lors de la transformation d'UML vers FoCaLiZe.

I.2 Diagrammes UML

La notation UML est décrite sous forme d'un ensemble de diagrammes. La spécification de la version d'UML 2.0 définit 13 diagrammes regroupés dans deux classes principales qui sont **[HOU10] [MOU13]** :

1- Diagrammes structurels
2- Diagrammes comportementaux

I.2.1 Diagrammes structurels

Ces diagrammes permettent de visualiser, spécifier, construire et documenter l'aspect statique ou structurel du système informatisé.

- ***Diagramme de classes :*** Le but d'un diagramme de classes est d'exprimer de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes. Une classe a des attributs, des opérations et des relations avec d'autres classes.
- ***Diagramme d'objets :*** Le diagramme d'objets complète un diagramme de classes en le justifiant par des exemples d'utilisation. Il est par exemple utilisé pour vérifier l'adéquation d'un diagramme de classes à différents cas possibles.
- ***Diagramme de composants :*** Il montre les composants du système d'un point de vue physique, tels qu'ils sont mis en œuvre (fichiers, bibliothèques, bases de données...). Il montre la mise en œuvre physique des modèles de la vue logique avec l'environnement de développement.
- ***Diagramme de déploiement :*** Ce type de diagramme UML montre la disposition physique des matériels qui composent le système (ordinateurs, périphériques, réseaux...) et la répartition des composants sur ces matériels. Les ressources matérielles sont représentées sous forme de nœuds, connectés par un support de communication.

- ***Diagramme des paquetages :*** Un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML, il sert à représenter les dépendances entre paquetages.
- ***Diagramme de structure composite :*** Le diagramme de structure composite permet de décrire sous forme de boîte blanche les relations entre les composants d'une seule classe.

I.2.2 Diagrammes comportementaux

Les diagrammes comportementaux modélisent les aspects dynamiques du système. Ces aspects incluent les interactions entre le système et ses différents acteurs, ainsi que la façon dont les différents objets contenus dans le système communiquent entre eux.

- ***Diagramme des cas d'utilisation :*** Ce diagramme a pour objectif d'assurer le lien entre l'utilisateur et les objets du système. Ce diagramme représente la structure des grandes fonctionnalités nécessaires aux utilisateurs du système.
- ***Diagramme états-transitions :*** Le diagramme d'états-transitions représente l'évolution des objets appartenant à une même classe. La modélisation du cycle de vie est essentielle pour représenter la dynamique du système.
- ***Diagramme d'activité :*** Un diagramme d'activité est une variante des diagrammes d'états-transitions. Il permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation. Dans un diagramme d'activité les états correspondent à l'exécution d'actions ou d'activités et les transitions sont automatiques.
- ***Diagramme de séquence :*** Les diagrammes de séquence contiennent des informations concernant les messages échangés entre les entités du système dans le temps. La notion de temps est représentée dans le diagramme de séquence de manière explicite, sur des lignes verticales, s'écoulant du haut en bas pour montrer l'ordre chronologique des échanges.
- ***Diagramme de communication :*** C'est une représentation simplifiée d'un diagramme de séquence, en se concentrant sur les échanges de messages entre les objets.
- ***Diagramme global d'interaction*** *:* permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences (variante du diagramme d'activité).
- ***Diagramme de temps :*** Le diagramme de temps permet de décrire les variations d'une donnée au cours du temps.

I.3 Diagramme de classe

Les diagrammes de classe spécifient une collection d'éléments modélisant la structure d'un système en faisant abstraction des aspects dynamiques et temporels. C'est à la fois l'axe essentiel de la modélisation objet et la notation la plus riche de tous les diagrammes UML [**UML2B**].

I. 3.1 Classe

La classe est une description abstraite d'un ensemble d'objets qui partagent les mêmes propriétés (attributs) et comportements (opérations).

Une classe y est caractérisée par (voir figure I.2) :

- son nom
- la liste de ses attributs
- la liste de ses opérations (également appelées méthodes)

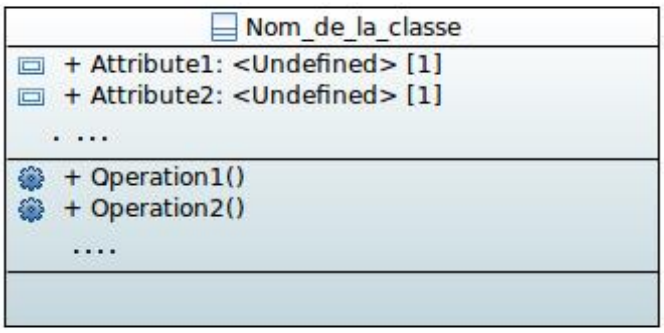


Figure I. 2: Représentation UML d'une classe

3.1.1 Attribut

Un attribut est déclaré au niveau d'une classe, éventuellement typée, à laquelle chacun des objets de cette classe donne une valeur. Un attribut peut posséder une multiplicité et une valeur initiale [**RUML**].

3.1.2 Opération

Une opération peut déclarer des paramètres (eux-mêmes typés) ainsi qu'un type de retour.

- Un type d'attribut ou de paramètre d'opération est :
  - Soit un type de donnée, primitif (entier, booléen, chaîne de caractères, ...) ou complexe (séquence, tuple, etc.) ;
  - Soit un type objet, dans le cas où il référence une autre classe du diagramme.

I.3.2 Relations entre les classes

Un diagramme de classe établit également des relations entre les classes. Celles-ci peuvent être [**UML2B**] :

- Une association : représentant le lien unissant les instances des classes, un lien étant une relation d'utilisation entre différents objets ;
- Une agrégation : qui est une forme d'association permettant d'exprimer une relation d'inclusion entre les classes (relation composant-composé) ;
- Une composition : c'est un cas particulier d'agrégation dans laquelle l'objet composé ne peut appartenir qu'à un seul objet composant (les objets composés sont créés et détruits en même temps que l'objet composant) ;
- La généralisation : On peut spécialiser une super-classe, en ajoutant à la sous-classe des attributs et des opérations (c'est l'équivalent du concept d'héritage en programmation orientée objet). On dit alors que la super-classe est une généralisation de la sous-classe.

➢ Chaque relation entre un couple de classes peut être dotée de rôles et de multiplicités. Un rôle sert à identifier de façon unique une extrémité d'association. Une multiplicité est associée à chaque extrémité pour indiquer, à l'aide d'un intervalle (0..1, 1..1, 0..* ou 1..*, * symbolisant l'infini), le nombre d'objets susceptibles de participer à une association donnée [**UML2B**].

I.4 Diagramme d'activité

I.4.1 Qu'est-ce que diagramme d'activité ?

UML permet de représenter graphiquement les aspects dynamiques des systèmes à l'aide de plusieurs diagrammes comportementaux. Parmi eux, on distingue les diagrammes d'activité qui permettent de mettre l'accent sur les traitements. Ils sont utilisés pour représenter le comportement d'une méthode ou le déroulement d'un cas d'utilisation [**DEV**].

Un diagramme d'activité est une variante des diagrammes d'états-transitions, dans lequel les états correspondent à l'exécution d'actions ou d'activités, et les transitions sont automatiques (les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre). Un diagramme d'activité peut être attaché à n'importe quel élément de

modélisation afin de visualiser, spécifier, construire ou documenter le comportement de cet élément.

Les diagrammes d'activité d'UML constituent un outil de modélisation des systèmes workflows, des modèles orientés service et des processus métiers (ils montrent l'enchaînement des activités qui concourent au processus). Un modèle d'activité consiste en activités liées par des flux de données et de contrôle. Une activité peut varier d'une tâche humaine à une tâche complètement automatisée [**HOU10**].

I.4.2 Intérêt des diagrammes d'activité

- Représente graphiquement le comportement interne d'une opération, d'une classe ou d'un cas d'utilisation sous forme d'une suite d'actions.
- Utilise le mécanisme de synchronisation pour représenter les successions d'états synchrones, alors que les diagrammes d'états-transitions sont utilisés principalement pour représenter les suites d'états asynchrones.
- Utilise des transitions automatiques évite la nécessité d'existence d'évènement de transition pour avoir un changement d'états.
- Modélise un workflow dans un cas d'utilisation, ou entre plusieurs cas d'utilisations.
- Définit avec précision les traitements qui ont cours au sein du système, Certains algorithmes ou calculs nécessitent de la part du modélisateur une description poussée.
- Spécifie une opération (décrire la logique d'une opération).
- Le diagramme d'activité est le plus approprié pour modéliser la dynamique d'une tâche ou d'un cas d'utilisation, lorsque le diagramme de classe n'est pas encore stabilisé [**AD**].

I.4.3 Composants de diagramme d'activité

Nous présentons dans cette section les éléments de diagramme d'activités qui nous intéressent dans notre démarche de transformation de UML vers FoCaLiZe, nous inspirent les référence [**GF10, UML08**]

4.3.1 Action

Une action est l'unité fondamentale de la spécification de comportement. L'action prend un ensemble d'entrées et les convertit en un ensemble de résultats, l'une ou les deux ensembles peuvent être vide. Une action ne peut être décomposée en actions plus simples.

Une action peut être, par exemple :

- une affectation de valeur à des attributs ;
- la création d'un nouvel objet ou lien ;
- l'émission d'un signal ;
- la réception d'un signal ;
- etc.

La notation d'une action est un rectangle avec des coins arrondis (voir figure 1.3).

Figure I. 3: Notation d'action

4.3.2 Transition et flot de contrôle

Dès qu'une action est achevée, une transition automatique est déclenchée vers l'action suivante (voir figure I.4). Il n'y a donc pas d'événement associé à la transition.

L'enchaînement des actions constitue le flot de contrôle.

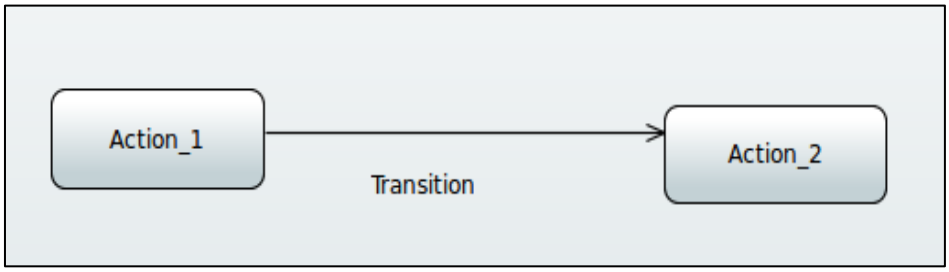


Figure I. 4: Formalisme de base du diagramme d'activité : actions et transition

4.3.3 Activité

Une activité contient des séquences d'actions et / ou d'autres activités. On utilise les activités pour grouper des séquences d'actions ensemble (voir figure I.5).

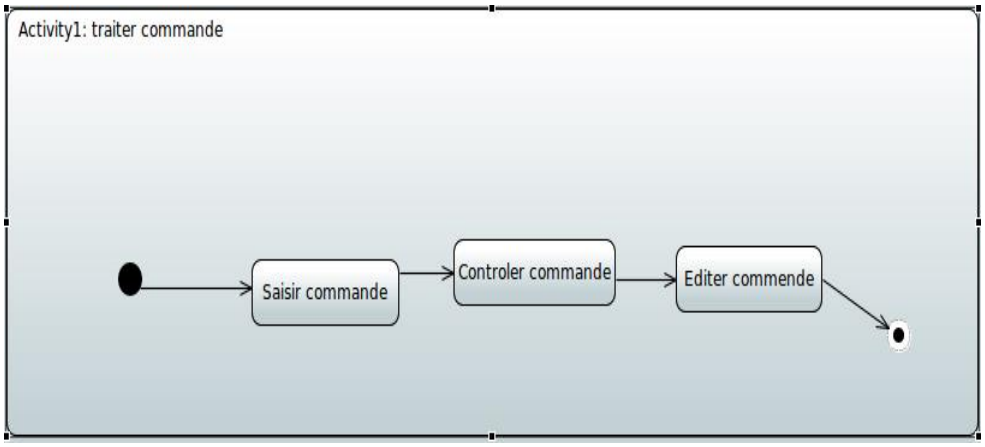


Figure I. 5: Exemple de représentation d'une activité

4.3.4 Nœud de contrôle

On utilise les nœuds de contrôle pour guider le flux de contrôle (et le flux d'objets) à travers un groupe d'activités et d'actions. Les nœuds de contrôle viennent dans une variété de formes, en fonction de ce qu'on a besoin, ils servent comme un agent de circulation pour le flux de contrôle et les flux d'objets.

Les nœuds de contrôle sont les suivants :

A. Initial : Un nœud initial est l'endroit où le flux de contrôle commence quand une activité est invoquée. La figure I.6 montre la notation d'un nœud initial.

Figure I. 6: Notation de nœud initial

B. Final : Un nœud final est un nœud de contrôle dans laquelle un ou plusieurs flux au sein d'une activité donnée s'arrêtent. La figure I.7 montre la notation d'un nœud final.

Figure I. 7: Notation de nœud final

C. **Décision** : Branche conditionnelle dans un flux .Possède une entrée et plusieurs sorties. Un jeton entrant émerge d'une seule des sorties ***[MICRO]***. La figure I.8montre sa notation.

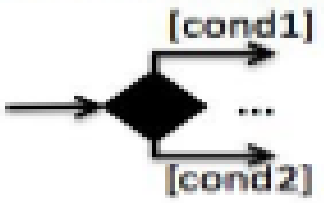


Figure I. 8: Notation de nœud de décision

D. **Fusion :** Nécessaire pour fusionner des flux qui ont été fractionnés avec un nœud de décision. Possède plusieurs entrées et une sortie. Un jeton sur une entrée quelconque émerge sur la sortie [**MICRO**].La figure I.9 montre sa notation.

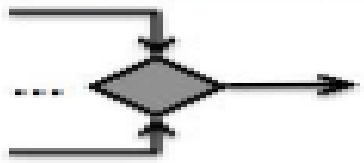


Figure I. 9: Notation de nœud de fusion

E. **Bifurcation** : permet à partir d'un flot unique entrant de créer plusieurs flots concurrents en sortie de la barre de synchronisation. La figure I.10 montre sa notation.

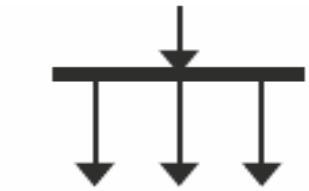


Figure I. 10: Notation de nœud de bifurcation

F. **Union (jonction)** : permet à partir de plusieurs flots concurrents en entrée de la synchronisation, de produire un flot unique sortant. Le nœud de jonction est le symétrique du nœud de bifurcation (voir figure I.11).

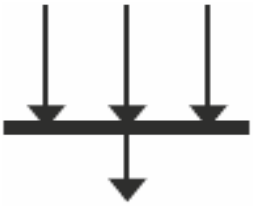


Figure I. 11: Notation de nœud d'union

La figure 1.12 illustre l'utilisation des notations des nœuds de contrôle, Ce diagramme décrit la gestion d'une commande vente [**DEV**].

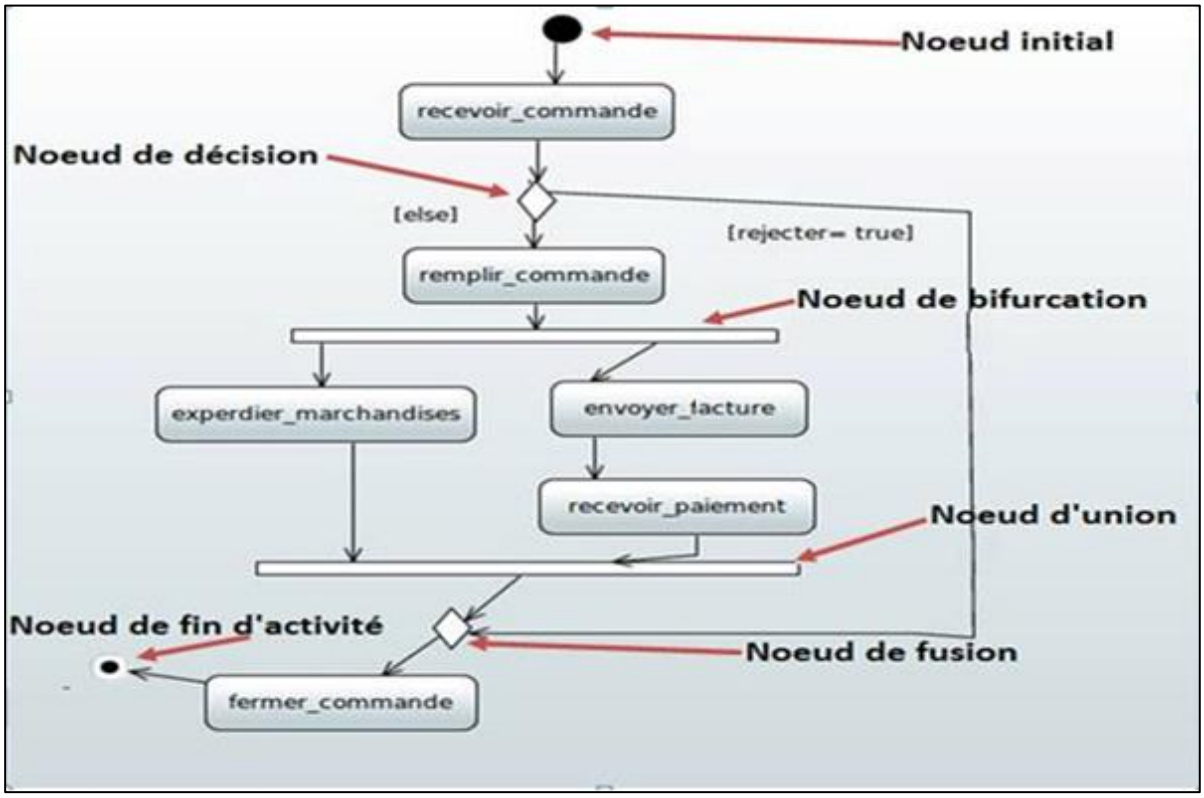


Figure I. 12: Exemple de nœuds de contrôle

4.3.5 Partition

Les activités peuvent impliquer de différents acteurs, tels que les différents groupes dans une organisation ou un système. Pour cela, on utilise des partitions pour montrer quel participant est responsable pour quelles actions.

Les Partitions divisent le diagramme en colonnes et/ou en lignes et contiennent des actions qui sont menées par le groupe responsable. Les partitions sont parfois dénommés *swimlanes.*

La figure 1.13 illustre la notation de partition [**GF10**].

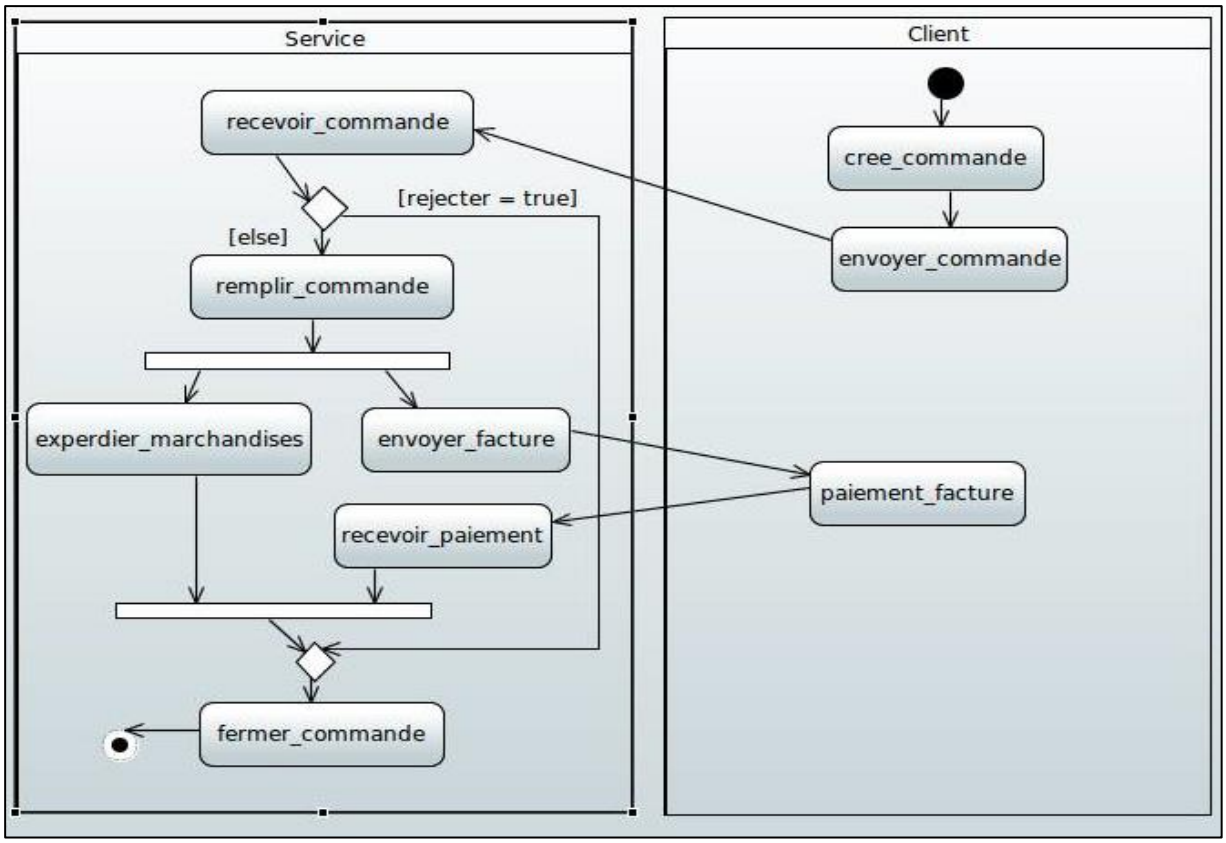


Figure I. 13: Exemple de diagramme d'activité avec partitions

I.5 Conclusion

Dans ce chapitre, nous avons présenté brièvement le langage de modélisation unifié UML, son évolution et ses diagrammes. Ensuite, nous avons donné une description brièvement les diagrammes de classe et les diagrammes d'activité d'UML 2.0 qui constituent un outil de modélisation des systèmes workflows, des modèles orientés service et des processus métiers.

Nous avons insisté sur les nœuds d'activité et les transitions, les nœuds de contrôle, la partition qui seront modélisés dans le formalisme source de la transformation constituant l'objet de notre travail.

Chapitre II

L'environnement FoCaLiZe

II.1 Introduction

Au cours des dernières décennies, plusieurs méthodes formelles ont vu les jours. Citons entre autres la méthode B [**Abr96**], Alloy [**Jac04**], le système Maude [**CDE+07**]...etc.

Ces méthodes formelles sont utilisées avec succès, à titre d'exemple, on trouve l'utilisation de la méthode B pour la protection automatique de système des trains en France et l'utilisation de system Maude pour l'analyse et la vérification des systèmes distribués et des protocoles de communications .

Dans le cadre de notre thèse, la méthode formelle FoCaLiZe a été initie à la fin des années 90 par Thérèse Hardin et Renaud Rioboo au sein de laboratoires LIP6, CEDRIC et INRIA [**PH11**]. FoCaLiZe issue de l'outil d'aide à la preuve Coq, est un environnement complet de développement composé d'un langage de programmation, un langage de spécification des besoins et un langage de preuve. Dans la suite, Nous allons expliquer cet environnement brièvement.

Dans la suite de notre représentation de l'environnement FoCaLiZe, nous inspirons les informations et les définitions posée de : [**DAM11**, **PHI11**], **PIE13**, **DAV10**] …etc.

II.2 FoCaLiZe en bref

FoCaLiZe est un environnement de programmation orientée objet qui combine les spécifications, les programmes et les preuves dans la même langue [**DAM11**]. Il est développé depuis 1997 au sein le projet éponyme [**PHI11**]. Dans ce mémoire, un nouveau langage a été conçu dans lequel il est possible de construire des applications pas à pas, partant de spécifications abstraites, appelées espèces, vers des implantations concrètes, appelées collections. Il permet également d'énoncer des propriétés et de prouver que les programmes respectent leurs spécifications. Ces différentes structures sont combinées en utilisant l'héritage et la paramétrisation, inspirés de la programmation orientée objet [**PIE13**, **DAV10**].

Dans 2003, V. Prevosto a développé un compilateur pour ce langage, capable de produire du code OCaml[1] pour l'exécution, du code Coq pour la certification, mais aussi du code pour la documentation. Ce compilateur a récemment été réécrit par F. Pessaux dans le cadre du projet SSURF [**DAV10**].

En 2007, D. Doligez fourni un outil de déduction automatique basée sur la logique de premier ordre, appelé Zenon. Cet outil de preuve automatique peut produire des preuves Coq[2], qui peuvent être réinsérées dans les spécifications Coq générées par le compilateur FoCaLiZe et vérifiés intégralement par Coq [**PIE13**].

Dans la suite de notre chapitre, nous allons détailler les différents aspects de FoCaLiZe. Nous décrivons d'abord l'aspect spécification, puis l'aspect implémentation.

II.3 Spécification (espèce)

La structure principale du langage FoCaLiZe est l'espèce, qui correspond au niveau le plus élevé d'abstraction dans une spécification. Une espèce peut être globalement vue comme une liste des méthodes: la représentation, les fonctions et les propriétés (contraintes).

La syntaxe générale d'une espèce est la suivante :

```
species species_name[(P1 is species_name1, p2 is species_name2, …)]  =
 [ inherit species_name1 ,species_name2,…;]
 [representation = rep_type;]                                  (*   representation   *)
 signature function_name : function type;                      (* fonction déclarée *)
 [local/logical] let [rec] function_name = function_body;      (*   fonction définie  *)
 property property_name : property_specification;              ( * propriété déclarée *)
 theorem theorem_name : theorem_specification                  (*   propriété définie *)
 proof = theorem_proof ;
end ;;
```

Table II. 1: Syntaxe générale d'une espèce

[1] **OCaml** est un langage fonctionnel augmenté de fonctionnalités permettant la programmation impérative. OCaml étend les possibilités du langage en permettant la programmation orientée objet et la programmation modulaire. Pour toutes ces raisons, OCaml entre dans la catégorie des langages multi-paradigme.

[2] **Coq** est un assistant de preuve utilisant le langage Gallina, fondé sur le calcul des constructions, une théorie des types d'ordre supérieur, et son langage de spécification est donc une forme de lambda-calcul typé

II.2.1 Représentation

La représentation d'une espèce indique le type de données des valeurs manipulées dans l'espèce. La définition de la représentation peut être reportée dans une espèce, ce qui signifie que la structure de donnée réelle encapsulée par l'espèce n'a pas besoin d'être connue à ce point de raffinement du développement. Dans ce cas, elle est représentée par une variable de type.

Par exemple on peut modéliser un point du plan cartésien représenté par ses cordonnées (latitude et longitude) sur les deux axes (l'axe x et l'axe y). La representation de cette espèce définit le type de ses entités par un couple de réels. Comme suit :

```
species point =
representation = float * float ;
…
end;;
```

Le type support peut simplement être le n-uplet regroupant toutes ces variables qui étaient disséminées dans l'objet. C'est celle qui a été vu dans l'exemple précédant nous avons utilisé le produit (***float * float***) pour définir la représentation de l'espèce point. Ou bien un type structuré "*Record type*" composé de plusieurs champs de types différents par exemple :

```
Type cordoonees = { lat : float; long : float ; } ;;
species point =
representation = cordoonees;
…
end;;
```

II.2.2 Fonctions

Comme les opérations d'une classe, les fonctions d'une espèce servent à manipuler ses entités (instances). Une fonction possède un identifiant unique (dans le contexte de l'espèce), un type et un corps calculatoire.

Pour permettre un niveau de spécification abstraite, une fonction peut être uniquement déclarée (statut abstrait). Dans ce cas, elle est spécifiée par le mot clé **signature**. Une fonction complètement définie (statut concret) est introduite par le mot clé **let.**

3.2.1 Fonction déclarée (signature)

La signature permet d'annoncer une fonction qui sera définie ultérieurement. Une fois déclaré, le nom de la fonction peut être utilisé pour définir d'autres méthodes (corps de fonction, propriétés et preuves). Cela permet de retarder le choix de l'implémentation de cette fonction. Donc, ces méthodes introduites par signature sont particulièrement utiles pour définir la spécification ou l'architecture.

Par exemple on peut représenter une signature de la fonction **egal** qui indique l'égalité de deux points, comme suit :

```
species point =
representation = float * float ;
signature egal : Self -> Self -> bool ;
…
end;;
```

3.2.2 Fonction définie (let)

Les fonctions définies attribuent par le mot-clé **let**, suivi d'un nom, d'un type facultatif, et d'une expression. Elles servent à présenter des constantes ou des fonctions, c'est-à-dire opérations informatiques. Le langage de base utilisé pour les implémenter est très proche des expressions à la **ML** (conditionnelle, filtrage où "pattern matching " est fait par "**match …with**", …) étendu d'une construction permettant d'appeler une méthode d'une espèce donnée [**PHI11**]. Des définitions mutuellement récursives sont présentées par **let rec**.

Dans notre exemple précédant, on continue par définition une opération "***different***" pour décider si deux points sont différentes ou non, comme suit :

```
species point =
…
let different (x:Self, y:Self) : bool = ~~( egal(x, y)) ;
…
end;;
```

La définition de la fonction "***different***" utilise la fonction **egal** qui déjà définie dans le contexte de l'espèce et la fonction "~~" qui désigne la négation booléenne (prédéfinie dans la bibliothèque standard de FoCaLiZe).

II.2.3 Propriétés

Les propriétés doivent être vérifiées par toute implantation de l'espèce, les propriétés peuvent être soit simplement déclaré des propriétés (quand seul l'énoncé est donné), soit des théorèmes (quand la preuve est également fournie) [**DAV10**].

3.3.1 Propriété déclarée (property)

Une propriété est un introduite par le mot-clé "**property**" suivie d'un nom et d'une formule de premier ordre. Par conséquent les propriétés servent à exprimer des conditions. C'est à dire, pour des spécifications.

Les formules utilisent les connecteurs logiques habituels, les quantificateurs universel (∀) et existentiels (∃) sur un type FoCaLiZe et des noms de entités connues dans le contexte de l'espèce.

Par exemple, nous introduisons les trois propriétés suivantes pour définir une structure respectant une relation d'équivalence sur l'opération ***egal*** de l'espèce point, comme suit :

```
species point =
…
property egal_refl : all x: Self, egal(x, x);
property egal_sym : all x y : Self, egal (x, y) ->egal (y, x);
property egal_tran : all x y z : Self, egal (x, y) /\ egal (y, z) ->egal (x, z);
end;;
```

3.3.2 Propriété déclarée (theorem)

Un théorème (theorem) est une propriété accompagnée de la preuve formelle que son énoncé est vérifié dans le contexte de l'espèce. La preuve accompagnant l'énoncé va être traitée par FoCaLiZe et ultimement vérifiée par le prouveur Coq.

Dans notre espèce point, le théorème ***non_different*** (**a = b ⇒¬(a≠b)**) qui met en cause les opérations ***egal*** et "***different***" est présenté comme suit :

```
species point =
…
theorem non_different : all x, y:Self, egal (x, y) -> ~(different(x, y))
proof = by definition of different;
end;;
```

3.3.3 Preuves

FoCaLiZe distingue entre la preuve des propriétés déclarées et des théorèmes :

- la preuve d'un théorème peut employer des propriétés disponibles dans le contexte de l'espèce. Elle doit accompagner son énoncé par écriture :(**Proof = "la preuve appropriée"**). Comme il est indiqué dans l'exemple précédent.

- La preuve des propriétés en fait sont retardées. En dépit du ce retarde, comme pour des signatures, des propriétés peut être employé pour prouver des théorèmes. Elle est présentée par le mot-clé proof suivi du nom d'une propriété précédemment présentée.

Par exemple la preuve de la propriété " ***egal_sym***" qui est déclarée précédemment avec les propriétés de l'espèce point (***egal_refl***, ***egal_sym***, ***egal_tran***) est représentée comme suit:

```
species point =
proof of egal_sym = assumed ;
…
end ;;
```

Où "**assumed**" est un mot clé utilisé pour admettre un théorème sans donner sa preuve (comme un axiome). Il peut être utilisé aussi pour accepter une propriété bien connue des mathématiques ou de la logique.

II.3 Combinaison des espèces

II.3.1 Héritage

Les concepts présentés ci-dessus (représentation, fonctions et propriétés) permettent de créer des espèces entièrement nouvelles. Le mécanisme d'héritage permet de créer une nouvelle espèce en utilisant des espèces existantes. La nouvelle espèce crée par héritage acquière toutes les fonctions et les propriétés (y compris la représentation) des ses super-espèce (parentes).

L'héritage permet aussi de faire la nouvelle espèce plus concrète, en fournissant des corps calculatoires aux signatures héritées, et des preuves formelles aux propriétés héritées. De cette manière, l'héritage permet un raffinement successif des espèces en concrétisant au fur et à mesure leurs méthodes. De plus la nouvelle espèce peut apporter de fonctions et propriétés

propre à lui par rapport à ses super-espèces, comme il peut redéfinir des fonctions et propriétés bien quelles étaient déjà définies au niveau des super-espèces (voir la syntaxe dans les premiers lignes de la table II.1)

À titre d'exemple, l'espèce ***PointConcret*** (voir le code suivant) est créée par l'héritage de l'espèce ***PointAbstrait***. Dans cet exemple, l'héritage est uniquement utilisé pour affiner l'espèce ***PointAbstrait***. L'espèce ***PointConcret*** fournie des définitions concrètes pour la représentation et pour les fonctions ***get_x***, ***get_y*** et ***deplacer*** qui étaient uniquement déclarées. Elle introduit la fonction ***creerPointConcret*** pour servir en tant que constructeur de l'espèce. La fonction distance (calcule la distance entre deux points) est hérité telle quelle.

```
species PointAbstrait =
signature get_x : Self -> float;
signature get_y : Self -> float;
signature deplacer : Self -> float-> float -> Self;
let distance( p1:Self, p2: self):float= #sqrt ( ((get_x(p1) – get_x(p2))* (get_x(p1) –
get_x(p2))) +((get_y(p1) – get_y(p2))* (get_y(p1) – get_y(p2)));
end;;

species PointConcret = inherit PointAbstrait;
representation = float * flaot;
let creerPointConcret(x:float, y:floatà: Self= (x, y);
let get_x(p) = fst(p);
let get_y(p) = snd(p);
let deplacer(p, x ,y ) = (get_x(p) + x, get_y(p) + y) ;
end ;;
```

II.3.2 Collection

Une espèce est dite *complète* si toutes ses déclarations ont reçu une définition et toutes ses propriétés sont prouvées. La définition d'une espèce peut être soumise à un processus d'abstraction de sa représentation pour créer une collection. Une collection est donc un ensemble de méthodes partageant une même structure de donnée qui est masquée. La syntaxe d'une collection est la suivante :

collection <*collection_ name*> = implement <*species_ name*> ; end;;

Par exemple, pour bien comprendre de collection il faut définit tous les déclarations des fonctions de l'espèce point et prouvé tous ses propriétés. Puis on utilise notre espèce complète point par créer de collection comme suit :

```
species point =
representation = float * float ;
let first(x:Self):float= fst(x);
let second(x:Self):float= snd(x);
let egal(a: float, b:float):bool= (first(a)=first(b))&(second(a)=second(b));
Proof of egal_refl = assumed;
Proof of egal_tran=assumed;
end;;
collection Cord = implement point;
end;;
```

II.3.3 Paramétrage

En plus de l'héritage, une seconde mécanisme de combinaison les espèces consiste à utiliser la paramétrisation. Les espèces peuvent être paramétrées soit par des collections soit par des entités provenant de collections. Si le paramètre est une collection, l'espèce paramétrée a seulement accès à l'interface de cette collection, c'est-à-dire seulement sa représentation abstraite, ses déclarations et ses propriétés. Quand c'est un paramètre d'entité, le mot-clé "***in***" est utilisé. Une espèce Cercle (qui définit par son centre (un point) et son diamètre) peut être créée en utilisant l'espèce point comme paramètre comme suit :

```
species Circle (P is point) =
representation  = P * float ;
let newCircle ( centre: P,  rayon: float ): Self = (centre,  rayon);
let getCenter (c:Self):P  =  fst(c);
let getRadius(c:Self):float = snd(c);
let belongs(p:P, c:Self):bool = (P!distance(p, getCenter(c)) =getRadius(c));
 theorem belongs_specification : all c:Self, all p:P, belongs(p, c) <-> (P!distance(p,
getCenter(c)) = getRadius(c))
```

```
proof = ... ;
end;
```

La notation "**!**" est équivalente à la notation "**.**" d'envoi de message en programmation orientée objet.

II.4 Compilation et mise en œuvre

La compilation d'une source FoCaLiZe sert à générer deux codes cibles : un code OCaml et un code Coq. Le code OCaml représente l'exécutable du programme, et le code Coq sert à vérifier et valider toutes les propriétés et les preuves du modèle. Le code Coq est entièrement vérifié par la machine. Lors de génération du code OCaml, toutes les propriétés logiques (non calculatoires) sont cachées, car elles n'ont pas d'exécutables.

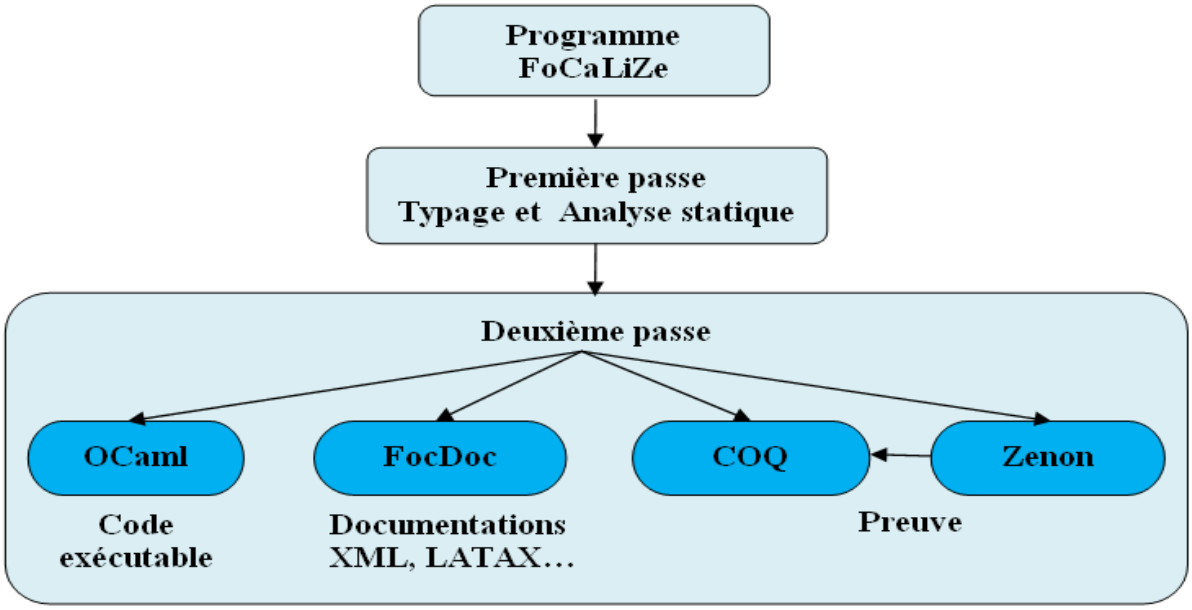


Figure II. 1: Compilation de FoCALiZe

La compilation d'une source FoCaLiZe est réalisée en invoquant la commande focalizec comme suit : **focalizec *source*.fcl**

Tel que ***source*.fcl** désigne le nom d'un fichier contenant une source FoCaLiZe. Quand nous obtiendrons ces messages, la compilation, la génération de code exécutable et la vérification sont faite avec succès :

```
Invoking ocamlc...
>>ocamlc -I /usr/local/lib/focalize -c point.ml
Invoking zvtov...
>>zvtov -zenon /usr/local/bin/zenon -new point.zv
Invoking coqc...
>>coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenonpoint.v
```

Pour plus détails voir le tutoriel et le manuel de référence de FoCaLiZe[3].

II.5 Conclusion

L'atelier FoCaLiZe est un environnement de développement de logiciels sûrs. C'est un outil d'automatisation de la mise en place des méthodes formelles pour spécifier formellement des besoins, implanter des programmes correspondant à ces besoins et exprimer des propriétés et prouver des théorèmes que ces programmes sont correctes vis-à-vis les spécifications définies précédemment.

Dans ce chapitre, nous avons présenté les aspects de l'environnement FoCaLiZe avec des exemples de développement concret pour chaque concept.

Nous avons expliqué aussi, les mécanismes de combinaison entre les espèces FoCaLiZe et la compilation d'une source FoCaLiZe.

Dans le chapitre suivant, nous allons rappeler la formalisation des diagrammes d'activités dans les recherches existantes.

3 **l'url** : http://focalize.inria.fr/

Chapitre III

Formalisation des diagrammes d'activité

III.1 Introduction

Actuellement, UML est l'un des langages de modélisation les plus répandus pour la conception des applications informatiques. UML étant un langage semi-formel, le principal reproche qui peut lui être fait réside dans l'absence de bases formelles permettant l'application des techniques de vérifications formelles. Nous choisissons des diagrammes d'activités qu'ils constituent un outil de modélisation des systèmes workflows, des modèles orientés service et des processus métiers.

Dans ce contexte, de nombreux travaux ont été réalisés afin de doter les modèles UML de bases formelles et d'outils de vérifications. La technique la plus anciennement utilisée pour la vérification des propriétés de diagramme d'activité est les réseaux de Pétri [**GV13**].

Cependant, l'approche la plus largement adoptée est la transformation d'un modèle UML vers une spécification formelle, en utilisant des méthodes formelles comme B [**AHM$^+$07**], Alloy [**YOA15**], CSP [**HOU10**], LOTOS [**LOT02**], etc. Cette dernière approche rentre dans le cadre de l'ingénierie de modèles MDE ("Model Driven Engineering"), qui vise la production de softwares par raffinements automatiques de modèles, depuis les spécifications abstraites jusqu'aux implémentations concrètes.

Dans ce chapitre, nous présenterons un état détaillé des différentes approches de formalisation de modèles UML diagramme d'activité. Nous commencerons par les techniques de transformation du modèle semi-formel vers langage formel, c'est la base de toute formalisation d'UML vers le langage formel, en indiquant les principes de notre approche.

Dans le reste du chapitre, nous allons présenter brièvement les principaux travaux de formalisation du diagramme d'activité UML.

III.2 Technique de transformation du modèle semi-formel vers langage formel

UML est un langage de modélisation objet possédant une popularité incomparable à la fois dans le monde industriel qu'académique. Mais, il demeure néanmoins une méthode semi-formel parce qu'il n'offre pas la possibilité pour une vérification et preuve formelles. C'est la raison pour laquelle nous avons jugé qu'une utilisation conjointe d'une notation semi-formelle et d'un langage formel.

Cette dernière génération de transformation a fait l'objet de plusieurs études de recherches [**SOP00**, **HIB13**, **KAR14**, **DEHIMI**].

Nous présenterons la technique de transformation du modèle semi-formel vers le langage formel. Les contributions existantes liées à cette solution peuvent être séparées principalement, comme le montre la figure III.1, en deux axes de recherches. Nous citerons les approches de traduction de spécification semi-formelle vers formelle et les approches de transformation du modèle.

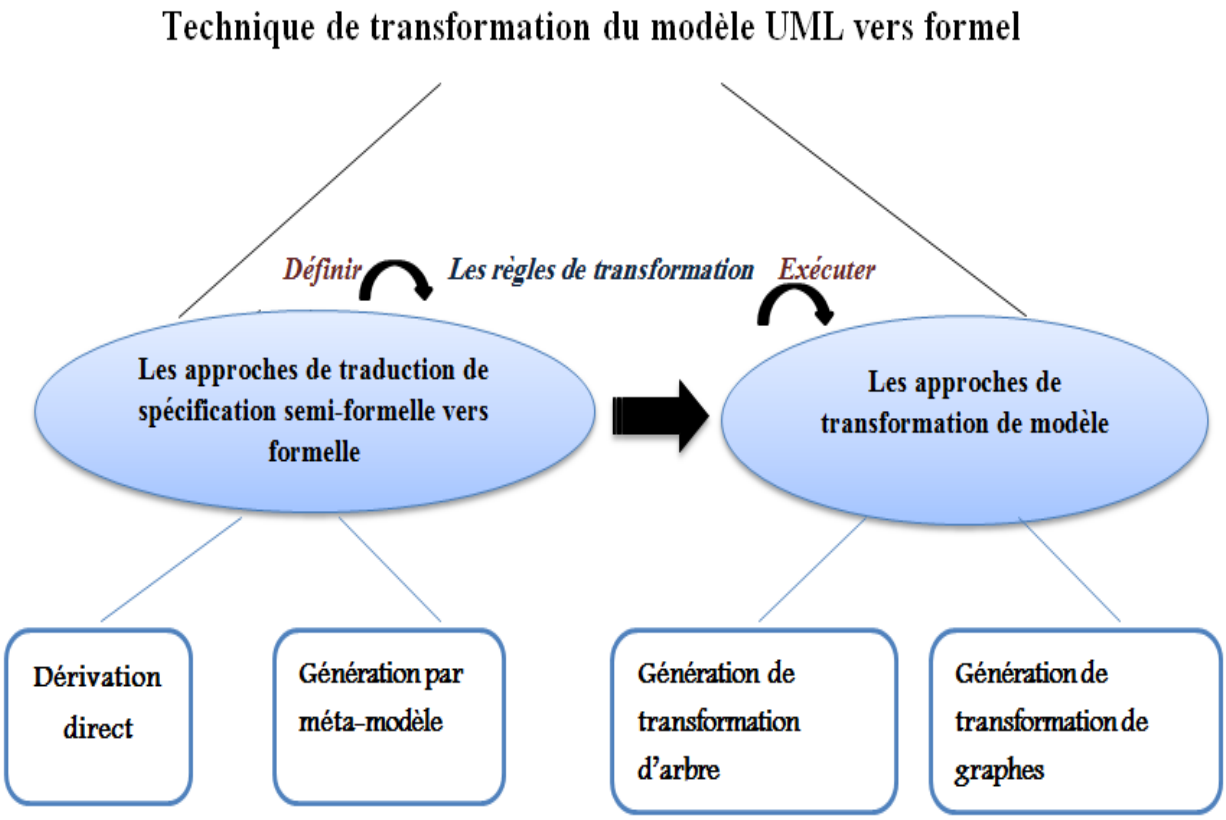


Figure III. 1:Technique de transformation du modèle UML vers formel

III.2.1 Les approches de traduction de spécification semi-formelle vers formelle

La traduction de spécification semi-formelle vers de spécification formelle comprend la traduction implicite ou explicite des concepts du modèle et celle de concepts du domaine. Elle peut donc s'effectuer selon deux démarches **[SAN10**, **SOP00**]:

2.1.1 Dérivation direct

Pour effectuer la translation, une correspondance est établie entre chaque élément du modèle UML et les éléments construisant le langage formel. Il y'a plusieurs travaux existant utilise cette démarche **[AHM+11, AHM+07, JAN+14].**

2.1.2 Génération par méta-modèle

Le principe de cette démarche sert à formaliser le méta-modèle du modèle semi-formel(UML) à travers un langage formel. Il y'a plusieurs travaux existant utilise cette démarche [**AHM$^+$14**, **HOU10**, **HARALD**].

Ces deux démarches de traduction différente dans leurs objectif .Nous ont appliqué une démarche de dérivation directe puisque notre objectif est de partir d'un modèle source UML de la spécification pour arriver à un modèle cible formel. Cette approche nous parait la plus pragmatique et la plus simple à mettre en œuvre. Le chapitre suivant illustre cette démarche.

Dans ce qui suit, nous présenterons de manière brève un état d'art les classifications d'approche de transformation de modèles UML/OCL.

III.2.2 Des approches de transformation de modèles

La transformation de modèles peut être classée en deux générations, en fonction de la structure de données utilisée pour décrire le modèle [**Sana10**] :

2.2.1 Génération de transformation d'arbre

Cette génération permet de parcourir un arbre d'entrée et de générer au cours de ce parcours les fragments de l'arbre de sortie. Généralement, ces méthodes utilisent des documents au format XML (eXtensible Markup Langage) et les outils adopté ce format comme Papyrus et TOPCASED, et les travaux utilise ces outils comme [**HARALD**, **Kha$^+$15].**

2.2.2 Génération de transformation de graphes

Cette génération permet de transformer un modèle d'entrée sous le format de graphes orienté étiqueté en un modèle de sortie et les outils adopté ce format comme ATOM3, et les travaux utilise cet outil comme [**DEHIMI**, **HOU10, HIB13**].

Ces deux démarches de transformation différente dans leurs objectif .Nous ont appliqué une démarche de génération de transformation d'arbre puisque notre objectif est de partir d'un modèle source UML de la spécification qu'il exprime en XMI (le standard XML de

modèle UML) pour arriver à un modèle cible formel. Chapitre V illustre cette démarche d'implémentation.

Dans ce qui suit, nous détaillons certain travaux existants pour formalisation de diagramme d'activité.

III.3 Travaux de formalisation de diagramme d'activité

Dans la littérature, la formalisation de diagramme d'activité d'UML est utilisée pour modéliser les systèmes workflow, les systèmes orientés services et les processus métiers. L'objectif de cette formalisation est de préciser une sémantique non ambiguë permettant de vérifier formellement la dynamique représentée par des diagrammes d'activité UML.

Dans cette section nous allons présenter l'état de l'art de certaines approches utilisant cette formalise et quelques exemples de travaux existent dans chaque approche.

III.3.1 Approche basées sur la méthode B

La méthode B est un langage de programmation formelle basé sur la théorie des ensembles, elle couvre toutes les étapes de développement de logiciels, de la spécification abstraite jusqu'à la génération de code exécutable. L'événement B est une variante de la méthode B proposée par Abrial pour traiter distribués, des systèmes parallèles réactifs [**AHM$^+$07**].

Dans cette approche nous avons trouvé quelques articles:

- Dans ces articles [**AHM$^+$07, AHM$^+$11**], ils ont concentré sur la traduction des diagrammes d'activité dans l'événement B, afin de vérifier les propriétés du système workflow et des applications distribuées et parallèles avec le prouver B. Ils ont présenté les règles de traduction des diagrammes d'activité dans l'événement B, et la relation entre la décomposition hiérarchique des activités dans les diagrammes d'activité d'UML. Ces travaux adoptent une approche dérivation direct, basée sur un ensemble de règles de traduction.

III.3.2 Approche basées sur le langage Alloy

Alloy [**SAG10**] est un langage formel de modélisation basé sur la logique relationnelle, utilisant un style de conception modulaire inspiré du paradigme orienté objet. Un module Alloy regroupant des spécifications (signatures) permet d'exprimer des contraintes structurelles et dynamiques sur les modèles. En plus, Alloy est supporté par une infrastructure logicielle (l'Analyseur d'Alloy 4), qui permet d'analyser et de vérifier d'une manière systématique les propriétés d'un modèle Alloy.

De nombreux travaux sont axés sur la transformation de l'aspect structurel d'UML vers Alloy. Nous trouvons d'abord, une étude exhaustive sur la transformation d'UML/OCL vers Alloy [**ABGR07**]. Cette étude a donné naissance à un prototype (appelé UML2Alloy) de transformation automatique des modèles UML/OCL en spécifications Alloy. L'aspect dynamique (diagrammes d'états-transitions d'UML) est aussi formalisé par Alloy [**GPCR12**] pour simuler et vérifier la consistance entre les aspects statique et dynamique, et les contraintes OCL.

Extension de ces travaux afin de vérifier automatiquement des modèles de procédés en utilisant les diagrammes d'activités UML [**YOA15**] pour la modélisation de procédés. L'intérêt principal de cette formalisation est de donner une sémantique mathématique claire et non ambiguë basée directement sur les concepts et la sémantique d'UML AD plutôt que de se reposer sur un formalisme tel que les réseaux de Petri. Afin de pouvoir raisonner sur chacune des perspectives du procédé ; la formalisation couvre à la fois la perspective de flux de contrôle et des données à travers l'utilisation de la notation des diagrammes d'activités.

Dans ce travail, ils ont défini un framework pour la vérification de procédé appelé Alloy4PV. Il utilise un sous-ensemble des diagrammes d'activités UML 2 comme langage de modélisation. Afin d'effectuer la vérification de procédés, ils ont défini un modèle formel des diagrammes d'activités basé sur la sémantique UML (le standard de l'OMG donnant une sémantique à un sous-ensemble d' UML) en utilisant la logique de premier ordre, pour mettre en œuvre cette formalisation en utilisant le langage Alloy afin d'effectuer du model-checking borné, et automatisé, dans un outil graphique intégré à Eclipse, la possibilité d'exprimer et de vérifier des propriétés sur toutes les perspectives du procédé et le figure suivant illustre cette approche [**YOA15**].

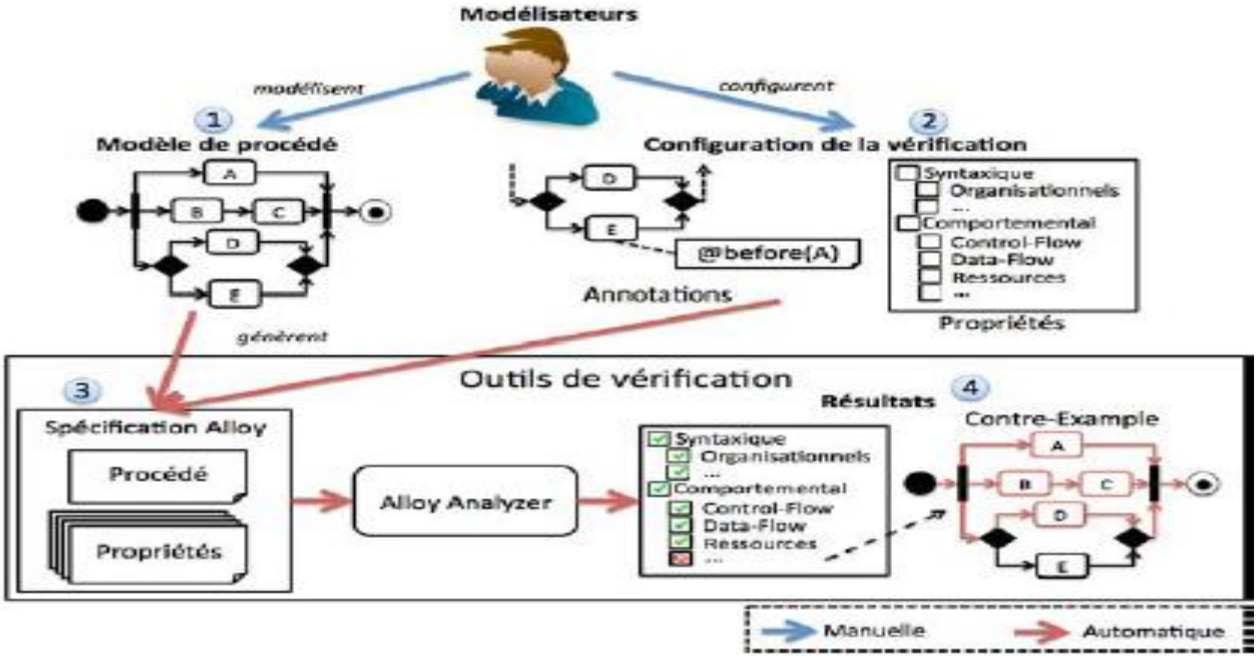


Figure III. 2: Vue globale de l'approche pour vérifier un procédé

III.3.3 Approches basées sur les algèbres de processus

C'est dans cette approche, qu'une représentation explicite des processus concurrents est autorisée. En effet, les contraintes sur toutes les communications autorisées et observables entre processus, permettent la représentation du comportement du système.

Nous distinguons deux langages dans cette approche CSP [**HOU10**], LOTOS [**LOT02**].

3.3.1 CSP (Communicating Sequential Processes)

CSP [**HOU10**] est un langage formel pour décrire les modèles d'interaction dans les systèmes concurrents. C'est un membre de la famille des théories mathématiques de concurrence nommé algèbre de processus ou calcul de processus. CSP a été proposé en 1978 par C. A. R. Hoare, après, il a des applications dans l'industrie comme outil de spécification et de vérification des aspects concurrents de différents systèmes.

Dans cette sous-catégorie on trouve un travail utilisé ce langage :

- L'objectif visé dans ce mémoire [**HOU10**] et de proposer une approche automatique de transformation des diagrammes d'activité d'UML vers CSP, basée sur la transformation de graphes, et réalisée à l'aide de l'outil ATOM³. Afin de réaliser cette méthode, ils ont proposé un méta-modèle des diagrammes d'activités et une

grammaire de graphe c'est-à-dire utilisé l'approche génération méta-modèle. Le méta-modèle permet de générer un outil visuel de modélisation des diagrammes d'activités, et la grammaire de graphe transforme ces derniers en code CSP équivalent. La vérification de programmes CSP est réalisée par le model-checker FDR (Failures-Divergences Refinement) pour vérifier automatiquement certaines propriétés comportementales .les propriétés sont exprimées sur le flux de contrôle seulement dans ce travail.

3.3.2 LOTOS (Language Of Temporal Ordered Systems)

LOTOS [**LOT05**] est un langage de spécification basé sur l'algèbre de processus et l'utilisation de types abstraits. Une modélisation LOTOS est une spécification algébrique exprimant les propriétés que doit vérifier toute réalisation du modèle sans imposer des contraintes d'implémentations. RTLOTOS étend LOTOS avec des opérateurs temporels : délai, latence et offre limitée dans le temps. La vérification de programmes LOTOS est réalisée par le model-checker CADP (Construction and Analysis of Distributed Processes) [**FGK+96**].

Dans cette sous-catégorie on trouve un travail utilisé ce langage :

- Dans ce travail [**LOT02**], ils sont proposés l'approche TURTLE c'est-à-dire de transformation de modèle UML temps réel vers le langage formel RTLOTOS. Cette conception peut être synthétisée à partir de scénarios. L'effort doit se poursuivre pour proposer aux concepteurs de systèmes temps réel et distribués des langages plus puissants en termes d'expression de propriétés à satisfaire par le système. Une conception TURTLE natif repose sur un diagramme de classes pour définir l'architecture du système comme un ensemble de *Tclasses* (TURTLE classes). Ces Tclasses communiquent par rendez-vous sur des portes (*gates*). Chacune Tclasse se voit associer un diagramme d'activité représentant son comportement interne décrit par un diagramme d'activité dont les actions font références soit à ces portes, soit aux attributs. Un diagramme d'activité TURTLE autorise la représentation de choix non déterministes. Il permet enfin est surtout de décrire des mécanismes et des contraintes temporelles.

III.3.4 Approches basées sur les réseaux de pétri

Un réseau de Pétri [**Mur89**] est un graphe biparti orienté qui permet la modélisation du comportement des systèmes dynamiques. Les nœuds de graphe sont des places (marquées par un nombre de jetons) et des transitions reliées par des arcs étiquetés. Le marquage de toutes les places d'un réseau de Pétri à un instant donné représente l'état du système, qui se modifie dynamiquement suite aux franchissements des transitions du réseau de Pétri.

De nombreux travaux sont axés sur la transformation de l'aspect dynamique (diagrammes d'activités) d'UML vers RDP, nous sont distingués deux travaux [**HARALD, GF10**].

- Dans ce travail [**HARALD**] a automatisé la transformation des diagrammes d'activités vers les réseaux de Pétri. Nous avons établi les règles de transformation en ATLAS Transformation Langage (ATL) pour obtenir un modelé conforme à notre Meta-modèle réseaux de Pétri. La sémantique du diagramme d'activité a été validée à l'aide de la transformation **PetriNet2Tina**[4] qui va permettre de vérifier de façon formelle la correspondance sémantique entre les deux modelés après transformation. Cette vérification est effectuée avec le "model-checker" TIme pétri Net Analyser (TINA) et le langage Linear Temporal Logic (LTL). L'utilisateur devra simplement établir le diagramme d'activité à partir des exigences des parties prenantes, la transformation et la vérification étant automatiques. Le formalisme des Réseaux de Pétri nous permet de fournir de précieuses informations sur le diagramme d'activités pour le jouer et le simuler.
- Dans ce travail [**GF10**], ils ont proposé une approche, permettant la modélisation des agents mobiles avec le formalisme de diagramme d'activités mobile. Ils sont compréhensibles et faciles mais souffrent d'un manque de sémantique formelle, cependant ils ne permettent pas l'analyse et la vérification. Une grammaire de graphe généré avec l'outil AToM3 permet la transformation automatique des diagrammes d'activités mobiles vers leurs équivalents des réseaux de Pétri imbriqués, ces derniers peuvent être utilisés comme base pour une éventuelle vérification et analyse.

[4]**PetriNet2TINA** Cette transformation de type "model2text" permet d'adapter le modèle réseauxde Pétri obtenue à l'outil TINA.

III.4 Conclusion

Les diagrammes d'activité sont des mécanismes pour exprimer le séquencement, le choix et le parallélisme. Mais ils souffrent du manque des outils de vérification de comportement, qui sont disponibles dans les méthodes formelles pour cela nous avons étude quelques exemples de travaux illustre la formalisation du diagramme d'activité.

Dans ce chapitre, nous avons commencé par les techniques de transformation de modèle semi-formel vers langage formel, c'est la base de toute formalisation d'UML vers le langage formel, Ensuite, nous avons présenté brièvement les travaux de formalisation du diagramme d'activité.

Le chapitre suivant présenté notre démarche de transformation du diagramme d'activité vers FoCaLiZe pour définir les règles de transformation adopte.

Chapitre IV

La transformation des diagrammes d'activité vers FoCaLiZe

IV.1 Introduction

La transformation de diagramme d'activité UML vers FoCaLiZe consisté à formaliser les actions, les transitions entre les actions et les différents nœuds de contrôle (décision, bifurcation, union et fusion). À l'aide de l'avancement des études et les recherches existant dans le domaine de transformations de diagramme de classes UML vers FoCaLiZe à cause de l'importance de diagramme de classes (tous les modèles UML basés sur ce diagramme) et la grande similitude entre les deux thèmes.

Alors, dans ce chapitre nous allons expliquer premièrement les règles simples de transformations d'une classe UML vers FoCaLiZe que seront utilisées dans notre démarche. Puis nous présentons les règles de transformation de diagrammes d'activités UML vers FoCaLiZe.

La transformation d'une classe présentée est basée sur des études et des travaux [**Abb+14**, **Abb14**, **kha+15**].

IV.2 Transformation d'une classe

Une classe UML est transformée en une espèce FoCaLiZe. Notons que nous préservons l'identificateur de la classe (nom de la classe) pour les espèces générées. Une classe est transformée en espèce, comme suit :

Classe UML	Code FoCaLiZe

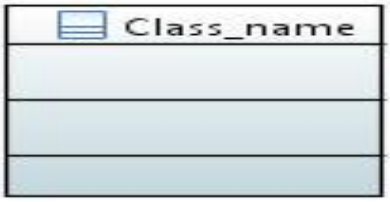
	`Species Class_name =`
	…
	
	`end;;`

Table IV. 1: Transformation de classe UML

Comme nous l'avons mentionné précédemment, une classe peut contenir une liste d'attributs et des opérations. Ensuite, nous allons décrire la transformation de ce dernier.

IV.2.1 Transformation des attributs

Chaque attribut de classe est formalisé par une fonction "getter" (qui renvoie la valeur de l'attribut) dans les espèces correspondantes.

Pour un attribut **attr: attr_type**, la fonction que modélise-la d'attribut **attr** dans les espèces correspondantes est:

Signature get_attr: Self ->Foc_attr_type;

Où ***Foc_attr_type*** est le type correspond à attr: attr_type en FoCaLiZe.

Il existe deux possibilités pour traduire le type d'attributs:

1. Si **attr_type** est un UML types primitifs (entier, réel, String et Boolean), nous utilisons directement les types primitifs en FoCaLiZe (int, float, string et bool) ;
2. Si **attr_type** n'est pas un type primitif (un type de classe), nous utilisons les espèces correspondantes.

La table suivant montre la transformation d'une classe UML avec des attributs:

Les attributs de classe	Code FoCaLiZe

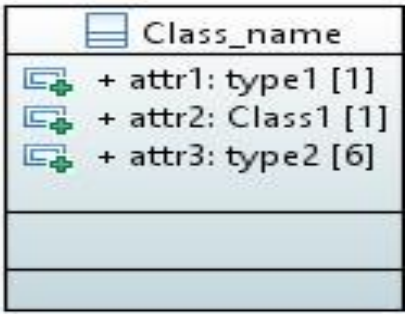
	`species Class_name(C is Class1)=`
	`signature get_attr1: Self ->`
	`attr_type1 ;`
	`signature get_attr2: Self -> C ;`
	`signature get_attr2: Self -`
	`>list(type2) ;`
	`end;;`

Table IV. 2: Transformation d'une classe UML avec des attributs

Si la multiplicité des **attr_type** est supérieur à un (comme l'attribut **attr3** dans la table IV.2): nous utilisons le type liste dans FoCaLiZe:

Signature get_attr: Self ->list (***Foc_attr_type)***;

IV.2.2 Transformation des opérations

Les opérations de classe seront transformées en fonctions (signature) d'espèces correspondantes. La signature dérivée d'une opération d'UML dépend des paramètres de l'opération (d'entrée et de sortie). Les types des paramètres peuvent être des types primitifs ou de type classe. La transformation des types d'opérations est similaire à la transformation des types d'attributs (voir la section précédente). Nous distinguons les cas suivants :

✓ Si l'opération a plusieurs paramètres "**in**" et un paramètre "**out**":

Op_name (p_name1: type 1...,p_name n: typen): return_type, sa transformation est:

Signature op_name: self -> type$_1$->… ->type$_n$->return_type;

✓ Si l'opération a plusieurs paramètres "**in**" et sans paramètre "**out**":

Op_name (p_name 1: type 1, ..., p_name n: type n), sa transformation est:

Signature op_name: self -> type$_1$->… ->type$_n$->self;

✓ Si l'opération sans paramètres du tous :

Op_name (), elle est transformée en une signature des espèces correspondantes:

Signature op_name: self ->self;

La table suivant montre la transformation d'une classe avec des opérations :

Les opérations de classe	Code FoCaLiZe

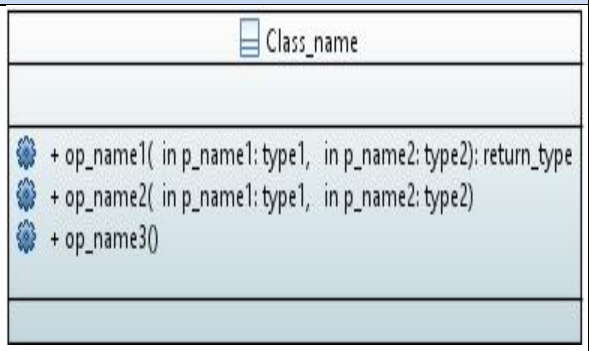
	`Species Class_name =`
	`signature op_name1: Self -> type1`
	`->type2 ->return_type ;`
	`signature op2_name: Self ->type1`
	`->type2 -> Self;`
	`signature op3_name: Self ->Self;`
	`end;;`

Table IV. 3: Transformation d'une classe UML avec des opérations

Quand une classe UML spécifiée sans un constructeur, nous ajoutons une fonction supplémentaire pour les espèces correspondantes qui doivent correspondre à la méthode "new" (de la programmation orientée objet) comme suit:

new species_name: type$_1$ ->…->type$_n$ -> self;

Où $type_1$, ..., $type_n$ sont les types des attributs de classe. Par exemple: Nous transformons le constructeur de la Class_name comme suit:

Classe UML	Code FoCaLiZe

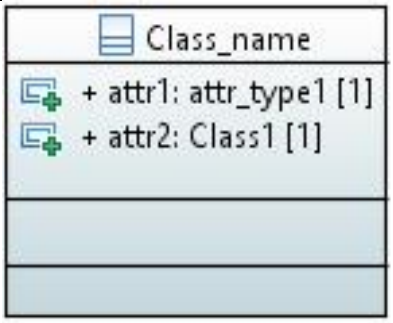
	`Species Class_name(C is Class1) =`
	`signature get_attr1: Self -> attr_type1 ;`
	`signature get_attr2: Self -> C ;`
	`new Class_name: attr_type1 -> C -> self ;`
	`end;;`

Table IV. 4: Transformation du constructeur de la classe

IV.3 Transformation de diagramme d'activité

Pour transformer les diagrammes d'activité en FoCaLiZe, nous allons proposer une approche basée sur les transformations précédentes (diagramme de classes). De plus, nous introduisons les règles de transformation de diagramme d'activité correspondant.

Exemple :

Dans les paragraphes suivants, nous allons utiliser le diagramme d'activité suivant (de la classe gestion de commande :

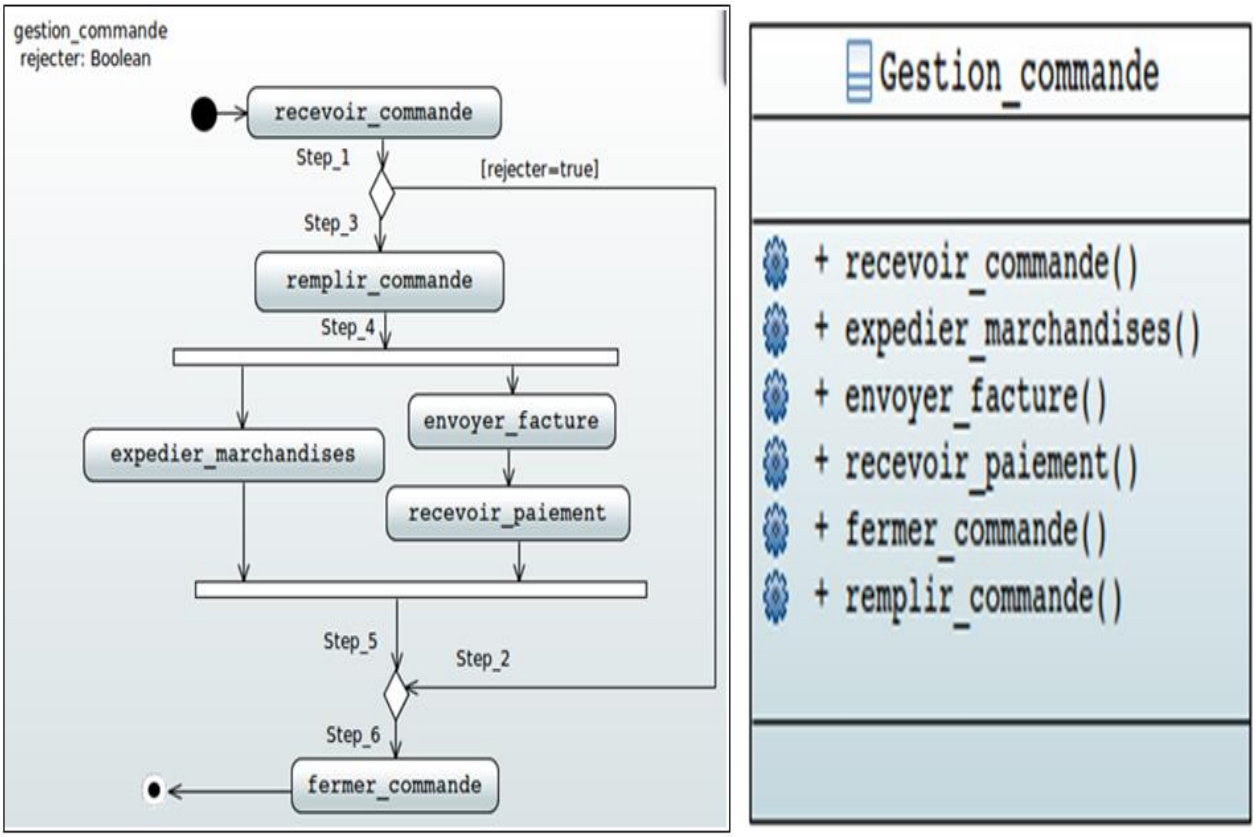


Figure IV. 1 : Diagramme d'activité de la classe "Gestion_commande"

Notons que chaque action dans le diagramme d'activité est une opération dans la classe UML.

Dans notre démarche de transformation, nous allons correspondre une fonction récursive définie (let rec) à chaque diagramme d'activité d'une classe UML. Cette fonction est définie dans l'espèce correspondante à la classe.

L'entête de la fonction modélisant le diagramme d'activité est spécifié comme suit :

let rec diagAc (s :trans, e :self):(trans*self) ;

Tel que : **s** est un paramètre de type **trans** (énumération regroupant toute les transitions d'un diagramme d'activité) et **e** est une entité de l'espèce.

La fonction retourne un couple de type (**trans*self**).

La table suivante montre la transformation générale d'un diagramme d'activité :

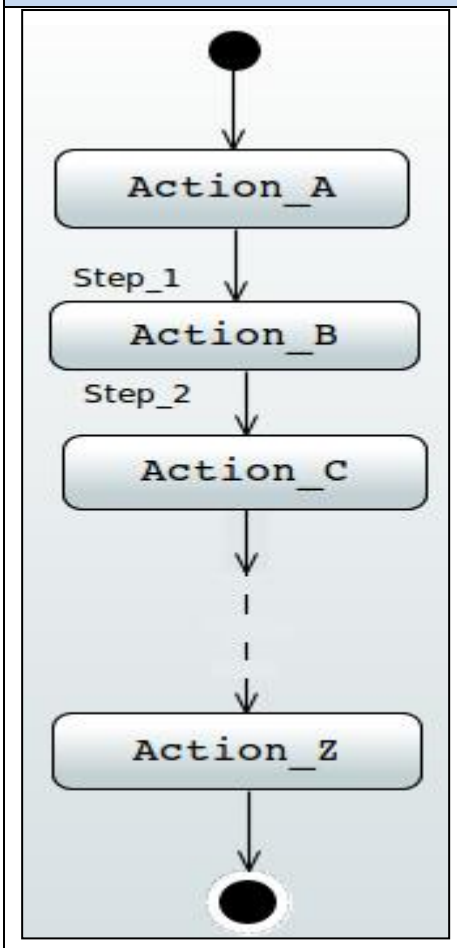
Activité	Code FoCaLiZe
	type trans = |Step_0|Step_Init| Step_1| Step_2 … |Step_Final; Species Class_name = (*déclaration des attributs existants*) (*déclaration des fonctions existants*) let rec diagAc(s:trans,e :self):(trans*self) = match s with (*partie de clarifier les passages des actions*) |Step_Init ->diagAc(Step_1,ActionA(e)) |... |... | Step_Final ->diagAc(Step_0, e) | Step_0 ->focalize_error("Fin de traitement") ; End;;

Table IV. 5: Transformation générale d'un diagramme d'activité

Dans ce qui suit, nous allons détailler la transformation de chaque composant de diagramme d'activité, à savoir les transitions, les nœuds de contrôle, la partition.

IV.3.1 Transformation de transition et flot de contrôle

Les transitions sont les plus utiles parce qu'ils nous donnent la chronologie de successivement des actions et ils décrivent le flot de données de chaque action. Ils sont modélisées en FoCaLiZe par la création d'un nouveau type d'énumération "**trans**" au niveau top (en dehors l'espèce). Chaque transition est modélisée par une valeur de ce type, comme suivant :

type trans = \|Step_0 \|Step_Init \| Step_1\| Step_2 … \| Step_n \|Step_Final

Tel que:

- ✓ Step_Init modélise la transition sortant du nœud initiale ;
- ✓ Step_Final modélise la transition entrant dans le nœud final ;
- ✓ Step_1, Step_2, …, Step_n modélisent les transitions intermediaries ;
- ✓ Step_0 est une transition supplémentaire introduite pour mettre fin à la fonction récursive "**diagAc** " ;

Chaque pattern de la fonction **diagAc** va correspondre au traitement d'une transition, comme suit :

Les transitions et flot de contrôle	Code FoCaLiZe

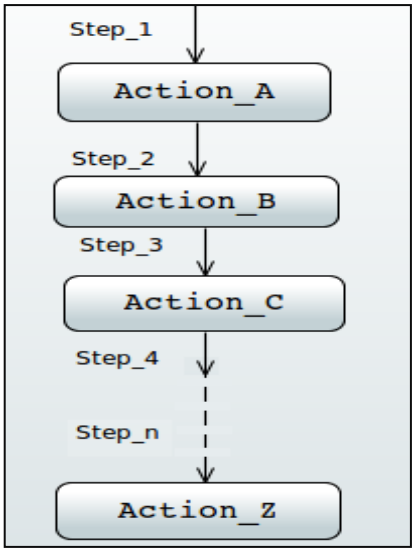
	`type trans = |Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;`
	`Species Class_name=`
	`  …`
	`let rec diagAc(s: trans, e:Self):(trans *Self)= match s with`
	`|Step_Init ->diagAc(Step1, e)`
	`|Step_1 ->diagAc(Step_2, e)`
	`|Step_2 ->diagAc(Step_3, e)`
	`|Step_3 ->diagAc(Step_4, e)`
	`.`
	`.`
	`.`
	`|Step_n ->diagAc(stp0, e)`
	`End;;`

Table IV. 6: Transformation des transitions

Tel que l'appel de la fonction **diagAc** (**Step_i, e**) invoque le traitement de la transition **Step_i**.

IV.3.2 Transformation de nœud de décision

Le nœud de décision est formalisé par l'expression :

if (cnd) then transition_i else transition_j ;

Où cnd est la condition de garde de la décision. Dans notre démarche nous utilisant le langage OCL (Object Constraint Langage) [**OMG12**] pour exprimer les conditions de gardes des nœuds de décisions.

Le langage OCL est un langage du standard OMG[5] (Object Management Group) qui permet la description des contraintes (propriétés) sur les modèles UML.

La table suivante présente la règle de transformation d'un nœud de décision :

Nœud de décision	Code FoCaLiZe

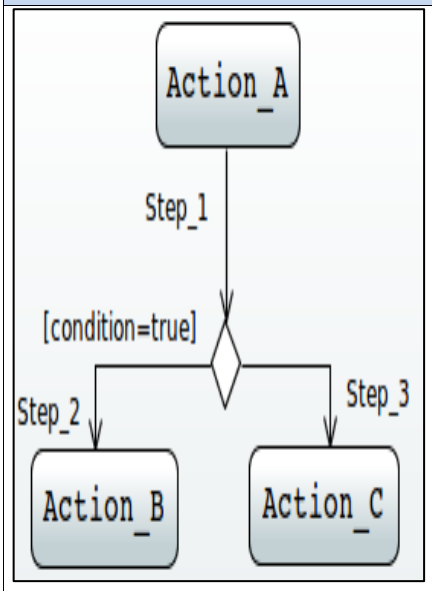
	`Type trans = |Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n`
	`Species Class_name=`
	…
	`let rec diagAc(s:trans,e:Self):`
	`(trans*Self) = match s with`
	`|Step_1->if [condition(e)] then diagAc(Step_2,e)else diagAc(Step_3,e)`
	`|Step_2-> ……`
	`|Step_3-> ……`
	`End;;`

Table IV. 7: Transformation de nœud de décision

La table suivante montre un exemple de transformation d'une décision de notre diagramme d'activité de la figure IV.1 donnée ci-dessus.

[5] OMG :http://www.omg.org/

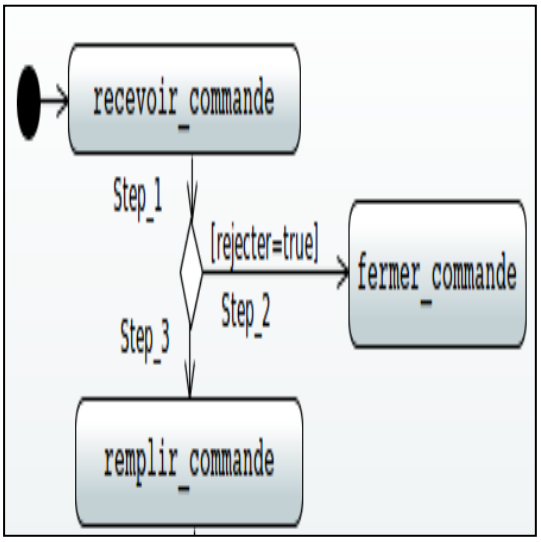
Nœud de décision	Code FoCaLiZe
	Type trans = |Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;; Species Class_name= … let rec diagAc(s:trans,e:Self): (trans*Self) = match s with | Step_1 -> if [rejecter(e) =true] then diagAc(Step_3, e) else diagAc(Step_2, e) |Step_2-> …… |Step_3-> …… End;;

Table IV. 8: Exemple de transformation de nœud de décision

IV.3.3 Transformation de nœud de fusion

La formalisation d'un nœud de fusion consiste à regrouper le traitement de plusieurs transitions dans un seul pattern dans la fonction modélisant le diagramme d'activité (diagAc), comme suit :

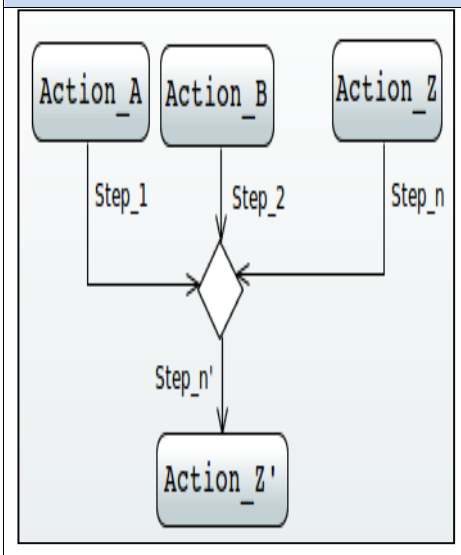
Nœud de fusion	Code FoCaLiZe
	Type trans = |Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;; Species Class_name= … let rec diagAc(s:trans,e:Self): (trans*Self)= match s with |Step_1->diagAc(step_n', e) |Step_2 ->diagAc(step_n', e) . . |Step_n ->diagAc(step_n', e) . End;;

Table IV. 9: Transformation de nœud de fusion

Dans la table suivante, nous représentons la transformation de nœud de fusion de regroupant les transitions Step_5 et Step_2 dans notre diagramme d'activité de figure IV.1 :

Nœud de fusion	Code FoCaLiZe

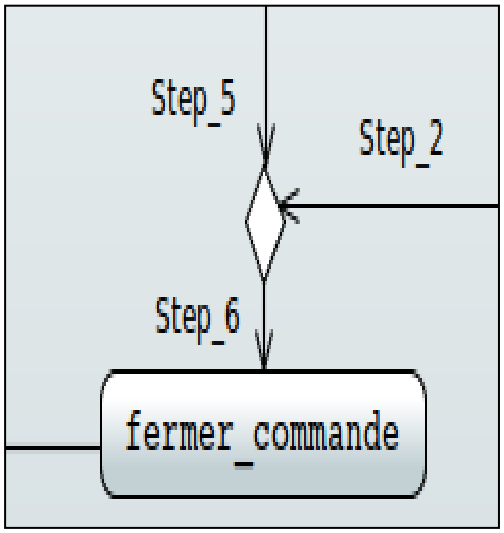
	`Type trans = Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;;`
	`Species Class_name=`
	`…`
	`let rec diagAc(s:trans,e:Self): (trans*Self) = match s with`
	`| Step_2 ->diagAc(Step_6, e)`
	`| Step_5 ->diagAc(Step_6, e)`
	`.`
	`.`
	`|Step_n ->diagAc(step_0, e)`
	`End;;`

Table IV. 10: Exemple de transformation de nœud de fusion

IV.3.4 Transformation de nœud de bifurcation et d'union

Logiquement les deux nœuds (union et bifurcation) ne peuvent pas gérer chacun individuellement. Ils sont transformés en FoCaLiZe par l'utilisation de l'expression FoCaLiZe :

let x_1= exp_1 and x_2=exp_2, … and x_n= exp_n in diagAc (Step_i, join(x_1, x_2, …, x_n)) ;

Où chaque $\mathbf{x_i}$ représente le traitement d'une transition sortant de la bifurcation et **join** est une fonction que nous définissons au niveau de l'espèce.

La règle de transformation de nœud de bifurcation et d'union est donnée dans la table suivante :

Nœud de bifurcation et union	Code FoCaLiZe

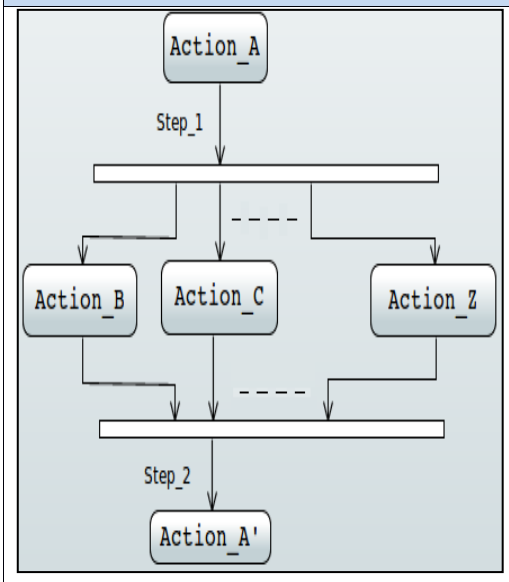
	`Type trans = Step_0 | Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;;`
	`Species Class_name=`
	`…`
	`let rec diagAc(s:trans,e:Self): (trans*Self) = match s with`
	`|Step_1-> let x1 nom_ActionB(e)`
	`and x2 nom_ActionC(e)`
	`…`
	`and x_n= nom_ActionZ(e)`
	`in diagAc( Step_2, join(x1, x2, …, x_n))`
	`…`
	`|Step_n ->diagAc(Step_0, e)`
	`End;;`

Table IV. 11: Transformation de nœud de bifurcation et union

La table suivante montre un exemple de transformation d'un nœud de bifurcation et d'union de notre diagramme d'activité de la figure VI.1 :

Nœud de bifurcation et union	Code FoCaLiZe

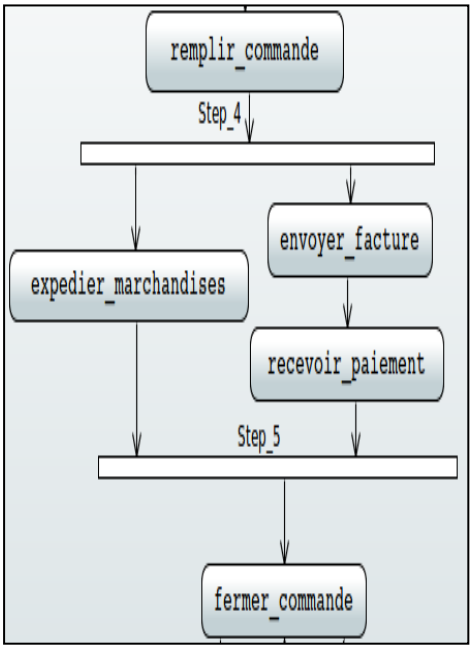
	`Type trans = |Step_0|Step_Init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;;`
	`species Class_name=`
	`…`
	`let rec diagAc(s:trans,e:Self):(trans*Self) = match s with`
	`|Step_4->let x1= expedier_marchandises(e) and x2= recevoir_paiement(envoyer_facture(e))`
	`in diagAc(Step_5 , join5(x1 , x2))`
	`…`
	`|Step_n ->diagAc(Step_0, e)`
	`End;;`

Table IV. 12: Exemple de transformation de nœud de bifurcation et union

3.4.1 Transformation de nœud de bifurcation

Comme nous avons vu dans la règle de transformation précédente, La bifurcation ou la synchronisation se traduit en FoCaLiZe par :

let x_1= nom_Action$_A$ and x_2= nom_Action$_B$ and … and x_n=nom_Action$_Z$.;

Comme montre la table suivante:

Nœud de bifurcation	Code FoCaLiZe

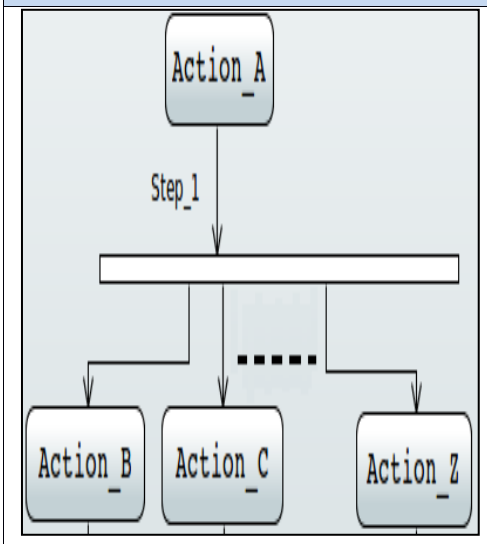
	`Type trans = Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;;`
	`Species Class_name=`
	`…`
	`let rec diagAc(s:trans,e:Self): (trans*Self ) = match s with`
	`|step_1-> let x1 = nom_ActionB(e)`
	`and x2 = nom_ActionC(e)`
	`…`
	`and x_n= nom_ActionZ(e)`
	`…`
	`end;;`

Table IV. 13: Transformation de nœud de bifurcation

3.4.2 Transformation de nœud d'union

L'union ou la fin de synchronisation se formalise par la fonction "**Join**". A partir la règle suivant : ***in diagAc (Step_i, join(x1, x2, …, xn)) ;***

Voilà le code de transformation de ce nœud dans la table suivant:

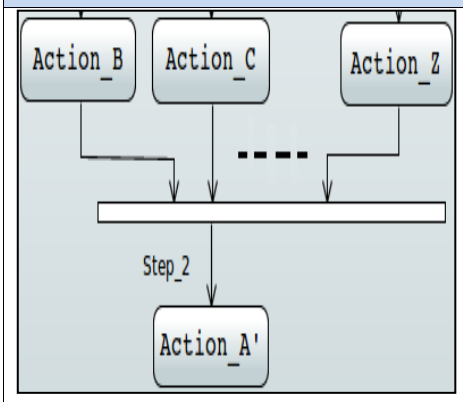
Nœud d'union	Code FoCaLiZe
	Type trans = Step_0 |Step_init|Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;; Species Class_name= … let rec diagAc(s:trans, e:Self): (trans*Self) = match s with |Step_1 -> … in diagAc(Step_2,join(x1, x2, …, x_n)) ….. End;;

Table IV. 14: Transformation de nœud d'union

IV.3.5 Transformation de nœud initial et de nœud final

Comme nous l'avons indiqué plus haut les nœuds initiale et finale sont formalisés en FoCaLiZe par les patterns Step_Init et Step_Final de la fonction **diagAc,** comme suit :

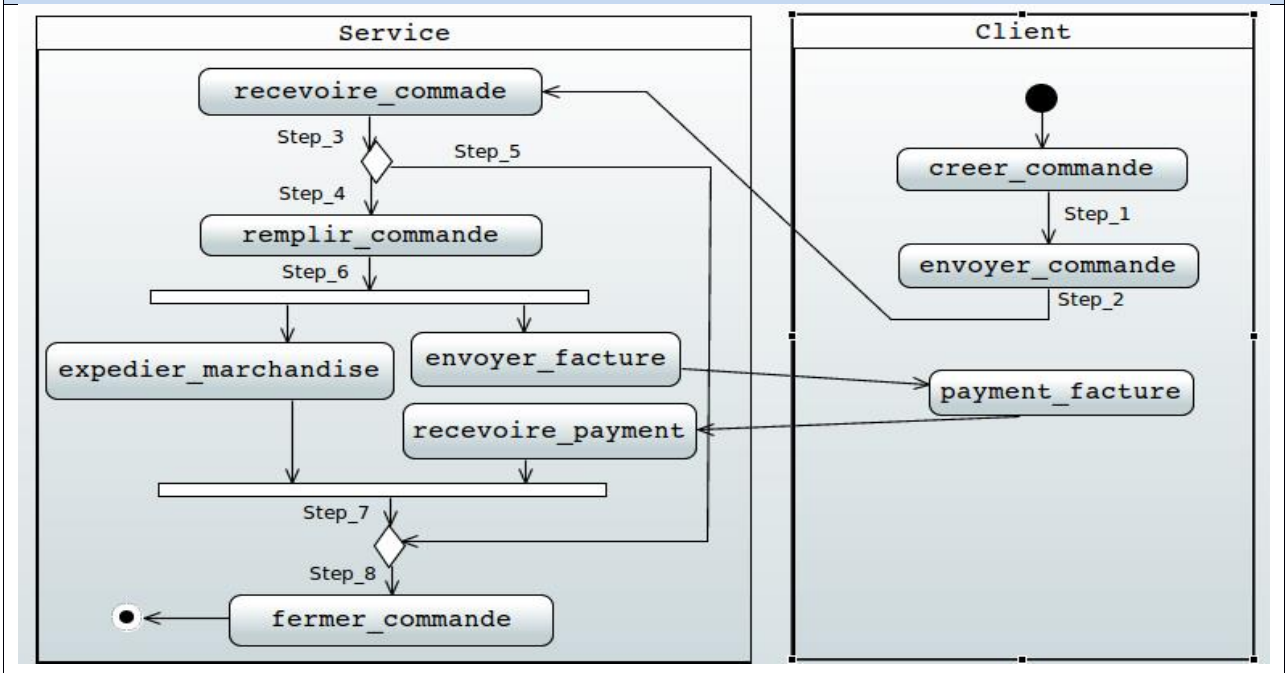
Nœud initial	Code FoCaLiZe
	Type trans = |Step_0 |Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n;; species Class_name= … let rec diagAct(s:trans,e:Self): (trans*Self) = match s with | Step_init-> (*l'action que fait en ce pas*) ….. End ;;
Nœud final	Code FoCaLiZe
	Type trans = |Step_0|Step_init| Step_1 | Step_2 | Step_3 | Step_4. . . | Step_n; species Class_name= … let rec diagAc(s:trans, e:Self): (trans*Self) = match s with …… |Step_Final->diagAc(Step_0, e) |Step_0->focalize_error("Fin de traitement") ; End ;;

Table IV. 15: Transformation de nœud initiale et finale

IV.3.6 Transformation de partition

La partition est modélisée en FoCaLiZe par une fonction définie dans une espèce paramétrée par des espèces dérivées des acteurs (classes). Ceci pour permettre de référencer les méthodes de chaque espèce :

Partition



Code FoCaLiZe

```
type trans = | Step_0 | Step_init | Step_1 | Step_3 | Step_2 |
Step_4 | Step_5 | Step_6 |  Step_Final;;

(* species Service*)
species  Service =
 signature recevoir_commande : Self -> Self;
 signature expedier_marchandises : Self -> Self;
 signature envoyer_facture : Self -> Self;
 signature recevoir_paiement : Self -> Self;
 signature fermer_commande : Self -> Self;
 signature remplir_commande : Self -> Self;
 signature rejecter: Self ->bool;
end;;

(* species Client*)
species  Client =
 signature creer_commande : Self -> Self;
 signature envoyer_commande : Self -> Self;
 signature payment_facture : Self -> Self;
end;;
```

```
species Gestion_commande(A is Service, B is Client) =

 let rec diagAc (s:trans, e:Self):
(trans * Self) = match s with
 | Step_init ->diagAc(Step_1,B!creer_commande(e))
 | Step_1 ->diagAc(Step_2, B!envoyer_commande(e))
 | Step_2 ->diagAc(Step_3,A!recevoir_commande(e))
 | Step_3 -> if ( rejecter(e) =true) Then
                diagAc(Step_5, e)
             else diagAc(Step_4, e)
 | Step_4 ->diagAc(Step_6, A!remplir_commande(e))
 | Step_5 ->diagAc(Step_8, e)
 | Step_6 -> let  x1 = A!expedier_marchandises(e)   and
   x2=A!recevoir_paiement(B!payment_facture(A!envoyer_facture(e))
   in diagAc(Step_7 , join5(x1 , x2))
 | Step_7 ->diagAc(Step_8, e)
 | Step_8 ->diagAc(Step_finale , A!fermer_commande(e))
 | Step_Final ->diagAc(Step_0, e)
 | Step_0 ->focalize_error("Fin de traitement") ;
  end;;
```

Table IV. 16: Transformation de partition

IV.4 Conclusion

Dans ce chapitre, nous avons d'abord décrit la relation entre le diagramme de classe et le diagramme d'activité. Puis, nous définissons les régles de transformation des classes et leurs diagrammes d'activités correspondants en FoCaLiZe.

Dans cette démarche, un diagramme d'activité est formalisé par une fonction récursive qui permettra la vérification des certaines propriétés en complétant la spécification FoCaLiZe générée.

Chapitre V

L'implémentation

V.1 Introduction

Dans ce chapitre, nous décrivons les différentes étapes de notre processus de transformation afin de transformer le modèle UML (diagramme d'activité) vers FoCaLiZe. Notre processus de transformation consiste en deux étapes: premièrement, Nous utilisons un environnement graphique UML qui prend en charge le méta-modèle OMG pour créer notre modèle d'UML et de générer son format XMI (XML Meta data Interchange) correspondant. La dernière étape consiste à appliquer nos règles de transformation pour générer le code FoCaLiZe.

Pour mettre en œuvre notre processus de transformation, nous avons utilisé l'environnement eclipse avec Papyrus plugin et le langage XSLT (eXtensible Stylesheet Language Transformation). Ce choix spécifique est à cause de la flexibilité de l'environnement éclipse, et la mise en œuvre complète de toutes les fonctionnalités d'UML2 par Papyrus.

V.1 L'environnement de travail

V.2.1 Eclipse

L'outil principal utilisé durant ces travaux est Eclipse[6] Modeling Framework (EMF), c'est un kit de composants logiciels structurels (plus couramment appelé framework) pour la construction d'outils basées sur une approche modèle. Les modèles y sont décrits en XML, mais peuvent être spécifiés dans des documents UML/OCL (solution utilisée durant ces travaux). L'interopérabilité avec d'autres outils basés sur Eclipse est un des points forts de ce framework. Un de ces outils appelé Graphical Modeling Framework (GMF) permet le développement d'éditeur graphique de modèle [**HARALD**].

V.2.2 Papyrus

Des boites à outils peuvent être ajoutées à EMF pour créer un environnement de travail oriente vers un domaine précis. Papyrus[7] est un logiciel d'ingénierie assiste par ordinateur. Il

6 Eclipse : https://eclipse.org/luna/

7 Papyrus : https://eclipse.org/papyrus/

ajoute à EMF des fonctionnalités essentielles à ce projet liées à la mise en œuvre des modèles UML/OCL sous forme graphique.

V.2.3 XSLT

XSLT[8] est une recommandation du W3C. Il est un langage déclaratif qui permet la transformation des documents XML en divers autres XML (avec une structure différente), un document HTML ou un document texte. Pour effectuer une transformation, nous définissons une feuille de style XSLT qui décrit les règles de transformation qui sera traités par un processeur XSLT. Ce dernier est mis en œuvre dans la plupart des environnements de développement (Java, C #, php, ...) **[Kha+15] ;** la figure suivante montre notre schéma de transformation basé sur l'utilisation d'une feuille de style XSLT.

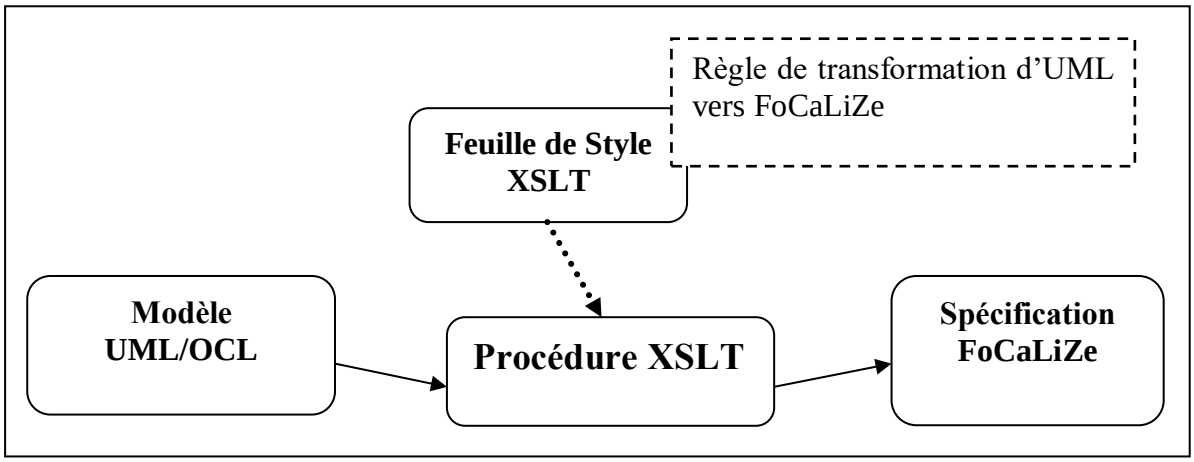


Figure V. 1: Représentation le rôle de feuille de style XSLT

V.2 Processus de génération de code FoCaLiZe

Après la description des règles de transformation d'UML vers FoCaLiZe, nous pouvons maintenant définir un cadre (framework) qui intègre un outil UML/OCL, les règles de transformation et l'environnement FoCaLiZe. Dans ce cadre, le développeur FoCaLiZe sera capable de compléter la spécification générée, jusqu'à obtenir un code exécutable. Pour atteindre cet objectif, nous adoptons le processus d'implémentation suivant (voir figure V.2) :

[8] XSLT : http://www.w3schools.com/xsl/default.asp

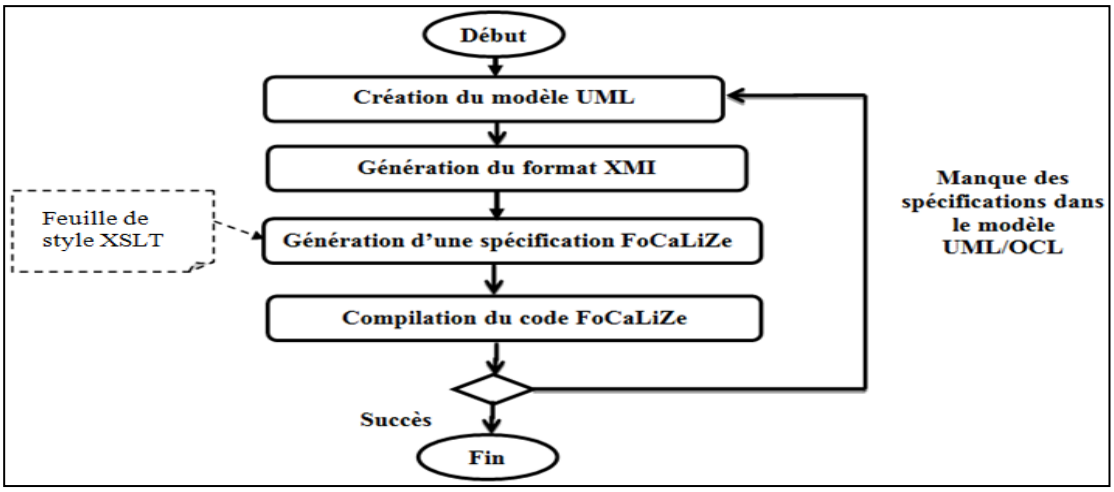


Figure V. 2: Processus de génération de code FoCaLiZe

Dans ce qui suit, nous décrivons les différentes étapes de notre processus de transformation, en commençant par la création du modèle UML et la génération de son format XMI. Ensuite, nous utilisons la feuille de style XSLT implémentant nos règles de transformation pour générer une spécification FoCaLiZe. Enfin, nous compilons le code FoCaLiZe pour vérifier le modèle.

V.2.1 Création du modèle UML

Dans cette étape, nous avons créés une classe UML et son diagramme d'activité correspondant pour décrit le dynamique des opérations.

La création de modèle UML se fait dans le même modèle "**model.id**" en utilisant l'outil Papyrus dans l'environnement Eclipse (voir les figures V.9 -V.10). Puis, nous pouvons générer son format XMI qui sera automatiquement enregistré sous le fichier "**model.uml** " après la sauvegarde de notre projet (voir la figure V.11).

V.2.2 Implémentation des règles de transformation

Cette étape consiste à implémenter les règles de transformation (voir chapitre IV) pour générer un code FoCaLiZe. Alors, Nous avons développés une feuille de style XSLT qui contient un ensemble de template pour représenter les règles de transformation d'une classe et d'un diagramme d'activité vers FoCaLiZe, basée sur le fichier XMI.

Notons que notre démarche de transformation d'un diagramme d'activité est basée sur la transformation de diagramme de classe présentée en [**Kha+15**].

Alors, notre processus de transformation se résume dans la figure suivante:

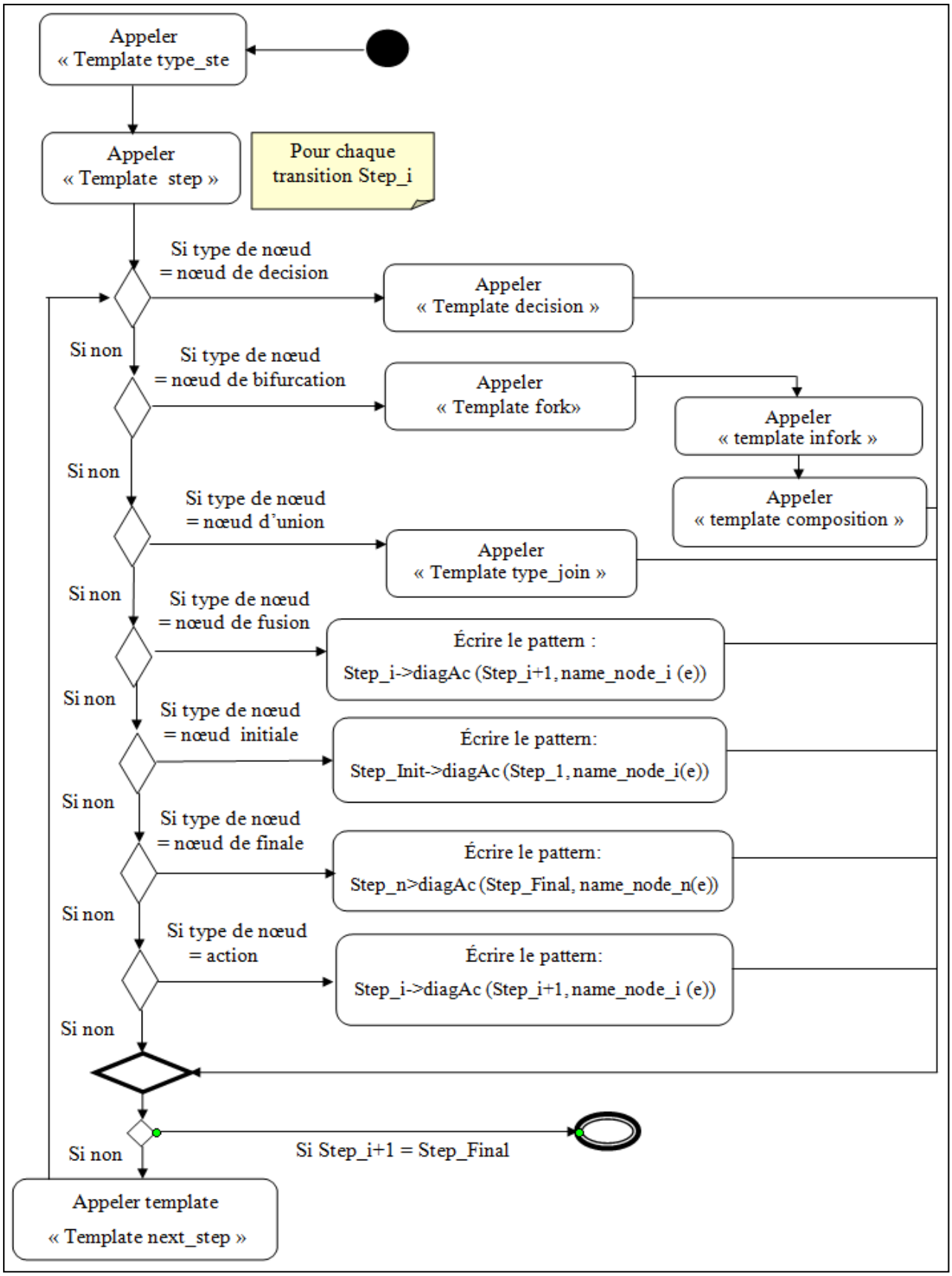


Figure V. 3: Implémentation des règles de transformation

La table suivant décrit tous les templates de notre feuille de style XSLT:

Templates de XSLT	Description
Template1: « type_steps »	Cette template définit la règle de transformation qui détermine le nombre de transitions dans le diagramme d'activité.
Template2: « step »	Cette template définit la règle qui génère la transformation de chaque transition en UML vers FoCaLiZe selon les types existants : (décision, fork, fusion,…, etc).
Template3: « decision »	Cette template transforme la décision en UML vers une condition au format OCL.
Template4: « fork »	Cette template transforme le cas de bifurcation et Union en UML vers un code FoCaLiZe.
Template5: « composition »	Cette template transforme la successivement des actions relier avec le nœud de bifurcation en UML vers une composition des fonctions en FoCaLiZe.
Template6: « type_join »	Cette template déclare une fonction join en FoCaLiZe pour chaque nœud d'union dans UML.
Template7: « infork »	Cette template permet de vérifier la transition en UML vis-à-vis le nœud de bifurcation.
Template8 : « next_step »	Cette template permet de passage de transition vers autre.

Table V. 1: Les templates de XSLT.

V.3 Installation et utilisation de l'outil de transformation

Dans cette section, nous allons détailler les techniques d'utilisation de notre outil de transformation (les étapes d'installation de l'outil sont présentées en Annexe).

❶ Tout d'abord, lancer Papyrus Pour créer une modèle UML / OCL (voir figure V.4).

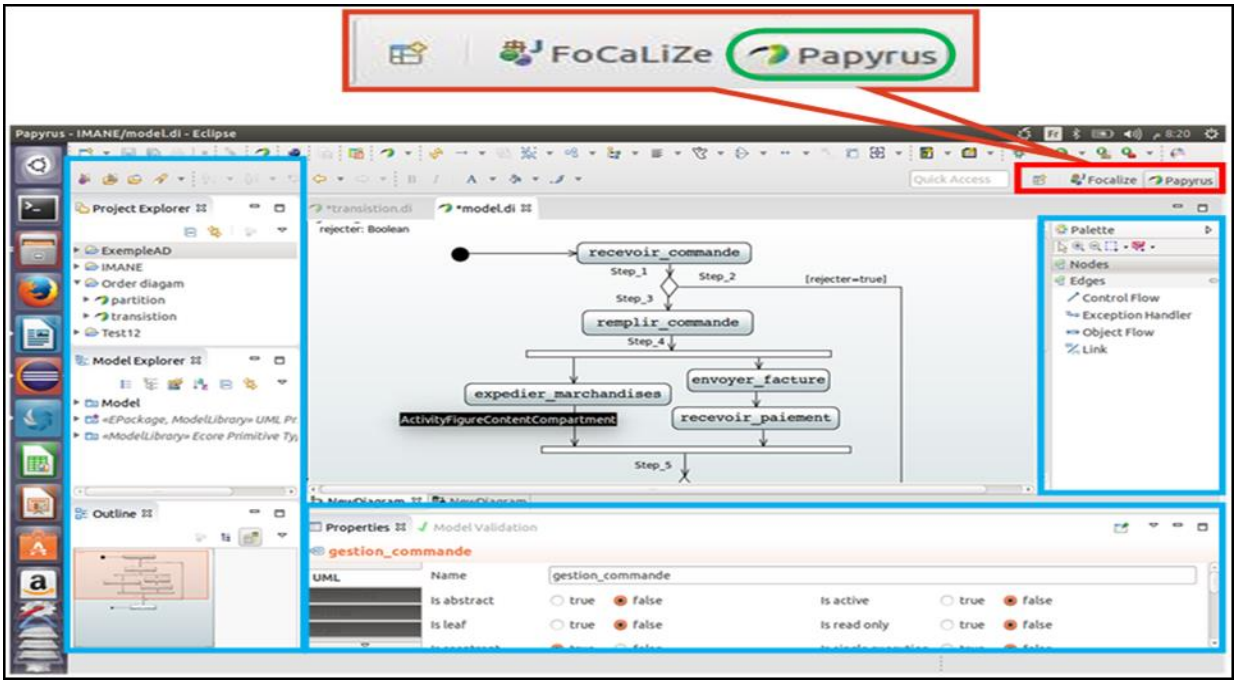


Figure V. 4: Création d'un modèle UML/OCL par papyrus

❷ Déclencher FoCaLiZe, puis faite une clique droite sur le fichier **model.uml** (à gauche de la fenêtre), puis choisir la commande **UmlToFocalize** pour générer le code FoCaLiZe (**model.fcl**) et afficher un message de succès ou échec de la transformation dans la console ; et les deux figures suivantes montre cette étape :

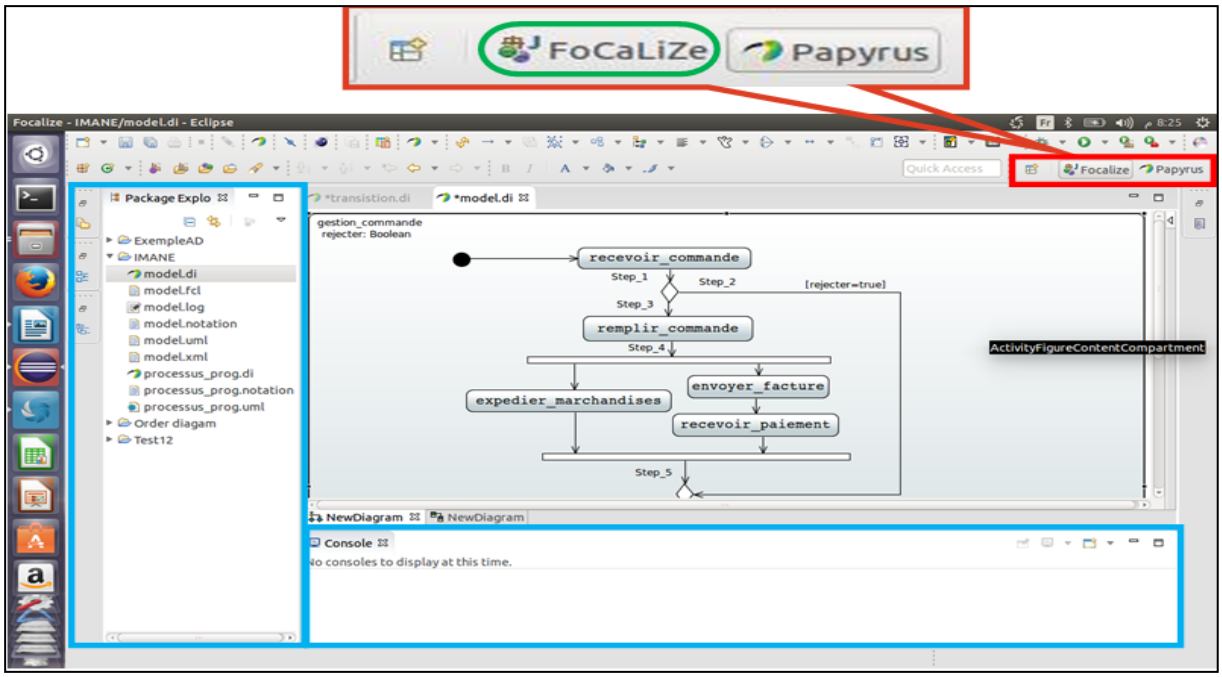


Figure V. 5: Lancement de FoCaLiZe

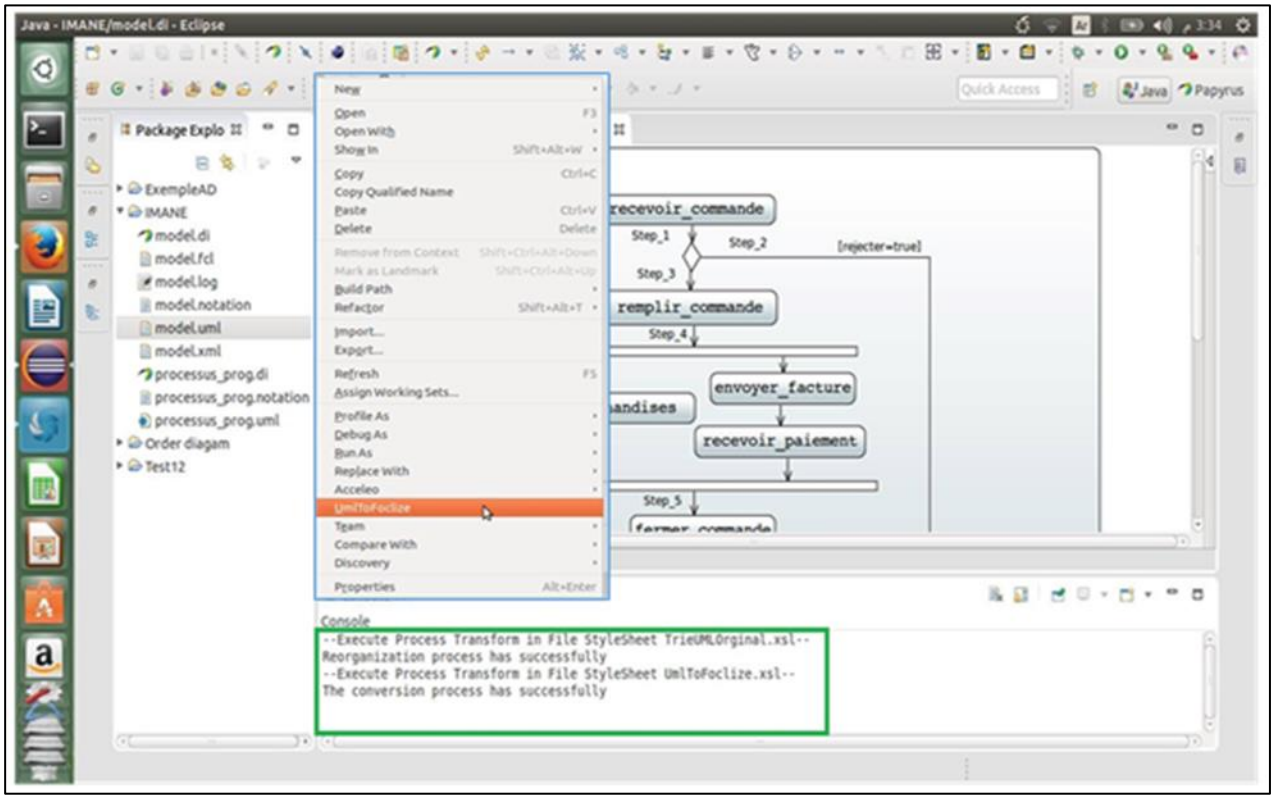


Figure V. 6: Génération de code FoCaLiZe

❸ Faite un clique droite sur le fichier **model.fcl** et choisir la commande "**Exécuter_Focalize**" pour appeler le compilateur FoCaLiZe. Le résultat sera affiché au niveau de la console (voir figure V.7).

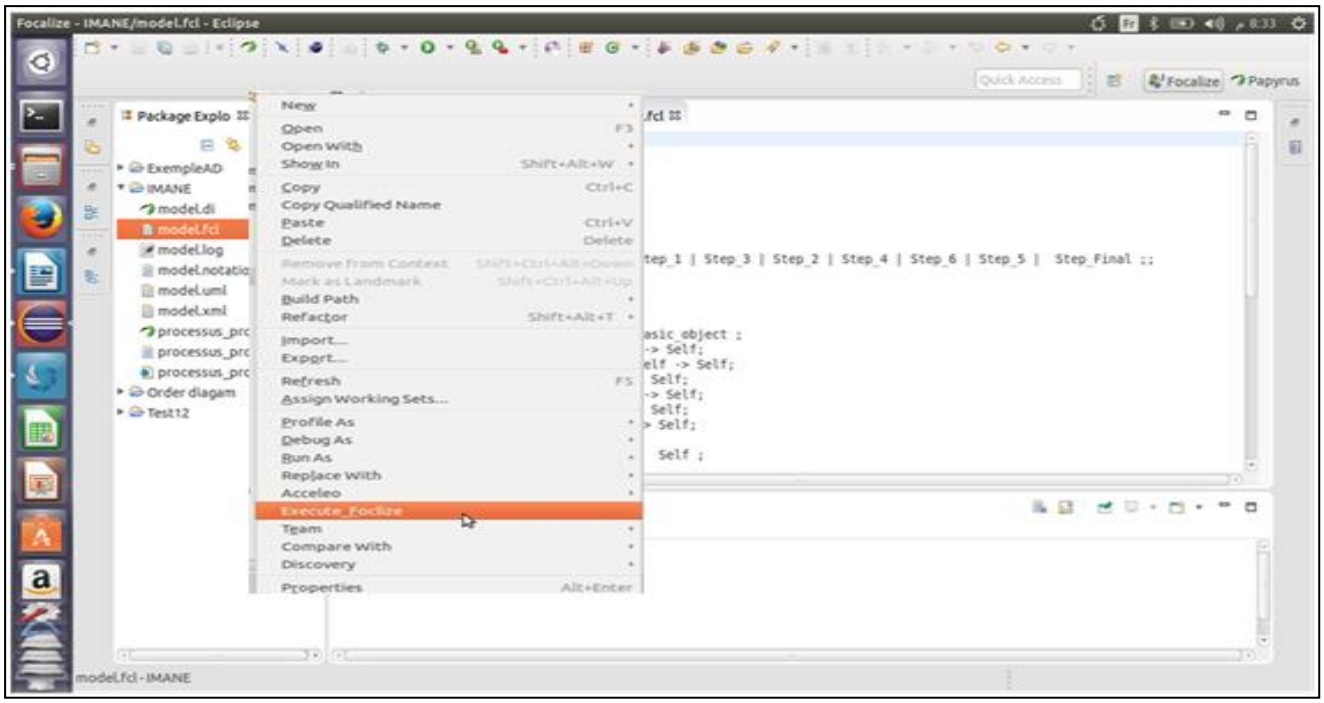


Figure V. 7: Représentation comment choisit la commande **Execute_Focalize** ?

V.4 Exemple de transformation

Pour illustrer notre transformation, nous avons représenté un exemple que décrit notre étapes de transformation en détaille.

V.4.1 Etape1 : la création de modèle UML

Les deux figures suivantes représentent la création de la classe "Gestion_commande" et son diagramme d'activité correspondant.

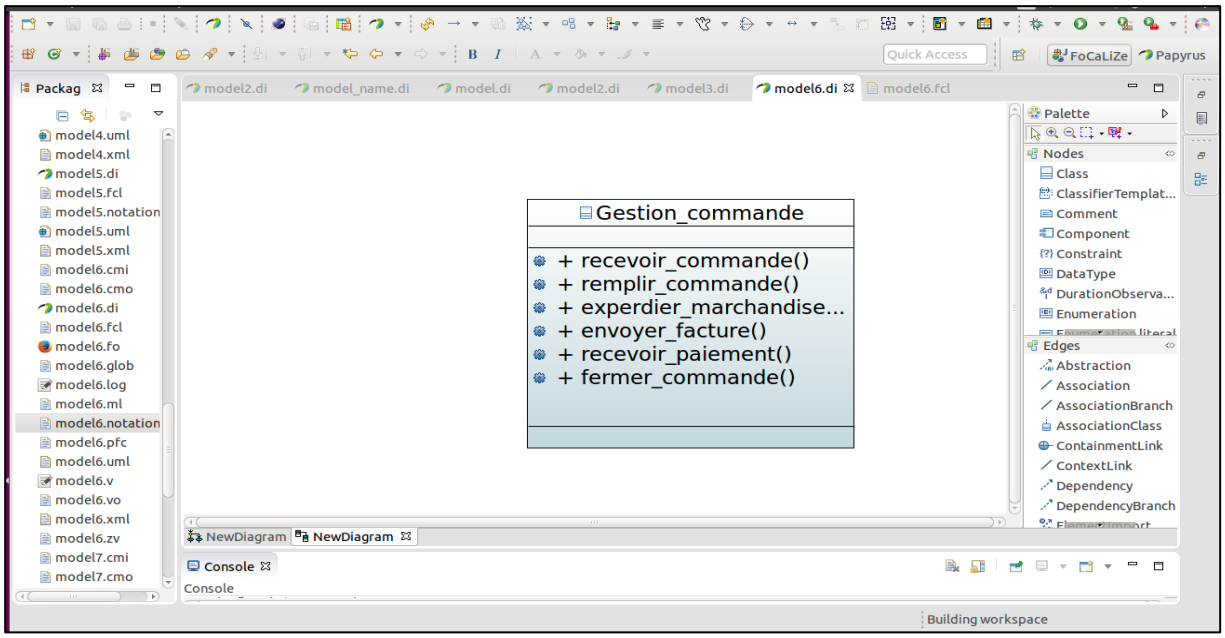


Figure V. 8: La classe "Gestion_commande"

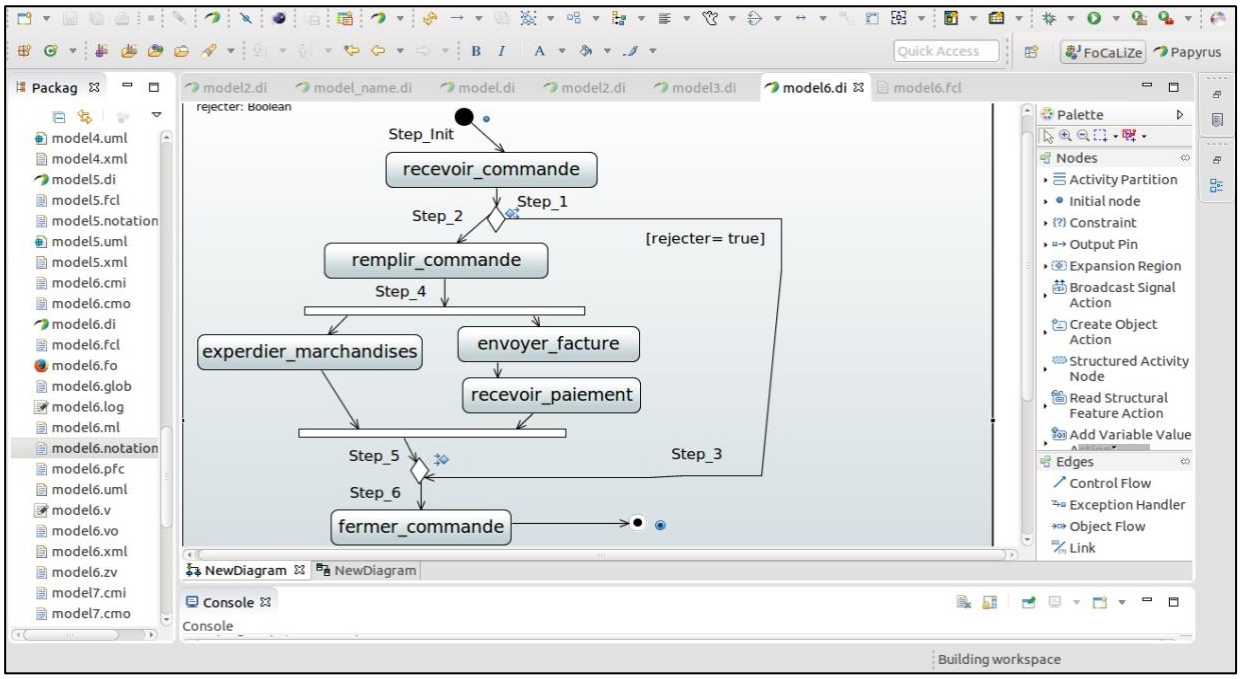


Figure V. 9: Diagramme d'activité de la classe "Gestion_commande"

V.4.2 Etape2 : la génération de format XMI

Le format XMI est enregistré automatiquement sous le fichier "**model.uml**" après l'enregistrement de notre modèle UML. Voir la figure suivante (figure V.10) :

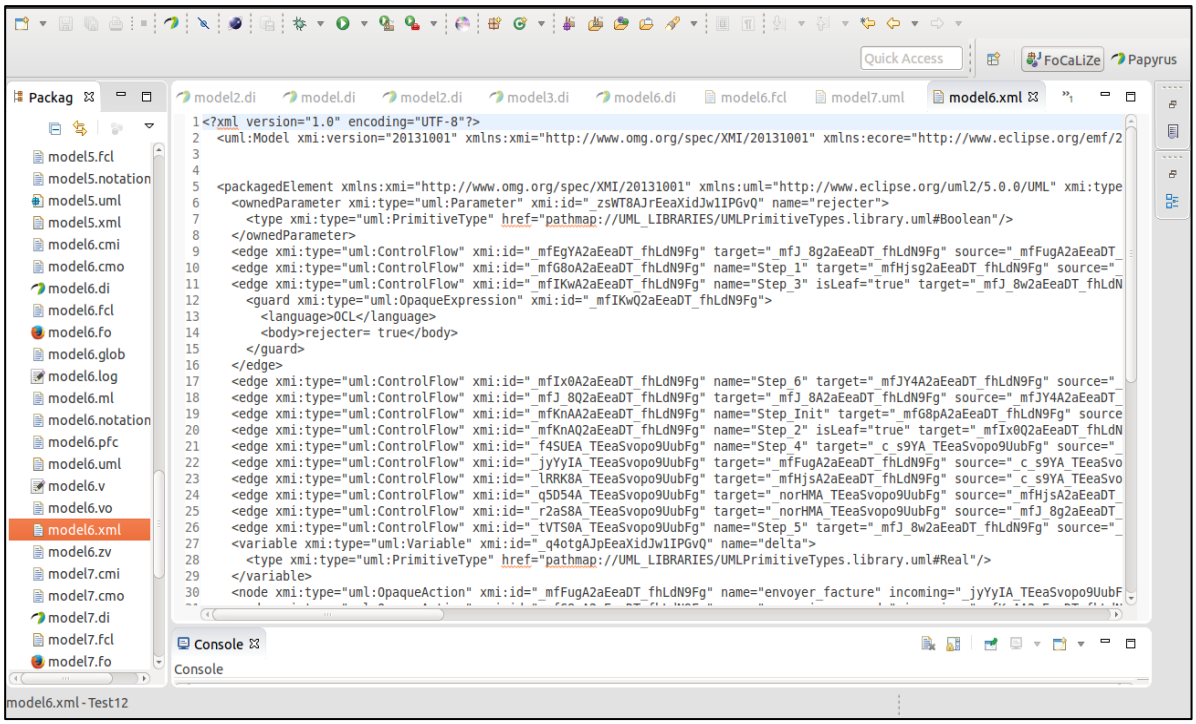


Figure V. 10: Le format XMI de diagramme d'activité "Gestion_commande"

V.4.3 Etape3 : génération de code FoCaLiZe

Le code FoCaLiZe correspondant à notre diagramme d'activité est donné comme suit :

```
open "basics" ;;
open "float_func" ;;

(* Activity Activity1*)
 type trans = | Step_0 | Step_init | Step_1 | Step_3 | Step_2
| Step_4 | Step_5 | Step_6 |  Step_finale;;

(* species Gestion_commande *)
species Gestion_commande = inherit Basic_object ;
       signature recevoir_commande : Self -> Self;
       signature remplir_commande : Self -> Self;
       signature experdier_marchandises : Self -> Self;
       signature envoyer_facture : Self -> Self;
       signature recevoir_paiement : Self -> Self;
       signature fermer_commande : Self -> Self;
       signature rejecter: Self -> bool;
       signature join_7 :  Self ->  Self ->  Self ;

let rec diagAc (s:trans, e:Self):(steps * Self) =match s with
```

```
| Step_1 -> if ( rejecter(e) = true) then
                diagAc(Step_3, e)
            else
                diagAc(Step_2, e)
| Step_3 ->diagAc(Step_6, e)
| Step_6 ->diagAc(Step_Final , fermer_commande(e))
| Step_Init ->diagAc(Step_1, recevoir_commande(e))
| Step_2 ->diagAc(Step_4, remplir_commande(e))
| Step_4 -> let   x1 = recevoir_paiement(envoyer_facture(e))
             and  x2 = expedier_marchandises(e)
             in diagAc(Step_5 , join_7(x1 , x2))

| Step_5 ->diagAc(Step_6, e)
| Step_Final ->diagAc(Step_0, e)
| Step_0 ->focalize_error("Fin de traitement") ;
end;;
```

Table V. 2: Le code FoCaLiZe généré

V.4.4 Etape4 : la compilation de code FoCaLiZe généré

Dans ce cas, La message de compilation désigne que n'existe pas des erreurs dans le code FoCaLiZe (voir table V.3).

```
Invoking ocamlc...
>>ocamlc -I /usr/local/lib/focalize -c
/home/noura/workspace/Test12/model6.ml
Invoking zvtov...
>>zvtov -zenonzenon -new  /home/noura/workspace/Test12/model6.zv
Invoking coqc...
>>coqc  -I /usr/local/lib/focalize  -I /usr/local/lib
/home/noura/workspace/Test12/model6.v
```

Table V. 3: Compilation de code FoCaLiZe

Dans le cas où le compilateur FoCaLiZe trouve des erreurs dans le code, il indique les lignes responsables de ces erreurs.

V.5 Conclusion

Dans ce chapitre, nous avons présenté les étapes de génération d'une spécification FoCaLiZe correspondant à un diagramme d'activité UML.

Dans ce but, nous avons abouti à une implémentation de la démarche de transformation proposée en utilisant le langage XSLT sous l'environnement Eclipse.

Ainsi, sous le même environnement (Eclipse), il est possible de construire un modèle UML/OCL en utilisant un outil UML/OCL graphique (Papyrus), puis en appelant les règles de transformation de générer le code FoCaLiZe correspondant. Toujours sous même l'environnement, on peut procéder à la compilation du code pour la vérification et la détection d'éventuelles incohérences. En cas d'erreur, l'outil proposé assiste l'utilisateur en indiquant la ligne de code FoCaLiZe responsable de l'erreur.

Conclusion générale

Dans ce travail de mémoire, nous avons proposés une démarche de transformation automatique des diagrammes d'activité d'UML en spécifications FoCaLiZe. Pour atteindre cet objectif, nous avons réalisé notre démarche en deux étapes :

- Dans la première étape, nous avons défini les règles de transformation d'un diagramme d'activité en spécification FoCaLiZe. Dans la formalisation proposée, le diagramme d'activité d'une classe est formalisé par la définition d'une fonction récursive, en utilisant le mécanisme de filtrage en FoCaLiZe. Chaque pattern correspondra au traitement d'une action du diagramme d'activité.
- Dans La deuxième étape nous avons mis en œuvre notre démarche de transformation. Pour cela, nous utilisons le langage XSLT. Nous avons défini une feuille de style XSLT, qui met en œuvre les règles de transformation à partir d'un document XMI et permettre la génération du code FoCaLiZe.

La démarche de transformation proposée permet une formalisation rigoureuse du diagramme d'activité d'une classe, même si cette dernière est créée par héritage ou paramétrage. Notre démarche permet aussi de supporter naturellement les partitions, la communication des diagrammes d'activité de plusieurs classes.

Comme perspectives, nous envisageons à compléter notre transformation pour considérer des fonctionnalités comme les partitions et les activités, afin d'arriver à une transformation complète des diagrammes d'activité.

Une autre perspective consiste à proposer des mécanismes pour la vérification des propriétés d'un diagramme d'activité UML, à base de notre démarche de transformation. Ceci peut être éteint en complétant le code généré par la définition de toutes ses signatures et la prouve de toutes ses propriétés.

Bibliographies

[**Abb+14**]: Messaoud Abbas, Choukri-Bey Ben-Yelles, and Renaud Rioboo. Modeling UML Template Classes with FoCaLiZe. In The International Conference on Integrated Formal Methods (IFM 2014, Bertinoro Italy). Published in LNCS, volume 56, pages 87–102.

[**Abb14**]: Messaoud Abbas, Choukri-Bey Ben-Yelles, and Renaud Rioboo. Generating FoCaLiZe Specifications from UML Models. In The International Conference on Advanced Aspects of Software Engineering (ICAASE 2014, Constantine Algeria). Conference proceedings, pages 157–164.

[**ABGR07**] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy :A challenging model transformation. In Model Driven Engineering Languages and Systems, pages 436_450. Springer, 2007.

[**Abr96**] : J.R. Abrial. The B book. 1996. Cambridge University Press.

[AD]: *"Diagramme d'activité"*, adresse du document http://saoudyihab.voila.net/cours_uml/Diagramme_d_activite.pdf **.**

[**AHM+07**] :Ahlem and Leila. "Using UML Activity Diagrams and Event B for Distributed and Parallel Applications" .Article .Research Unit of Technologies of Information and Communication (UTIC) - ESSTT6Tunisia Faculty of Science of Tunis .2007.

[**AHM+11**] :Ahlem and Leila. "A Formalisation of UML AD Refinement Patterns in Event B". Article. Reearche Unit of Technologies of Information and Communication (UTIC)-ESSTT6Tunisia Faculty of Science of Tunis .2011.

[**AHM+14**]: Ahlamand Yousre ; Leila . "A Méta-Model Transformation from UML Activity Diagrams to Event-B Models". Article .Laboratory LaTICE-University of Tunisia / ENSI. University of Manouba .Tunisia .2014.

[**AHM12**] : Ahmed Mekki. " Contribution à la Spécification et à la Vérification des Exigences Temporelles : Proposition d'une extension de SRS d'ERTMS niveau 2 ". Thèse de Doctorat. ECOLA CENTRALE DE LILLE .18/04/2012 .

[**BOO91**]Grady Booch. Object-oriented design with applications benjamin. Cummings, Redwood City (CA), 1991.

[**CDE+07**]: Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. All about maude-a high-performance logical

framework: how to specify, program and verify systems in rewriting logic . Springer-Verlag, 2007.

[**DAM11**]: Damien Doligez, Mathieu Jaume, Renaud Rioboo, "Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment A case study within the FoCaLiZe environment", article,2011.

[**DAV10**]: David Delahaye, Catherine Dubois, Pierre-Nicolas Tollitte,"Génération de code fonctionnel cérite à partir de spécifications inductives dans l'environnement Focalize", article, 2010.

[**DEHIMI**] : DEHIMI NARDJESS TISSILIA. " Un Cadre Formel pour la Modélisation et L'analyse Des Agents Mobilités". Thèse de Doctorat. Université Mentouri de Constantine.

[**DEV**]:http://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-activitesmars 2016-03-08.

[**FGK+96**] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and MihaelaSighireanu. Cadp a protocol validation and veri_cationtoolbox.In Computer AidedVeri_cation, pages 437_440. Springer, 1996.

[**GF10**] : GUERROUF FAYÇAL. " Une Approche de Transformation des Diagrammes d'Activités d'UML Mobile 2.0 vers les Réseaux de Pétri ". Mémoire de Magister. Université El Hadj Lakhdar – BATNA. 2010

[**GPCR12**]: Ana Garis, Ana CR Paiva, Alcino Cunha, and Daniel Riesco. Specifying UML protocol state machines in Alloy. In Integrated Formal Methods, pages 312_326. Springer, 2012.

[**GV13**]: Claude Girault and Rüdiger Valk. Petri nets for systems engineering : a guide to modeling, veri_cation, and applications. Springer Science & Business Media, 2013.

[**HARALD**]: Harald Storrle and Jan Hendrik Hausmann. "Transformation des diagrammes d'activités -SysML1.2 vers les réseaux de Petri dans un cadre MDE". Groupe Ingénierie Système et Intégration. Université Paul Sabatier - Laboratoire d'Analyse et d'Architecture des Systèmes.

[**HIB13**] : Hiba HACHICHI. " Test formel des systèmes temps réel : Approche de transformation de graphes ". Thèse de Doctorat. Université Mentouri de Constantine .17/03/2013 .

[**HOU10**] : Houda HAMROUCHE. " Une Approche de transformation des diagrammes D'activité d'UML vers CSP basée sur la transformation de graphes ". Mémoire de Magister. UNIVERSITE 20 AOUT 1955 – SKIKDA .2010.

[**Jac04**]: Jackson D.: Alloy 3.0 Reference Manual. http://alloy.mit.edu/. Software Design Group, Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, 2004.

[**JAN+14**] :Jan Czopik. Michael Alexander. Jakub Stolfa. And SvatoplukStolfa. "Formalisation of Software Process Using Intuitive Mapping of UML Activity Diagram to CPN ".Article. Technique University of Ostrava.2014.

[**JCC94**]: Ivar Jacobson, Magnus Christerson, and Larry L Constantine. The oose method: à useâcase-drivenapproach. In Object development methods, pages 247_270. SIGS Publications, Inc., 1994.

[**JEA14**] : Jean-Christophe Bach. " Un ilot formel pour les transformations de modèles qualifiables". Thèse de Doctorat. Université de Lorraine .12/09/2014 .

[**KAR14**] : Karima BERRAMLA. " Vérification et Validation des transformations de Modèles". Mémoire de Magister. UNIVERSITE D'ORAN .02/07/2014.

[**Kha+15]** : GHENDIR MABROUK Nacira and MENACEUR Khadija. "Automatic transformation tool of UML class diagrams into FoCaLiZe". Thése Master. UNIVERSITY OF ECHAHID HAMMA LAKHDAR EL OUED. 2015.

[**LOT02**]: Christophe Lohr. "Contribution à la conception de systèmes temps-réel s'appuyant sur la technique de description formelle RT-Lotos ". Thèse Doctorat. Institut National Polytechnique de Toulouse. 2002.

[**LOT05**] : Pierre De Saqui-Sannes. " Conception basée modèle des systèmes temps réel et Distribués ". Thèse Doctorat. Institut National Polytechnique de Toulouse. 7 juillet 2005.

[MICRO]: https://msdn.microsoft.com/fr-fr/library/dd409360.aspx mars 2016-03-04

[**MOU13**] : Mouna Bouarioua. " Une approche basée transformation de graphes pour la génération de modèles de réseaux de Pétri analysables à partir de diagrammes UML". Thèse de doctorat. UNIVERSITÉ CONSTANTINE 2. 2013

[**Mur89**] Tadao Murata. Petri nets : Properties, analysis and applications. Proceedings of the IEEE, 77(4):541_580, 1989

[OMG09]: *"OMG Unified Modeling Language TM (OMG UML), Superstructure", Février 2016,* URL Standard du document: http://www.omg.org/spec/UML/2.2/Superstructure.

[**OMG12**]: OMG. OCL: Object constraint language 2.3.1. January 2012. Available at: http://www.omg.org/spec/OCL.

[Perochon09] : L.Perochon, *"UML : langage graphique de modélisation", Février 2016,* http://www.projet-plume.org/ressource/uml(INRA)

[**PHI11**]: Philippe AYRAULT, "Développement de logiciel critique en FoCaLiZe méthodologie et outils pour l'évaluation de conformité", Thèse de doctorat, Université Pierre et Marie Curie - Paris 6, 22 Avril 2011.

[**PIE13**]: Pierre-Nicolas TOLLITTE, "Extraction de code fonctionnel certifié à partir de spécifications inductives", Thèse de doctorat, 6 décembre 2013.

[**RBL+90**]: James R Rumbaugh, Michael R Blaha, William Lorensen, Frederick Eddy, and WilliamPremerlani. Object-oriented modeling and design. 1990.

[RUML] : "Résumé du sous-ensemble de la notation UML 2utilisé dans ce livre ", adresse du document: https://www.google.dz/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0ahUKEwjomOqf16fLAhVHWRoKHe9BB5cQFggrMAI&url=http%3A%2F%2Fizibook.eyrolles.com%2Fextract%2Fshow%2F8382&usg=AFQjCNHySgL5WFXVTRdFoh2Rnwy2BFJKZA&sig2=YxyD72uQ6qDka6CP3juYQg mars 2016-03-08.

[**SAG10**] : Sagar Sen. "Découverte automatique de modèles effectifs". Thèse Doctorat .UNIVERSITÉ DE RENNES 1 sous le sceau de l'Université Européenne de Bretagne. 2010.

[**SAN10**] : Sana Jabri. " Génération de scénarios de tests pour la vérification de systèmes répartis : application au système européen de signalisation ferroviaire(ERTMS) ". Thèse de Doctorat. ECOLA CENTRALE DE LILLE .22/06/2010 .

[**SOP00**] : Sophie DUPUY. " Coulage de notations semi-formells et formelles pour la spécification des systèmes d'information ". Thèse de Doctorat. UNIVERSITE JOSEPH FOURIER-GRENOBLE I .22/09/2000 .

[**UML08**]: J. Gabay, D. Gabay. ' UML 2 ANALYSE ET CONCEPTION '.Dunod - Paris. 2008.

[UML2B]: http://download.gna.org/brillant/docs/2005-Szyndler/CDuce-UML2B.html mars 2016-03-04.

[**YOA15**] : Yoann Laurent. "Alloy4PV : un Framework pour la Vérification de Procèdes Métiers". Thèse Doctorat. L'Université Pierre & Marie Curie. 2015.

Glossaire

A

ATL : ATLAS Transformation Langage ;

ATOM3: A Tool for Multi-formalism and Meta-Modelling

C

CSP: Calculus of Sequential Processes;

CADP : Construction and Analysis of Distributed Processes ;

E

EMF : Eclipse Modeling Framework,

F

FDR : Failures-Divergences Refinement ;

G

GMF: Graphical Modeling Framework ;

L

LOTOS: Language Of Temporal Ordered Systems;

LTL: Linear Temporal Logic;

O

OCL: Object Constraint Language;

OMG: Object Management Group;

OMT : *Object Modeling Technique* ;

OOSE: *Object Oriented Software Engineering* ;

M

MDE: Model Driven Engineering;

R

RdP: Réseau de Petri ;

T

TINA : TImepétri Net Analyser ;

TOPCASED: Toolkit in Open Source for Critical Applications &Systems Development ;

U

UML: Unified Modeling Language;

UML AD: UML Diagram Activity;

W

W3C: World Wide Web Consortium;

X

XML: eXtensible Markup Language ;

XMI : XML Meta data Interchange ;

XSLT: eXtensible Stylesheet Language Transformations ;

Annexe

Installation des outils de développement

L'installation de notre plugin sera mis en place 2 composants:

- Eclipse Environnement avec Papyrus plugin.
- compilateur Focalize.

Pour utilise notre outil de transformation il faut installer les softwares suivantes :

1. installation de plugin papyrus

Pour installer le plugin Papyrus voir le site Web suivant:

https://wiki.eclipse.org/Papyrus-RT/User_Guide/Installation

2. installation de focalize

L'installation de Focalize se fait par ce lien :

http://FoCaLiZe.inria.fr/download/

Il requiert l'installation des outils externes suivants dans le racine de répertoire de foCalize (habituellement "Focalize").

- **OCaml** (toute version récente -> = 3.11 -) doit être fine.

Installez #sudo apt-get install ocaml

- **Coq** (toute version récente -> = 8.1pl5 -) doit être fine.

Installez #sudo apt-get install coq

- **Zenon** (soit télécharger une archive ou l'obtenir à partir du dépôt GIT par Invoquer.

Getclone http://FoCaLiZe.inria.fr/zenon.git

3. Installation de plugin de transformation (UmlToFocalize)

Pour les consignes d'installation, suivi les points suivants :

- Fermer l'environnement eclipse ;
- Ajouter plugin 'umlToFoCaLiZe' dans path 'eclipse / plugins /' ;
- Ajouter des fichiers XSLT ('TrieUMLOrginal.xsl' et 'umlTofoclize.xsl') dans le chemin 'eclipse /' ;
- Ouvre l'eclipse.