N° d'ordre:

N° de série:

République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

---

UNIVERSITE ECHAHID HAMMA LAKHDAR EL OUED

FACULTE DES SCIENCES EXACTES

DEPARTEMENT D'INFORMATIQUE

Mémoire de fin d'étude

MASTER ACADEMIQUE

Domaine : Mathématique et Informatique

Filière : Informatique

Spécialité : Systèmes Distribués et Intelligence Artificielle (SDIA)

Thème

Transformation de diagramme d'états-transitions UML vers FoCaLiZe

Présenté par

KHECHIM MADI Abdelaziz KHECHIM-MADI Houda

Soutenu devant le jury composé de

M.	Abbas Messaoud	Rapporteur	Univ.d'ElOued
M.	Boucherit Ammar	Président	Univ.d'ElOued
M.	Yakoub Med Amine	Examinatrice	Univ.d'ElOued

Année universitaire 2016/2017

Remerciements

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

Nous voudrions remercier notre encadreur **Mr ABBAS Messaoud** qui nous a aidé et nous a orienté tout au long de la préparation de ce mémoire. Il est toujours présent pour clarifier certains aspects de nos lacunes et de discuter de nos pensées. Ses critiques, commentaires et conseils ont certainement permis à cette mémoire d'être comme ce qu'elle est.

Nous remercions sincèrement ceux qui ont bien voulu prendre part à ce jury.

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

Khechim madi Abdelaziz & Khechim-madi Houda

Dédicaces

Je dédie ce travail

A

Mon père

A

Ma chère mère

A

Ma chère Houda

A

Ma petite soeur Fatima Elzahra

A

Mes frères

A

Tous mes amis

ABDELAZIZ

Dédicaces

Je dédie ce modeste travail

A

Mes très chers parents

A

Mes grands-parents

A

Mon cher mari Abdelaziz

A

Ma chère sœur Safa

A

Mes frères Soufian et Bachir

A

Mes amis Chaima, Aldjia, Imane, Wahiba, Saida, Khadidja.

HOUDA

Résumé

UML est l'un des langages les plus répandus pour la conception des applications informatiques. Il étant un langage semi-formel, le principal reproche d'UML est l'absence de bases formelles permettant l'application des techniques de vérifications formelles. Il ne dispose d'aucuns outils pour la vérification et la preuve des propriétés de ces modèles.

L'approche la plus largement adoptée est la transformation d'un modèle UML vers une spécification formelle, en utilisant des méthodes formelles comme FoCaLiZe. Cette approche rentre dans le cadre d'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.

FoCaLiZe est un environnement complet pour exprimer tous ces aspects du développement logiciel.

Dans cette mémoire, nous proposons une approche MDE pour la transformation automatique des diagrammes d'états-transitions UML en spécifications FoCaLiZe, puis une implémentation des règles de transformation en utilisant le langage XSLT qui nous permet de générer une spécification FoCaLiZe à partir d'un document XMI (XML Meta data Interchange).

Mots clés : diagrammes d'états-transitions d'UML, Transformation, FoCaLiZe, MDE, XSLT, XMI.

Abstract

UML is one of the most common languages for the design of computer applications. As it is a semi-formal language, the main criticism that can be made is that there is no formal basis for the application of formal verification techniques. It does not have tools for verification and proof of the properties of these models.

The most widely adopted approach is the transformation of a UML model into a formal specification, using formal methods such as FoCaLiZe. This approach is part of the Model Driven Engineering (MDE), which aims to produce softwares through automatic model refinements, from abstract specifications to concrete implementations.

FoCaLiZe is a development environment based on a formal approach, which integrates an automated theorem prover (Zenon) and a proof checker (Coq).

In this thesis we propose an MDE approach for automatically transforming state transitions diagrams from UML into FoCaLiZe specifications and then proposing an implementation of transformation rules using the XSLT language which allows us to generate a FoCaLiZe specification from An XMI (XML Meta data Interchange) document.

Keywords: UML statechart diagram, Transformation, FoCaLiZe, MDE, XSLT, XMI.

ملخص

UML هي واحدة من اللغات الأكثر شعبية بالنسبة للتصميم تطبيقات الحاسوب. UML هي لغة شبه تحققية، والنقد الرئيسي الذي يمكن القيام به هو عدم وجود أساس رسمي لتطبيق تقنيات التحقق الرسمية, وليس لديه أدوات للتحقق وإثبات خصائص نموذج UML.

النهج الأكثر اعتمادا هو التحول من نموذج UML إلى مواصفات رسمية، وذلك باستخدام الطرق الرسمية مثل فوكاليز(Focalize). يقع هذا النهج في إطار MDE ("Model Driven Engineering ")، وهو إنتاج برامج لصقل التلقائي للنماذج، من الناحية النظرية للمواصفات الى تطبيقات ملموسة.

فوكالايز(FoCaLiZe) هو بيئة تطوير ترتكز على المنهج النظامي، والذي يشتمل على دمج المبرهن الآلي(Zenon) والمدقق للبراهين (Coq).

في هذه المذكرة، نقترح طريقة MDE ("Model Driven Engineering") للتحويل التلقائي للرسم البياني الحالات- تحولات الخاص ب UML وفق مواصفات FoCaLiZe. حيث إقترحنا تطبيق قواعد التحويل بستخدام لغة XSLT التي تسمح لنا بتوليد مواصفات Focalize انطلاقا من ملف XMI.

الكلمات المفتاحية: UML, FoCaLiZe, Transformation, XSLT, MDE.

Sommaire

Listes des figures

Liste des tableaux

Introduction générale

Introduction générale

Aujourd'hui avec le développement industriel, les systèmes développés sont devenus très volumineux, plus compliquées qui nécessitent un degré élevé de sécurité, ce qui rend le contrôle et la maintenance des systèmes très difficile.

Le standard de l'industrie pour le développement de systèmes UML (Unified Modelling Language) permet la modélisation et la conception d'un système en utilisant un ensemble de diagrammes graphiques **[3]**. Il définit des diagrammes structurels et comportementaux pour représenter respectivement des vues statiques et dynamiques d'un système **[4]**. Ceci permet de mieux gérer la complexité du système.

Mais, UML est un langage semi-formel parce qu'il manque les bases formelles nécessaires pour faire la vérification des modèles (modèles UML) et il ne dispose pas de moyens pour découvrir les défauts de modélisation. Pour remédier cette faiblesse, il est pertinent d'utiliser des techniques et des mécanismes formels permettant l'analyse et la vérification des modèles. Parmi les outils les plus adoptés dans la formalisation des modèles UML, les méthodes formelles.

Les méthodes formelles fournissent des notations mathématiques permettant à guider le programmeur pour l'amener à donner, puis prouver les propriétés nécessaires à la cohérence de son modèle. Alors le modèle est moins intuitif par rapport à UML.

Dans ce contexte, Plusieurs travaux se sont intéressés à la transformation d'un modèle UML vers une spécification formelle, en utilisant des méthodes formelles comme B **[17]**, Alloy **[15]**, Maude **[20],** FoCaLiZe **[23, 24, 25]**… etc.

Dans cette mémoire, nous proposons une approche de transformation du diagramme d'états-transitions UML vers FoCaLiZe dans le contexte de MDE (Model Driven Engineering). Les techniques de ce dernier peuvent être utilisées pour générer du code à partir de ces modèles.

FoCaLiZe est une méthode formelle, initiée à la fin des années 90 par Thérèse Hardin et Renaud Rioboo au sein des laboratoires LIP6, CÉDRIC et INRIA. Nous choisissons FoCaLiZe parce qu'il fournit un environnement complet de la spécification jusqu'à l'implémentation.

Notre transformation est basée sur la transformation de classe qui sont traitées dans **[12, 23]** et utiliser un ensemble des règles de transformation que nous définissons, pour transformer les différents composants de diagramme d'états-transition.

Pour mettre en œuvre notre approche de transformation, nous avons utilisé un environnement Eclipse avec plugin Papyrus pour créer notre modèle d'UML (diagramme d'états-transition) et générer son format XMI et nous avons développé une feuille de style XSLT afin d'implémenter les règles de transformation pour générer le code FoCaLiZe.

Le plan du mémoire est le suivant :

- **Chapitre I:** Présentation du langage UML, décrit les diagrammes d'UML, le diagramme de classe, les diagrammes d'états-transitions, leurs concepts de base que nous avons utilisés dans notre démarche de transformation.
- **Chapitre II** : Présentation de concepts de base de l'environnement FoCaLiZe.
- **Chapitre III** : Synthétisation des travaux connexes liés à la transformation de modèle UML vers FoCaLiZe.
- **Chapitre IV** : Proposition de notre transformation des diagrammes d'états-transitions vers FoCaLiZe.
- **Chapitre V** : Présentation l'implémentation de notre démarche de transformation.

CHAPITRE I

Les diagrammes d'états-transitions

1. Introduction
2. Diagrammes UML
3. Digramme de classe
4. Diagramme états-transitions

1. Introduction

UML, se définit comme un langage de modélisation graphique et textuel destiné à comprendre et décrire des besoins, spécifier et documenter des systèmes **[1]**.

UML est un formalisme visuel pour la modélisation orientée objet, défini par l'OMG[1] (Object Management Group).

Le langage UML est né de la mise en commun des trois plus importantes méthodes de modélisation orientées objet **[2]**:

- La méthode Booch dont l'auteur, Grady Booch, travaillait au développement de systèmes en Rational Software Corporation.
- La méthode Object Modeling Technique (OMT) développée par Jim Rumbaugh qui dirigeait une équipe de recherche chez General Electric.
- La méthode Object-Oriented Software Engineering (OOSE) résultant des travaux d'Ivar Jacobson sur les commutateurs téléphoniques chez son employeur Ericsson. La méthode OOSE était la première à introduire le concept des cas d'utilisation (use case). La figure 1.1 expose l'histoire d'UML.

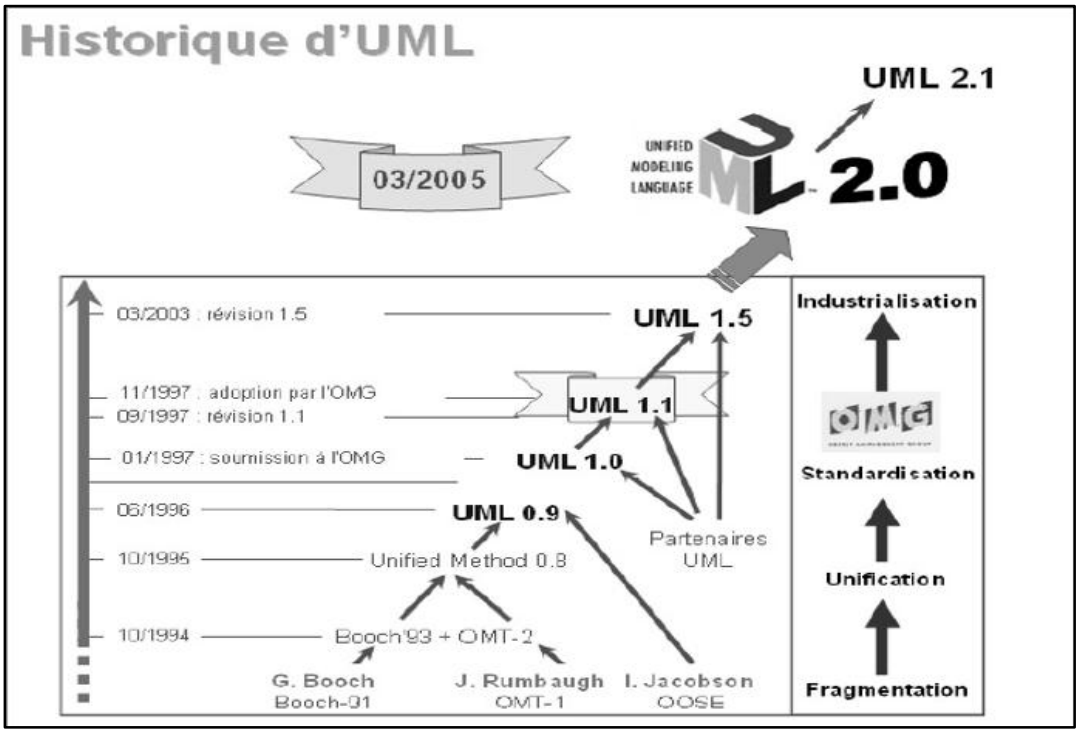


Figure I. 1: Historique d'UML **[1]**.

[1] http://www.omg.org/

Le langage UML **[3]** permet la modélisation et la conception d'un système en utilisant un ensemble de diagrammes graphiques. Certains diagrammes UML permettent de modéliser l'aspect statique d'un système et d'autres diagrammes servent à la modélisation de l'aspect dynamique (aspect comportemental).

Dans ce contexte orienté objet, la modélisation du comportement des objets est la partie qui retient notre attention. En UML le comportement des objets est modélisé notamment par les diagrammes d'états-transitions.

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 aux diagrammes d'états-transitions, qui sont les seuls pris en compte lors de la transformation d'UML vers FoCaLiZe.

2. Diagrammes UML

Un diagramme donne à l'utilisateur un moyen de visualiser et de manipuler des éléments de modélisation **[4]**.

UML définit des diagrammes structurels et comportementaux pour représenter respectivement des vues statiques et dynamiques d'un système **[4]**. Les diagrammes incluent des éléments graphiques qui décrivent le contenu des vues **[5].**

UML 2 s'articule autour de treize types de diagrammes, chacun d'eux étant dédié à la représentation des concepts particuliers d'un système logiciel. Ces types de diagrammes sont répartis en deux grands groupes **[1]**:

- Six diagrammes structurels :
  - **Diagramme de classes :** Il montre les briques de base statiques : classes, associations, interfaces, attributs, opérations, généralisations, etc.
  - **Diagramme d'objets :** Il montre les instances des éléments structurels et leurs liens à l'exécution.
  - **Diagramme de packages :** Il montre l'organisation logique du modèle et les relations entre packages.
  - **Diagramme de structure composite :** Il montre l'organisation interne d'un élément statique complexe.
  - **Diagramme de composants :** Il montre des structures complexes, avec leurs interfaces fournies et requises.

- **Diagramme de déploiement :** Il montre le déploiement physique des « artefacts » sur les ressources matérielles.

➢ Sept diagrammes comportementaux :

- **Diagramme de cas d'utilisation :** Il montre les interactions fonctionnelles entre les acteurs et le système à l'étude.
- **Diagramme de vue d'ensemble des interactions :** Il fusionne les diagrammes d'activité et de séquence pour combiner des fragments d'interaction avec des décisions et des flots.
- **Diagramme de séquence :** Il montre la séquence verticale des messages passés entre objets au sein d'une interaction.
- **Diagramme de communication :** Il montre la communication entre objets dans le plan au sein d'une interaction.
- **Diagramme de temps :** Il fusionne les diagrammes d'états et de séquence pour montrer l'évolution de l'état d'un objet au cours du temps.
- **Diagramme d'activité :** Il montre l'enchaînement des actions et décisions au sein d'une activité.
- **Diagramme d'états-transitions:** Il montre les différents états et les transitions possibles des objets d'une classe à l'exécution.

3. Diagramme de classe

Les diagrammes de classes expriment de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes **[4]**.

Le diagramme de classes est le point central dans un développement orienté objet. Au niveau analyse, il a pour objet de décrire la structure des entités manipulées par les utilisateurs. Au niveau conception, le diagramme de classes représente la structure d'un code orienté objet **[1]**.

Les classes sont représentées par des rectangles compartimentés. Le premier compartiment contient le nom de la classe. Le nom de la classe doit permettre de comprendre ce que la classe est, et non ce qu'elle fait. Une classe n'est pas une fonction, une classe est une description abstraite – condensée – d'un ensemble d'objets du domaine de l'application. Les deux autres compartiments contiennent respectivement les attributs et les opérations de la classe **[4]**.

3.1 Les concepts de diagrammes de classes

Le diagramme de classe comporte les concepts **[1]**:

- **Classe et objet :** une classe représente la description abstraite d'un ensemble d'objets possédant les mêmes caractéristiques.

  Un objet est une entité aux frontières bien définies, possédant une identité et encapsulant un état et un comportement. C'est une instance (ou une occurrence) d'une classe.
- **Attribut et opération :** un attribut représente un type d'information contenu dans une classe. Une opération représente un élément de comportement contenu dans une classe. Nous ajouterons plutôt les opérations en conception objet.
- **Visibilité des attributs et des opérations :** UML définit trois niveaux de visibilité pour les attributs et les opérations :
  - Public qui rend l'élément visible à tous les clients de la classe.
  - Protégé qui rend l'élément visible aux sous- classe de la classe.
  - Privé qui rend l'élément visible à la classe seule **[4]**.
- **Association :** une association représente une relation sémantique durable entre deux classes ou plusieurs.
- **Multiplicité :** Elle spécifie sous la forme d'un intervalle le nombre d'objets qui peuvent participer à une relation avec un objet de l'autre classe dans le cadre d'une association.
- **Agrégation et composition :** une agrégation est un cas particulier d'association non symétrique exprimant une relation de contenance. Les agrégations n'ont pas besoin d'être nommés : implicitement elles signifient « contient » ou « est composé de ». Une composition est une agrégation plus forte impliquant que :
  - Un élément ne peut appartenir qu'à un seul agrégat composite (agrégation non partagé).
  - La destruction de l'agrégat composite entraine la destruction de tous ses éléments (le composite est responsable du cycle de vie des parties).
- **Classe d'association :** Il s'agit d'une association promue au rang de classe. Elle possède tout à la fois les caractéristiques d'une association et celles d'une classe et peut donc porter des attributs qui se valorisent pour chaque lien.
- **Généralisation, superclasse, sous classe :** Une superclasse est une classe plus générale reliée à une ou plusieurs autres classes plus spécialisées (sous classe) par une

relation de généralisation. Les sous classes « héritent » des propriétés de leur superclasse et peuvent comporter des propriétés spécifiques supplémentaires.

- **Package :** c'est un mécanisme général de regroupement d'éléments en UML, qui est principalement utilisé en analyse et en conception d'objet pour regrouper des classes et des associations. Les packages sont des espaces de noms : deux éléments ne peuvent pas porter le même nom au sein de même package. Par contre deux éléments appartenant à des packages différents peuvent porter le même nom.

La structuration d'un modèle en packages est une activité délicate. Elle doit s'appuyer sur deux principes: cohérence et indépendance.

- **Cohérence :** consiste à regrouper les classes qui sont proches d'un point de vue sémantique. Un critère intéressant consiste à évaluer les durées de vie des instances de concepts et à rechercher l'homogénéité.
- **Indépendance :** s'efforce de minimiser les relations entre packages c'est-à-dire plus concrètement les relations entre classe de packages différents.

3.2 Intérêts des diagrammes de classes

Dans UML, les diagrammes de classes constituent la vue logique (car normalement indépendante du langage de programmation cible du système).

Les intérêts des diagrammes de classes sont multiples:

- Rassembler les données utilisées par le système dans des entités encapsulées : les classes (collections d'objets d'attribut communs).
- Définir les opérations qui permettent de manipuler ces données, celles-ci seront intégrées aux classes.
- De réaliser une vision des éléments statiques du système, c'est-à-dire de recenser les parties des structures qui ne se modifieront pas avec le temps.
- Mettre en œuvre les concepts objets (en particulier l'héritage qui permet la réutilisation du code).

4. Diagramme d'états-transitions

UML a repris le concept bien connu de machine à états finis, qui consiste à s'intéresser au cycle de vie d'un objet générique d'une classe particulière au fil de ses interactions, dans tous les cas possibles. Cette vue locale d'un objet, qui décrit comment il réagit à des événements en fonction de son état courant et comment il passe dans un nouvel état, est représentée graphiquement sous la forme d'un diagramme d'états **[1]**.

Le diagramme d'états-transitions est l'un des diagrammes qui décrit le comportement dynamique des objets dans le temps en modélisant les cycles de vie des objets de chaque classe, mais il décrit aussi le comportement dynamique des cas d'utilisation, les collaborations et les méthodes **[6]**.

4.1 Intérêt des diagrammes d'états-transitions

Les intérêts de la modélisation par les diagrammes d'états-transitions sont:

- Donner dynamicité aux objets (s'ils s'y prêtent), représentés jusqu'à présent de manière statique comme des occurrences de classes qu'on peut généraliser par les classes.
- Visualiser le système en diminuant sa complexité.
- Tenir compte des états lors de l'implémentation (en effet la traduction des états peut être faite simplement, la plupart de langages le permettent).
- Présenter un aspect du modèle dynamique, l'autre étant illustré par les diagrammes d'activités, les diagrammes de collaboration et les diagrammes de séquence.
- Pouvoir décrire les changements d'états des automates.

4.2 Les concepts des diagrammes d'états-transitions

Les diagrammes d'états-transitions visualisent des automates d'états finis, du point de vue des états et des transitions.

Le comportement des objets d'une classe peut être décrit de manière formelle en termes d'états et des transitions, au moyen d'un automate relié à la classe considérée **[8]**.

Un automate est une abstraction des comportements possibles, à l'image des diagrammes de classes qui sont des abstractions de la structure statique. Chaque objet suit globalement le comportement décrit dans l'automate associé à sa classe et se trouve à un moment donné dans un état qui caractérise ses conditions dynamiques **[8]**.

Le diagramme d'états-transitions manipule plusieurs concepts :

4.2.1 Les états

Chaque objet est à un moment donné dans un état particulier. Un état représente une condition ou situation qui se produit dans la vie d'un objet pendant laquelle il satisfait une certaine condition, exécute une activité particulière ou attend certains événements **[1]**.

- Les états se représentent sous la forme de rectangles arrondis; chaque état possède un nom qui l'identifie.
- Les états se caractérisent par la notion de durée et de stabilité. Un objet est toujours dans un état donné pour un certain temps et un objet ne peut être dans un état inconnu ou non défini.
- Un état est toujours l'image de la conjonction instantanée des valeurs contenues par les attributs de l'objet, et de la présence ou non de liens, de l'objet considéré vers d'autres objets.
- Les états simples sont représentés comme suit (figure I.2) :

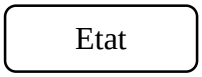


Figure I. 2: Syntaxe d'un état simple **[2]**.

Un diagramme d'états a toujours un et un seul état initial pour un niveau hiérarchique donné. Il peut n'avoir aucun état final ou plusieurs **[1]**.

- **L'état initial:** l'état initial du diagramme d'états correspond à la création de l'instance.
- **L'état final:** l'état final du diagramme d'états correspond à la destruction de l'instance.

La figure I.3 extraite de **[2]** désigne la syntaxe d'un état initial et état final.

Figure I. 3: Syntaxe d'un état initial et état final [2].

Un état comporte plusieurs parties **[7]:**

a. **Nom :** une chaine textuelle qui distingue l'état d'autres états ; un état peut être anonyme.
b. **Action d'entrée (ou de sortie) :** Les états peuvent également contenir des actions; elles sont exécutées à l'entrée ou à la sortie de l'état.

c. **Transition interne:** un état peut avoir une liste de transitions internes qui sont comme des transitions normales mais qui n'ont pas d'état cible et ne provoquent pas de changement d'état.
d. **Sous état :** si la machine d'état à une sous structure imbriquée, on l'appelle état composite. Un état composite contient une ou plusieurs régions, chacune contenant un ou plusieurs sous états directs.
e. **Evénements rapportés :** (déférés) c'est une liste d'événements qui ne sont pas traités dans cet état mais ils sont sauvegardés dans une file d'attente pour être traiter par l'objet dans d'autres états.

4.2.2 Les transitions

Lorsque les conditions dynamiques évoluent, les objets changent d'état en suivant les règles décrites dans l'automate associé à leurs classes. Les diagrammes d'états-transitions sont des graphes dirigés. Les états sont reliés par des connexions unidirectionnelles, appelées transitions. Le passage d'un état à l'autre s'effectue lorsqu'une transition est déclenchée par un événement qui survient dans le domaine du problème. Le passage d'un état à un autre est instantané car le système doit toujours être dans un état connu **[8].**

- Transition : connexion entre deux états d'un automate, qui est déclenchée par l'occurrence d'un événement, et conditionnée par une condition de garde, induisant certains effets.
- Une transition permet de passer d'un état à un autre état ; elle se représente au moyen d'une flèche qui pointe de l'état de départ(Etat source)vers l'état d'arrivée (Etat cible).

En règle générale, une transition possède cinq parties un événement déclencheur, une condition de garde, un effet, un état cible et un état source **[7]**.

a. **Etat source :** l'état source est l'état affecté par la transition. Si un objet se trouve dans l'état source, une transition sortante de l'état peut se déclencher si l'objet reçoit l'événement déclencheur de la transition de garde (soit satisfaite), l'état source devient inactif après le déclenchement de la transition.
b. **Etat cible :** l'état cible est l'état qui devient actif après l'achèvement de la transition.
c. **Effet :** une transition peut contenir un effet, c'est-à-dire une action qui s'exécute lorsque la transition se déclenche.

d. **Déclencheur d'événement :** c'est un événement reconnu par l'état source et avec lequel la transition est franchissable une fois que la garde de la transition est vérifié. Un événement peut être un signal, un appel de méthode, un passage de temps ou un changement d'état.
e. **Les gardes :** Une garde est une condition booléenne qui valide ou non le déclenchement d'une transition lors de l'occurrence d'un événement **[8]**. Elle peut concerner les attributs de l'objet concerné ainsi que les paramètres de l'événement déclencheur. Plusieurs transitions avec le même événement doivent avoir des conditions de garde différentes **[1]**.

Figure I. 4: presentation de transition

Le tableau suivant illustre les types et les effets implicites. Le tableau I.1 résume les différents types de transition et effets implicites.

Type de transition	Description	Syntaxe
Transition entry	Spécification d'une activité d'entrée qui s'exécute lorsqu'on saisit un état	**Entry/activity**
Transition exit	Spécification d'une activité de sortie qui s'exécute lorsqu'on quitte un état.	**Exit /activity**
Transition externe	Réponse à un événement qui engendre un changement d'état ou une auto transition, ainsi qu'un effet spécifié. Elle peut également entrainer l'exécution d'une activité exit et/ou d'une activité entry pour les états dont l'on sort ou dans lesquels on entre.	**(a:T)[guard]/activity**
Transition interne	Réponse à un événement qui entraine l'exécution d'un effet mais pas d'un changement d'état, ou l'exécution d'activités exit ou entry.	**e(a:T)[guard]/activity**

Tableau I. 1: Type de transition et effets implicites **[6]**.

4.2.3 Les événements

Un événement est une spécification d'une occurrence remarquable qui a une localisation dans le temps et l'espace. Un événement peut porter des paramètres qui matérialisent le flot d'information ou de données entre objets.

Un événement sert à déclencher pour passer d'un état à un autre. Les transitions indiquent les chemins dans le graphe des états. Les événements déterminent quels chemins doivent être suivis. Les événements, les transitions et les états sont indissociables dans la description du comportement dynamique. Un objet, placé dans un état donné, attend l'occurrence d'un événement pour passer dans un autre état. De ce point de vue, les objets se comportent comme des éléments passifs, contrôlés par les événements en provenance du système.

La syntaxe générale d'un événement suit la forme suivante **[8]**:

Nom_De_L_Evénement (Nom_De_Paramètre : Type, ...)

La spécification complète d'un événement comprend **[8]**:

- Le nom de l'événement.
- La liste des paramètres.
- L'objet expéditeur.
- L'objet destinataire.
- La description de la signification de l'événement.

Différentes sortes d'événements **[7]**:

a. **Evénement signal :** causé par la réception d'un signal. Un signal est un type de classeur destiné explicitement à véhiculer une communication asynchrone à sens unique entre deux objets. L'objet expéditeur crée et initialise explicitement une instance de signal et l'envoi à un objet explicite ou à tout un groupe d'objets. Il n'attend pas que le destinataire traite le signal pour poursuivre son déroulement. La réception d'un signal est un événement pour l'objet destinataire. Un même objet peut être à la fois expéditeur et destinataire. Les signaux sont déclarés par la définition d'un classeur portant le stéréotype « signal » ne fournissant pas d'opération et dont les attributs sont interprétés comme des arguments.

b. **Evénement appel :** un événement d'appel représente la réception de l'appel d'une opération par un objet. La classe destinataire choisit d'implémenter une opération comme une méthode ou comme un déclencheur d'événement d'appel dans une machine d'état (ou éventuellement les deux). Les paramètres de l'opération sont des paramètres de l'événement d'appel. Une fois que l'objet destinataire a traité l'événement d'appel en empruntant une transition déclenchée par l'événement ou pas, le contrôle retourne à l'objet appelant. Contrairement à une méthode, la transition d'une machine d'états déclenchée par un événement d'appel peut continuer de s'exécuter en parallèle de l'appelant.

c. **Un événement de changement :** est généré par la satisfaction (i.e. passage de faux à vrai) d'une expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière déclarative d'attendre qu'une condition soit satisfaite. Notez la différence entre une condition de garde et un événement de changement. La première est évaluée une fois que l'événement déclencheur de la transition a lieu et que le destinataire le traite. Si elle est fausse, la

transition ne se déclenche pas et la condition n'est pas réévaluée. Un événement de changement est évalué continuellement jusqu'à ce qu'il devienne vrai, et c'est à ce moment-là que la transition se déclenche.

d. Evénement temporel : représente le passage du temps. Ils sont spécifiés soit de manière absolue (date précise), soit de manière relative (temps écoulé). Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.

4.2.4 Effet, action, activité

Une transition peut spécifier un comportement optionnel réalisé par l'objet lorsqu'elle est déclenchée. Ce comportement est appelé « effet » en UML 2 : cela peut être une simple action ou une séquence d'actions.

- Une action peut représenter la mise à jour d'un attribut, un appel d'opération, la création ou la destruction d'un autre objet, ainsi que l'envoi d'un signal à un autre objet **[1].**
- L'action correspond à une des opérations déclarées dans la classe de l'objet destinataire de l'événement. L'action a accès aux paramètres de l'événement, ainsi qu'aux attributs de l'objet. En réalité, toute opération prend un certain temps à s'exécuter ; la notion d'action instantanée doit s'interpréter comme une opération dont le temps d'exécution est négligeable devant la dynamique du système.
- Les états peuvent également contenir des actions sont exécutée à l'entrée (**entry**) ou à la sortie (**exit**) de l'état ou lorsqu'une occurrence d'événement interne (**on**) survient (exécutée lors de l'occurrence d'un événement qui ne conduit pas à un autre état) **[8].**
- Les activités durables (**do**-activity) ont une certaine durée, sont interruptibles et sont associées aux états. Une activité, c'est-à-dire une opération qui prend un temps non négligeable et qui est exécutée pendant que l'objet est dans un état donné **[1].**

4.2.5 Etat composite

Un état composite est un état décomposé en régions contenant chacune un ou plusieurs sous-états. Quand un état composite comporte plus d'une région, il est qualifié d'état orthogonal. Lorsqu'un état orthogonal est actif, un sous-état direct de chaque région est simultanément actif, il y a donc concurrence. Un état composite ne comportant qu'une région est qualifié d'état non orthogonal. La figure I.5 désigne le formalisme de représentation d'un état orthogonal et la figure I.6 désigne le formalisme de représentation d'un état non orthogonal.

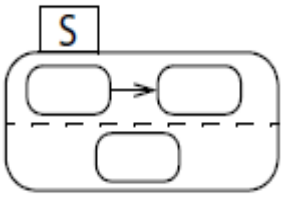


Figure I. 5: Représentation graphique d'un état orthogonal **[6]**.

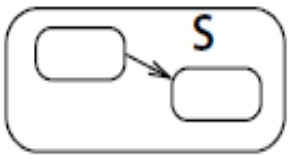


Figure I. 6: Représentation graphique d'un état non orthogonal **[6]**.

4.2.6 Etat historique

Un état historique est un pseudo-état qui mémorise le dernier sous-état actif d'un état composite. Graphiquement, il est représenté par un petit cercle contenant la lettre H. La figure I.7 désigne le formalisme de représentation d'un état historique **[2]**.

Figure I. 7: Représentation graphique d'un état historique [2].

4.2.7 Point de choix

Il est possible de représenter des alternatives pour le franchissement d'une transition. On utilise pour cela des pseudo-états particuliers : les points de jonction et les points de décision.

a. Point de jonction :

Pseudo–état qui relie des segments de transition entrante en une seule transition sortante, Ou il peut être utilisé pour diviser une transition entrante en plusieurs segments de transition sortants avec des contraintes de garde différentes **[2]**.

La figure I.8 désigne le formalisme de représentation de point de jonction.

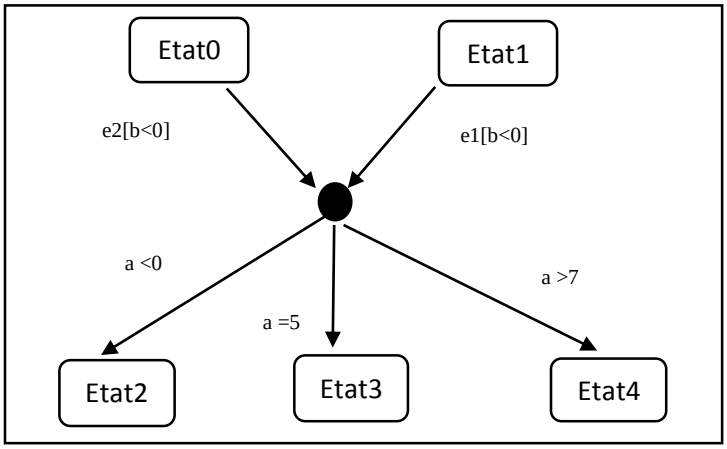


Figure I. 8: Représentation graphique de point de jonction **[2]**

b. Point de décision:

Pseudo état qui crée un embranchement dynamique dans une transition. Il possède une entrée et au moins deux sorties. Contrairement à un point de jonction, les gardes situées après le point de décision sont évaluées au moment où il est atteint. Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix. Si, quand le point de décision est atteint, aucun segment en aval n'est franchissable, c'est que le modèle est mal formé.

Il est possible d'utiliser une garde particulière, notée [else], sur un des segments en aval d'un point de choix. Ce segment n'est franchissable que si les gardes des autres segments sont toutes fausses. L'utilisation d'une clause [else] est recommandée après un point de décision car elle garantit un modèle bien formé **[2].**

La figure I.9 désigne le formalisme de représentation de point de décision.

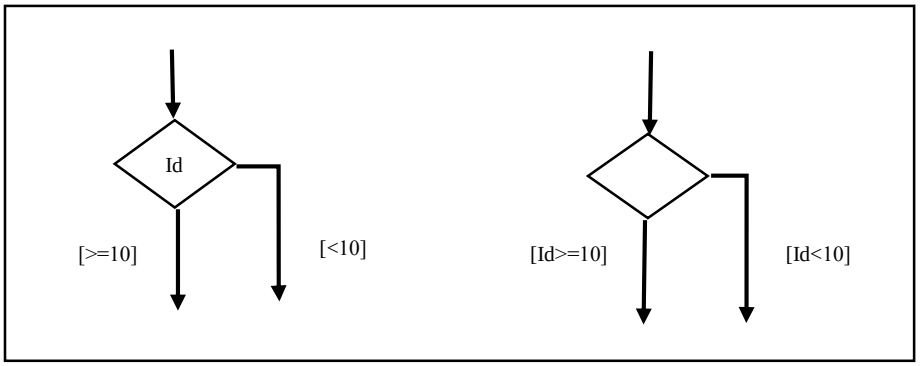


Figure I. 9: Représentation graphique de point de décision **[2]**.

4.2.8 Transitions complexes (débranchement et jointure)

Transitions complexes est une transition ayant plusieurs états source et/ou cible.

a. **Débranchement (fork) :** création de deux états concurrents.
b. **jointure (join) :** supprime la concurrence.

La figure I.10 désigne la représentation graphique de débranchement (fork) et jointure.

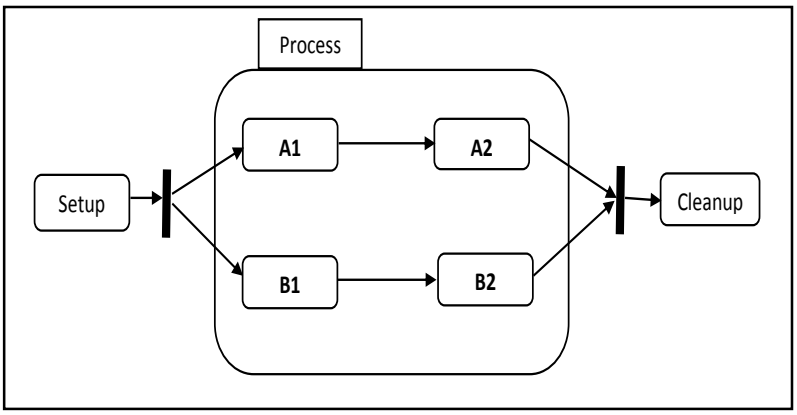


Figure I. 10: Représentation graphique de débranchement (fork) et jointure [2].

5. Conclusion

Dans ce chapitre, nous avons présenté le langage de modélisation unifié UML et ses diagrammes. Ensuite, nous avons donné une description de diagramme de classe et de diagramme d'états-transitions, et les intérêts de chaque diagramme.

Notre objectif dans cette mémoire est de transformer le diagramme d'états-transitions UML vers FoCaLiZe, donc nous allons présenter dans le chapitre suivant le langage FoCaLiZe et les caractéristiques les plus importants.

CHAPITRE II

L’environnement FoCaLiZe

1. Introduction.
2. Spécification (espèce).
3. Abstraction (collection).
4. Preuve.
5. Compilation.
6. Conclusion.

1. Introduction

FoCaLiZe est un environnement du développement se basé sur une approche formelle, qui intègre un prouver automatique (ZENON) et un outil d'aide à la preuve (Coq). Il a été initié à la fin des années 90 par Thérèse Hardin et Renaud Rioboo au sein de laboratoires LIP6, CEDRIC et INRIA **[13, 14]**.

FoCaLiZe a été conçu pour construire des applications étape par étape, partant de spécifications abstraites, appelées espèces, vers des implantations concrètes, appelées collections. Ces différentes structures sont combinées en utilisant l'héritage et la paramétrisation, inspirés de la programmation orientée objet **[9]**.

Dans le présent chapitre, nous allons détailler les différents concepts de l'environnement FoCaLiZe, nous commençons tout d'abord par l'aspect spécification, puis nous décrivons l'aspect implémentation et l'aspect preuve.

2. Spécification (espèce)

La première notion principale du langage FoCaLiZe est la structure d'espèce, qui correspond au niveau le plus élevé d'abstraction dans une spécification. Une espèce peut être globalement vue comme une liste d'attributs de trois sortes **[9]**:

- Le type support, appelé représentation, qui est le type des entités qui sont manipulées par les fonctions de l'espèce; la représentation peut être soit abstraite soit concrète.
- Les fonctions, qui dénotent les opérations permises sur les entités de la représentation ; les fonctions peuvent être soit des définitions (quand un corps est fourni), soit des déclarations (quand seul le type est donné).
- Les propriétés, qui doivent être vérifiées par toute implantation de l'espèce ; les propriétés peuvent être soit simplement des propriétés (quand seul l'énoncé est donné), soit des théorèmes (quand la preuve est également fournie). Le tableau suivant décrit la syntaxe générale d'une espèce. Le tableau II.1 la syntaxe générale d'une espèce.

species *<name> =*	
representation *[= <type>] ;*	*(* représentation*)*
signature *<name> : <type>;*	*(*déclaration*)*
let *<name> = <body>;*	*(*definition*)*
property *<name> : <prop>;*	*(*propriétés*)*
theorem *<name> : <prop> ;*	*(*théorème*)*
proof *= <proof>;*	
end *; ;*	

Tableau II. 1: La syntaxe générale d'une espèce [9].

Où **<name>** est un nom, **<type>** une expression de type, **<body>** un corps de fonction, **<prop>** une proposition et **<proof>** une preuve (exprimée au moyen d'un langage de preuve déclaratif).

2.1 Représentation

La représentation d'une espèce définit la structure des données manipulées par les fonctions et les propriétés de l'espèce. C'est un type, défini par une expression de type en FoCaLiZe **[11]**: variables de types, types primitifs, types inductifs, enregistrements, etc.

Exemple: Nous définissons l'espèce Point qui modélise les points du plan. La représentation de cette espèce est exprimée par un couple de réels. Le premier représente l'abscisse et le deuxième désigne l'ordonnée d'un point :

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

La représentation d'une espèce peut demeurer abstraite (la représentation n'est pas donnée) le long de quelques étapes de la spécification, mais elle doit être explicitée par une expression de type avant l'implémentation finale de l'espèce. Une fois la représentation d'une espèce donnée par une expression de type, elle ne peut plus être redéfinie. Ceci exprime le fait que les fonctions et les propriétés d'une espèce opèrent sur un type de données unique **[11]**.

Entité d'espèce : est un élément de type représentation de l'espèce, créée par une fonction (constructeur) de l'espèce. Par exemple, la paire (1.5, 0.5) est une entité de l'espèce Point.

2.2 Fonctions

Les fonctions d'une espèce permettent de manipuler les entités d'une espèce et de décrire leurs comportements. Dans l'environnement FoCaLiZe existe deux catégories de fonctions **[14]**:

1. Une fonction déclarée sert à spécifier une fonction sans rentrer dans les détails de son implémentation, et elle est spécifiée par le mot clé **signature**.
2. Une fonction définie est une fonction complètement définie via son corps calculatoire, et elle est spécifiée par le mot clé **let**.

2.2.1 Fonction déclarée (signature)

Une signature est une fonction énoncée uniquement par son type fonctionnel. Elle permet de décrire la vue abstraite d'une fonction, qui sera concrétisée par les héritières de l'espèce **[12]**.

Nous complétons notre espèce **Point** par la signature de la fonction constructrice **deplacer_point** (pour déplacer un point d'une position vers une autre) et la fonction binaire **egal** (pour décider de l'égalité de deux points) comme suit :

```
species Point = representation = float * float;
signature deplacer_point : Self -> float-> float -> Self;
signature egal : Self -> Self -> Bool; ... end;;
```

Le mot clé **Self** (utilisé dans les types des fonctions **deplacer_point** et **egal**) dénote une entité de l'espèce en cours de spécification, même si la représentation d'une espèce n'est pas définie (abstraite).

Le type **Self** -> **float**-> **float** -> **Self** de la fonction **deplacer_point** spécifie une fonction paramétrée par une entité de l'espèce Point (le premier **Self**) et par deux réels, et retournant une nouvelle entité de l'espèce (le dernier **Self**).

2.2.2 Fonction définie (let)

Une fonction définie est une fonction énoncée parallèlement avec son corps calculatoire. Le corps calculatoire d'une fonction est exprimé par des instructions fonctionnelles comme **[12, 14, 11]**: la liaison let (**let**...**in**), le filtrage (**match**...**with**), la conditionnelle (**if**... **then**...**else**), les fonctions d'ordre supérieur, etc.

Une fonction définie est soit nouvellement énoncée (pour la première fois), ou bien la définition d'une signature énoncée au niveau des ascendants de l'espèce ou même la redéfinition d'une fonction déjà définie. FoCaLiZe permet aussi la définition des fonctions récursives en utilisant **let rec [14].**

Continuons avec notre espèce **Point**, les fonctions **get_x** (récupérer l'abscisse d'un point) et **get_y** (récupérer l'ordonnée d'un point) sont définies en utilisant les fonctions **fst** (retournant le premier composant d'une paire) et **snd** (retournant le deuxième composant d'une paire) de la bibliothèque basics de FoCaLiZe. La fonction **different** (décidant de l'inégalité de deux points) est définie à partir de la signature **egal** (donnée ci-dessus) :

```
species Point =
...
let get_x(p:Self) = fst(p);
let get_y(p:Self) = snd(p);
let different(a, b) = ~~(egal(a, b));
...
end;;
```

Le symbole ~~ désigne la négation logique.

2.3 Propriétés

Les propriétés sont des contraintes à satisfaire par les entités de l'espèce. FoCaLiZe permet deux niveaux de spécification de propriétés. Une propriété est soit uniquement déclarée (**property**), sans se préoccuper de sa preuve (niveau abstrait), soit définie (**theorem**) avec sa preuve (niveau concret). Nous détaillons les deux types de propriétés dans les paragraphes suivants **[14, 12, 9].**

2.3.1 Propriétés déclarées

Une propriété déclarée est une formule de la logique du premier ordre énoncée sans fournir sa preuve. Elle permet d'exprimer une exigence de sûreté sur les entités de l'espèce dès les premières étapes de la spécification, même avant la définition de la représentation et des fonctions de l'espèce. La preuve d'une telle propriété devra être donnée au niveau des héritiers de l'espèce **[9, 12, 11].**

Pour définir une structure respectant une relation d'équivalence sur l'opération **egal** de l'espèce Point, nous n'introduisons les trois propriétés suivantes:

```
species Point =
 property egal_reflexive : all p:Self, egal(p, p);
property egal_symetrique : all p1 p2: Self, egal(p1, p2) -> egal(p2, p1) ;
property egal_transitive : all p1 p2 p3: Self, egal(p1, p2) -> egal(p2, p3)-> egal(p1,p3);
... end;;
```

Le mot clé **all** dénote le quantificateur universel **(∀)** et le symbole **->** désigne le connecteur d'implication.

2.3.2 Théorèmes

Un théorème est une propriété énoncée conjointement avec sa preuve. Un théorème peut être nouvellement énoncé, ou produit la preuve d'une propriété énoncée au niveau des ascendants de l'espèce **[14]**.

FoCaLiZe dispose de trois modes de preuve de théorèmes **[14]**:

- Soit en utilisant le mot clé **assumed** pour admettre un théorème sans donner sa preuve (axiome).
- Soit en introduisant manuellement un script de la preuve en Coq.
- Soit en utilisant le langage de preuve de FoCaLiZe (FPL, FoCaLiZe Proof Language) pour écrire une preuve qui sera directement déchargée par **Zenon** (le prouveur automatique de théorèmes de FoCaLiZe).

Un théorème peut faire référence à n'importe quelle fonction ou propriété reconnue dans le contexte de l'espèce courante, comme il peut être utilisé pour spécifier d'autres propriétés de la même espèce ou de ses héritières.

En utilisant le langage FPL, la preuve de théorème **egal_est_non_differen**

((a = b)) => ⅂ (6 ≠ b)) est acceptée par **assumed**, comme suit :

```
species Point =
...
theorem egal_est_non_different : all p1 p2:Self, egal(p1, p2) -> ~(different(p1, p2))
proof = assumed;
    ...
end;;
```

2.4 Héritage

Un développement FoCaLiZe est organisé comme une hiérarchie pouvant avoir plusieurs racines. Les niveaux les plus hauts de cette hiérarchie sont construits durant les étapes de spécification, tandis que les niveaux inférieurs correspondent aux implémentations. Chaque noeud de la hiérarchie, c'est-à-dire chaque espèce, est une progression vers une implantation complète.

On peut utiliser le mécanisme d'héritage pour créer de nouvelles espèces en utilisant des espèces existantes. La nouvelle espèce créée par héritage acquiert toutes les méthodes (y compris la représentation) de ses super-espèces.

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 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 **[14, 11]**.

Exemple : les espèces ***PointColore*** héritent des espèces **Point**.

```
type couleur = | Rouge | Vert | Bleu ;
species PointColore = inherit Point;
representation = (float * float) * couleur;
let get_color(p:Self):couleur = snd(p);
let creerPointColore(x:float, y:float, c:couleur):Self = ((x, y), c);
let get_x(p) = fst(fst(p));
let get_y(p) = snd(fst(p));
let deplacer(p, dx, dy) = ((get_x(p) +f dx, get_y(p) +f dy ), get_color(p) );
let affiche_point (p:Self):String =
let affiche_couleur (c:couleur) = match c with
| Rouge -> "Rouge"
| Vert -> "Vert"
| Bleu -> "Bleu"
in ( " X = " ^ string_of_float(get_x(p)) ^
" Y = " ^ string_of_float(get_y(p)) ^
" COULEUR = " ^ affiche_couleur(get_color(p)));*)
end;;
```

Pour manipuler des points colorés (points graphiques), une nouvelle espèce **PointColore** est créée en héritant de l'espèce **Point**. L'espèce **PointColore** a besoin de manipuler la couleur d'un point en plus de ses cordonnées. Par conséquent, elle hérite de toutes les méthodes de l'espèce **Point** et apporte les enrichissements suivants : Elle définit la représentation par l'expression **(float * float) * couleur** (le type couleur modélise la couleur d'un point), fournit des corps calculatoires aux signatures **get_x**, **get_y** et **deplacer** et introduit la fonction **get_color** pour récupérer la couleur d'un point. Elle définit aussi la fonction **affiche_point**, pour permettre l'affichage de la couleur du point en plus de ses cordonnées.

2.5 Espèce compléte

Une espèce est dite complète si toutes ses déclarations ont reçu une définition et toutes ses propriétés des preuves**,** Cette notion implique que **[9, 12]**:

- La représentation a été associée à une définition de type.

- Chaque déclaration est associée à une définition.
- Une preuve est donnée pour chaque propriété.

3. Abstraction (collections)

3.2 Collection

L'autre notion majeure du langage FoCaLiZe est la structure de collection, qui correspond à l'implantation d'une spécification. Une collection implante une espèce de manière à ce que chaque attribut devienne concret : la représentation doit être concrète, les fonctions doivent être définies et les propriétés doivent être prouvées. Si l'espèce implantée est paramétrée, la collection doit aussi fournir des implantations pour ces paramétrées: soit une collection s'il s'agit d'un paramètre de collection, soit une entité s'il s'agit d'un paramètre d'entité. De plus, une collection est vue (par les autres espèces et collections) à travers son interface correspondante ; en particulier, la représentation est un type abstrait de données et seules les définitions de la collection peuvent manipuler les entités de ce type. Enfin, une collection est un objet terminal, qui ne peut pas être étendu ou raffiné par héritage. La syntaxe d'une collection est la suivante **[9]** :

collection <name> = **implement** <name> (<pars>) **end** ; ;

Nous implémentons les espèces complètes **PointAbstrait** comme suit:

```
species PointAbstrait =
signature get_x : Self -> float;
signature get_y : Self -> float;

let get_x(p:Self) = fst(p);
let get_y(p:Self) = snd(p);
let distance (p1:Self, p2: Self):float = #sqrt( ( get_x(p1) -f get_x(p2) ) *f ( get_x(p1) –f
get_x(p2) ) +f ( get_y(p1) -f get_y(p2) ) *f ( get_y(p1) -f get_y(p2) ) );

Proof of distance = assumed;

end;;

collection PointAbstrait Collection = implement PointAbstrait;

end;;
```

3.2 Paramétrage

Le paramétrage est un mécanisme qui permet de créer une nouvelle espèce en utilisant des espèces existantes. Le paramétrage permet à une espèce d'utiliser les méthodes et les entités des autres espèces. FoCaLiZe permet deux formes de paramétrage **[14, 11]**:

- Une espèce peut être paramétrée par collections (des espèces spécifiées).
- Une espèce peut être paramétrée par entités de collections.

3.2.1 Paramètres de collections

Un paramètre de collection est une espèce (fournisseur) utilisée par une autre espèce (la cliente) comme type abstrait, par l'intermédiaire de son interface.

Un paramètre de collection est introduit par un nom (**parameter_name**) et une interface désignée par le nom d'une espèce (**supplier_species_name**), en utilisant la syntaxe suivante:

species *client_species_name* (*parameter_name* is *supplier_species_name*, . . .) . . . **end**;;

Le nom du paramètre (**parameter_name**) est utilisé par l'espèce cliente comme une variable de type, et pour appeler les méthodes de l'espèce fournisseur. En géométrie, un cercle est défini par son centre (un point) et son diamètre. Une espèce Cercle (modélisant les cercles) peut être créée en utilisant l'espèce **PointAbstrait** comme paramètre de collection :

```
species Cercle (P is PointAbstrait) = representation = P * float ;

let get_centre(c:Self):P = fst(c);

 let get_rayon(c:Self):float = snd(c);

let creeCercle(centre:P, rayon:float):Self = (centre, rayon);

let appartient(p:P, c:Self): Bool = (P!distance(p, get_centre(c)) =
get_rayon(c));

theorem appartient_spec : all c:Self, all p:P, appartient(p, c) <->
(P!distance(p, get_centre(c)) = get_rayon(c))

proof = assumed ; end;;
```

Pour créer une collection à partir d'une espèce paramétrée par paramètres de collections, chaque paramètre devra être substitué par le nom d'une collection effective du modèle. C'est pour cette

raison qu'elles sont appelées paramètres de collections. Par exemple, nous pouvons créer les deux collections suivantes à partir de l'espèce complète Cercle :

collection CercleCollection1 = **implement** Cercle(PointConcretCollection); **end**;;

collection CercleCollection2 = **implement** Cercle(PointColoreCollection); **end**;;

Pour générer la collection CercleCollection1, nous avons substitué le paramètre de l'espèce Cercle par la collection **PointConcretCollection**, créée à partir de l'espèce **PointConcret**. Pour la deuxième collection, le paramètre de l'espèce Cercle est substitué par la collection **PointColoreCollection**, issue de l'espèce **PointColore**.

3.2.2 Paramètres d'entités

Un paramètre d'entité est une entité d'une espèce existante (espèce fournisseur), utilisée par l'espèce cliente pour servir en tant que valeur effective au sein du développement de ses propres méthodes.

Un paramètre d'entité est introduit par le nom d'une entité (**entity_name**) et le nom d'une collection (**collection_name**), en utilisant la syntaxe :

species *client_species_name* (*entity_name* in *collection_name*, . . .) . . .

La collection **collection_name** peut servir comme type dans l'espèce, comme elle peut être utilisée pour invoquer les fonctions de l'espèce complète sous-jacente.

Dans cet exemple de la géométrie, l'origine d'un repère orthogonal est un point particulier (valeur) qui peut être une entité de la collection **PointConcretCollection** donnée ci-dessus. Ainsi, une espèce **RepereOrthogonal** qui modélise les repères orthogonaux utilise un paramètre d'entité de la collection **PointConcretCollection** comme suit :

```
species RepereOrthogonal (origine in PointConcretCollection) =
...end;;
 let o=PointConcretCollection!creerPointConcret(0.0, 0.0);;
collection RepereOrthogonalZero = implement
RepereOrthogonal(o);end;;
```

FoCaLiZe ne permet pas la spécification d'un paramètre d'entité de type primitif (int, string, bool . . .). Cela signifie que les types primitifs doivent être intégrés dans des collections pour être utilisés comme paramètres d'entités.

4. Preuves en FoCaLiZe

Prouver une propriété consiste à s'assurer que son énoncé est satisfait dans le contexte de l'espèce courante. FoCaLiZe distingue entre les propriétés déclarées et les théorèmes. La preuve d'un théorème doit accompagner son énoncé. Par contre, la preuve d'une propriété déclarée (property) est remise plus tard **[14].** La syntaxe générale pour introduire la preuve d'un théorème ou d'une propriété est présentée comme suit :

```
theorem theorem_name : theorem_specification
    proof = proof ;                    (* cas d'un théorème *)
proof of property_name = proof ;       (* cas d'une propriété déjà déclarée*)
```

5. Compilation

La compilation d'un fichier source FoCaLiZe génère 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 les preuves et la cohérence du modèle. La compilation est réalisée, en quatre étapes **[14]**:

1. **Compilation du fichier source FoCaLiZe (source.fcl) :** Le compilateur FoCaLiZe (focalizec) lit le fichier source (source.fcl) et génère un code OCaml (source.ml) et un code Coq intermédiaire (source.zv) qui contient une traduction du code FoCaLiZe (y compris l'aspect calculatoire) vers Coq. A la place des preuves de propriétés, nous trouvons des emplacements vides (trous) dans source.zv qui sera complétés par les preuves que Zenon va produire.
2. **Compilation du code OCaml :** A cette étape, FoCaLiZe invoque le compilateur ocamlc pour produire un exécutable à partir du code OCaml (source.ml), généré dans l'étape précédente.
3. **Compilation du code Coq intermédiaire (source.zv) :** FoCaLiZe invoque la commande zvtov pour produire un code Coq complet (source.v) à partir du code intermédiaire (source.zv) généré dans la première étape. Tous les vides laissés dans source.zv sont remplis par des preuves Coq effectives réalisées par Zenon.
4. **Compilation du code Coq :** Enfin, FoCaLiZe invoque la commande coqc pour compiler le code Coq (source.v), généré dans l'étape précédente.

6. Conclusion

FoCaLiZe est le résultat d'un travail collectif de plusieurs chercheurs, lorsqu'il se caractérise par plusieurs spécifications telles que:

- ✓ Est un environnement de développement de logiciels sûrs.
- ✓ Est un atelier de développement et de la programmation orienté objet.
- ✓ Couvre toutes les phases du cycle de vie du logiciel.
- ✓ Est une langue basée sur des résultats théoriques fermes avec une sémantique claire et fournit une mise en œuvre efficace.
- ✓ Est une méthode formelles qui permet d'exprimer des propriétés et prouve, que ces propriétés sont satisfaits vis-à-vis les spécifications.

CHAPITRE III

Travaux connexes

1. Introduction.
2. Transformation en Alloy.
3. Transformation en B.
4. Transformation en Maude.
5. Transformation en FoCaLiZe.
6. Conclusion.

1. Introduction

Comme nous avons mentionné précédemment, UML est l'un des langages de modélisation les plus répandus pour la conception des applications informatiques, UML aussi étant un langage semi-formel, mais il y a un point faible est l'absence de bases formelles permettant l'application des techniques de vérifications formelles, pour cette raison, il y a plusieurs travaux qui ont transféré les modèles UML aux langages formelles tels que Maud, B, Alloy et FoCaLiZe.

Dans ce chapitre, nous allons se concentrer sur les travaux liés au titre dans notre mémoire, qui transforme UML aux langages formels.

2. Transformation en Alloy

Alloy est un langage formel avec une syntaxe basée sur un ensemble restreint, pour décrire des propriétés structurelles et est un langage de modélisation textuel basé sur une logique relationnelle de premier ordre. Un modèle d'Alloy consiste en un certain nombre de déclarations de signature, de champs, de faits et de prédicats **[15]**.

Dans la transformation de l'aspect structurel d'UML vers Alloy il y a de nombreux travaux, nous trouvons une étude exhaustive sur la transformation de diagrammes d'états-transitions d'UML vers Alloy **[16]**. Cette étude a donné un formalisme des diagrammes d'états-transitions par Alloy, pour simuler et vérifier la consistance entre les aspects statiques et dynamiques.

3. Transformation en B

Selon **[17,18]** la méthode B est un langage de programmation formelle basée sur la théorie des ensembles. Elle a l'avantage de couvrir toutes les phases du cycle de vie, depuis l'analyse des besoins jusqu'à l'implémentation finale. Elle est de plus très bien outillée. Le langage B est basé sur la notion de machine abstraite qui est fondée sur les notions d'état et de propriétés d'invariance. Les outils associés à la méthode permettent d'une part de vérifier la correction des machines spécifiées et d'autre part de prouver des propriétés sur les spécifications obtenues.

Clin Snook et Michael Butler dans son document **[18],** proposent des règles de la transformation des diagrammes UML (diagramme de classe et d'état) vers la méthode B.

4. Transformation en Maude

Maude est un langage formel, peut être comprise la conception de Maude comme un effort pour maximiser simultanément trois dimensions **[19]**:

- La simplicité: Un programme doit être le plus simple possible et aussi facile à comprendre. Il doit également posséder une sémantique très claire.
- La performance: Maude est un langage concurrentiel en termes d'exécution avec d'autres langages de la programmation impérative, il peut exécuter des millions de réécriture par seconde, cela permet la vérification de différents chemins d'exécution possibles dans une spécification.
- L'expressivité: Il est possible d'exprimer naturellement une vaste gamme d'applications. Il est aussi possible de le faire autant pour un programme déterministe que pour un programme hautement concurrentiel. Maximiser l'expressivité du langage est sans doute l'une des avantages les plus remarquables du langage Maude. Dans ce contexte, Maude devrait alors être vu comme un métalangage avec lequel il est très aisé de développer un langage spécifique.

BOUDIA MALIKA dans sa mémoire **[20]** propose une approche de la transformation du diagramme d'états-transitions vers le langage Maude. Cette approche englobe en réalité deux étapes de transformation:

1- Application des algorithmes de Saldhana **[21]**.
2- Utilisation de l'outil ATOM3 **[22]**.

5. Transformation en FoCaLiZe

Dans **[23, 24, 25],** ils ont étudié les règles de transformation de modèles UML exprimés par des diagrammes de classes, des diagrammes d'activités en spécifications FoCaLiZe, puis ils ont proposé une mise en œuvre de ces règles de transformation. Le but de leur transformation est de tirer une partie des outils de vérification de FoCaLiZe pour vérifier les propriétés d'un modèle UML.

6. Conclusion

Comme un résumé, les méthodes puissantes comme B, Maude et Alloy ont prouvé leurs capacités pour prendre en charge plusieurs fonctionnalités UML, mais elles ignorent plusieurs fonctionnalités essentielles.

Dans cette mémoire, on va proposer une transformation des diagrammes d’états-transitions vers FoCaLiZe, et on va choisir ce dernier parce qu'il comprend la plupart des fonctionnalités architecturales et conceptuelles et prend en charge la majorité des exigences imposées par les normes sur l'évaluation de cycle de vie de développement de logiciels.

CHAPITRE IV

Transformation des diagrammes d’états-transitions vers FoCaLiZe

- **Introduction.**
- **Transformation d'une classe.**
- **Transformation de digramme d’états-transitions.**
- **Conclusion.**

1. Introduction

La transformation du diagramme d'états-transitions UML vers FoCaLiZe consiste à formaliser les évènements, les transitions entre les états et les différents pseudo-états. Toutes les tentatives de formalisation des modèles UML sont basées sur la transformation de diagramme de classe, ainsi nous avons utilisé le travail présenté dans **[23]** et les recherches existant dans le domaine des transformations de diagramme de classe UML vers FoCaLiZe **[12]**.

Donc, dans ce chapitre nous allons expliquer:

- Les règles simples de transformations d'une classe UML vers FoCaLiZe.

- Les règles de transformation de diagrammes d'états-transitions UML vers FoCaLiZe.

2. Transformation d'une classe

Une classe UML est transformée en une espèce FoCaLiZe comme suit :

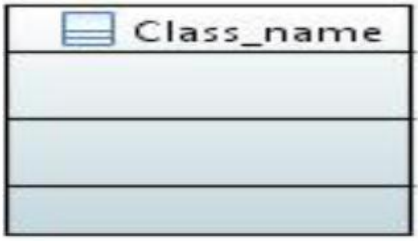
Classe UML	Code FoCaLiZe
	**Species Class_name =** **…** **end;;**

Tableau IV. 1: Transformation de classe UML.

Une classe peut contenir une liste d'attributs et des opérations. Ensuite, nous allons décrire la transformation de ces derniers.

2.1 Transformation d'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 qui modélise l'attribut **attr** dans les espèces correspondantes est:

Signature get_attr: Self -> [attr_type];

Où **[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.

Le tableau suivant montre la transformation d'une classe UML avec des attributs:

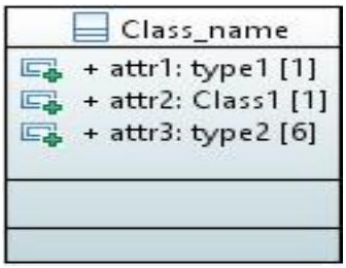
Les attributes de classe	Code FoCaLiZe
	**species Class_name(C is Class1)=** **signature get_attr1: Self -> attr_type1 ;** **signature get_attr2: Self -> C ;** **signature get_attr2: Self ->list(type2) ;** **end;;**

Tableau 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 le tableau IV.2): nous utilisons le type liste dans FoCaLiZe:

Signature get_attr: Self ->list (Foc_attr_type);

2.2 Transformation des opérations

Les opérations de classe seront transformées en fonctions (signature) des 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, 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 -> type1->… ->typen->return_type;

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

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

Signature op_name: self -> type1->… ->typen->self;

- Si l'opération est sans paramètres:

Op_name (), elle se transforme en une signature des espèces correspondantes:

Signature op_name: self ->self;

Le tableau suivant montre la transformation d'une classe avec des opérations :

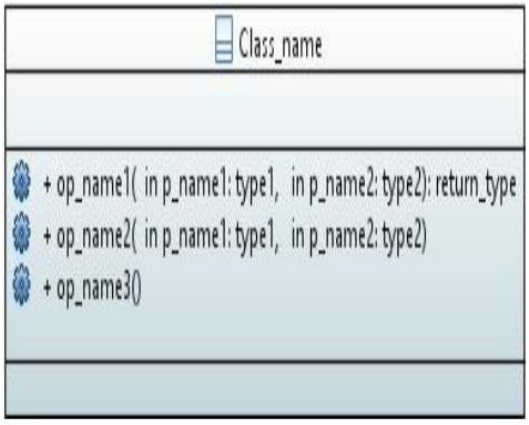
Les operations de classe	Code FoCaLiZe
	**Species Class_name =** **signature op_name1:Self -> type1 -> type2 -> return_type;** **signature op2_name: Self ->type1 ->type2 -> Self;** **signature op3_name: Self ->Self;** **end;;**

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

Quand une classe UML spécifiée par un constructeur, dans ce cas, 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:

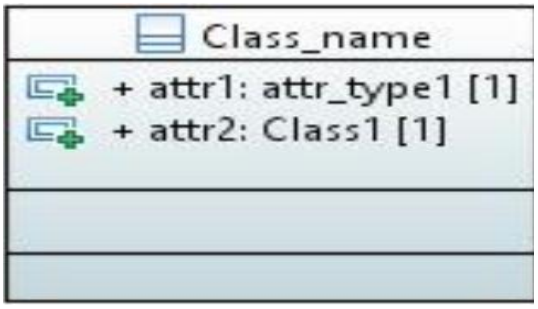
Classe UML	Code FoCaLiZe
	**Species Class_name(C is Class1) =** **signature get_attr1: Self -> attr_type1 ;** **signature get_attr2: Self -> C ;** **new Class_name: attr_type1 -> C -> self ;** **end;;**

Tableau IV. 4: Transformation du constructeur de la classe.

2.3 Transformation des signaux

La transformation des signaux est nécessaire pour la transformation du diagramme d'états-transition, parce que le signal peut être l'événement qui déclenche une transition pour passer d'un état à un autre.

Chaque signal sera transformé en fonctions (signature) des espèces correspondantes et on ajoute à chaque fonction (transformation d'un signal) une autre fonction booléenne pour situer l'événement.

Le signal de classe	Code FoCaLiZe
	**species Class1(C is Class1) =** **signature signal1 : Self -> Self ;** **signature is_Signal1 : Self -> bool;** **signature signal2 : Self -> Self ;** **signature is_Signal2 : Self -> bool;** **end;;**

Tableau IV. 5: Transformation du signal de la classe.

3. Transformation de digramme d'états-transitions

Nous proposons maintenant une approche pour la transformation de diagramme d'états-transitions en FoCaLiZe, cette approche s'est basée sur les transformations de diagramme de classe.

Exemple: Pour expliquer notre transformation, nous définissons le diagramme d'états-transitions de la classe **ATM**.

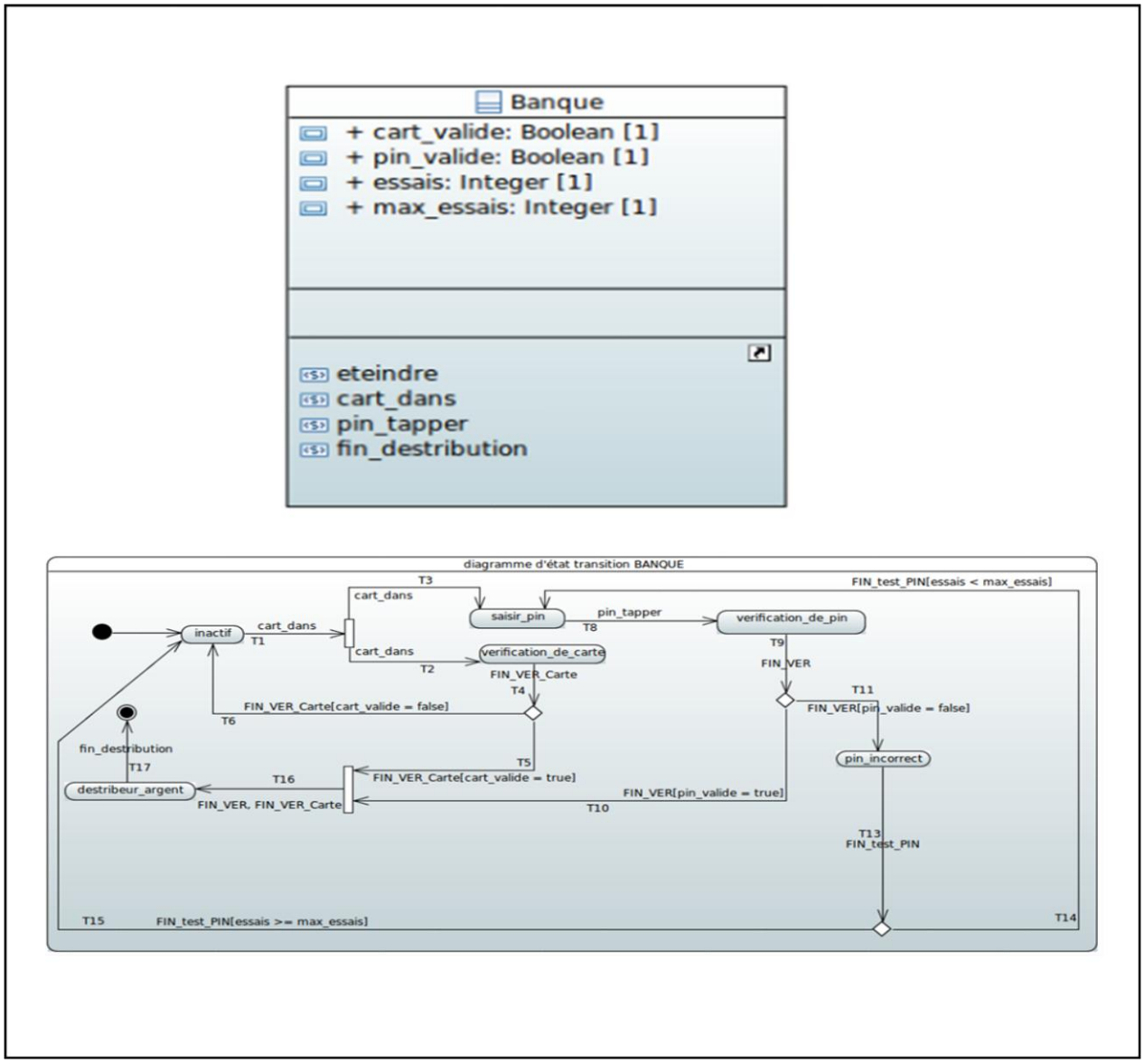


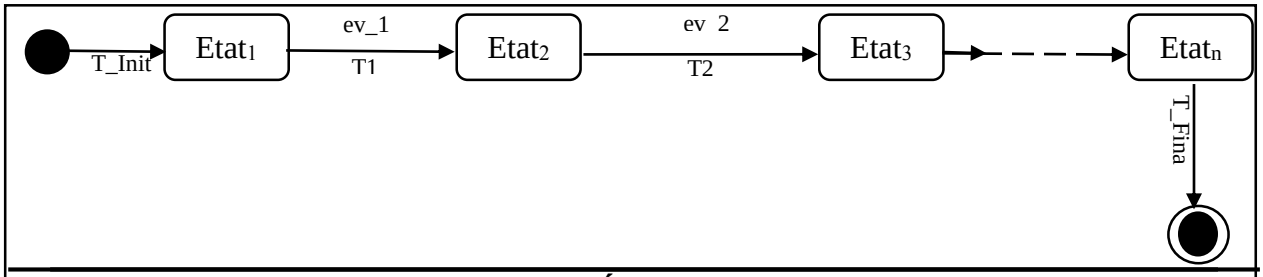
Figure IV. 1: Diagramme d'états-transitions de la classe 'BANQUE'.

Dans notre approche, nous avons créé une fonction récursive définie (**let rec**) à chaque diagramme d'états-transitions de classe UML.

L'entête de la fonction résultante de la transformation diagramme d'états-transitions est spécifié comme suit : **let rec diagSM (t :trans, e :self):(trans*self) ;**

Tel que:

- **t** : est un paramètre de type **trans**.
- **e** : est une entité de l'espèce.
- La fonction retourne un couple de type (**trans*self**).




```
type trans = | T_0 | T_Init | T1 | T2 | ... |  T_Final ;;
signature ev_1 : Self -> Self ;
signature is_ev1 : Self -> bool;
signature ev_2 : Self -> Self ;
signature is_ev2 : Self -> bool;
…
Species Class_name =
(*déclaration des attributs existants*)
(*déclaration des fonctions existants*)
let rec diagSM (t:trans, e:Self):(trans * Self) = match t with
| T_Init -> if (is_ev_1(e)) then diagSM(T1, ev_1(e))
else diagSM(T1, e)
| T1 -> if (is_ev_1(e)) then diagSM(T2, ev_1(e))
else diagSM(T1, e)
…
| T_Final -> diagSM(T_0, e)
| T_0 -> focalize_error("Fin de traitement") ;
end;;
```

Énumération regroupant toutes les transitions d'un diagramme d'états-transitions

Transformation des événements

Transformation du diagramme d'états-transitions en fonction récursive (diagSM)

Tableau IV. 6: Transformation générale d'un diagramme d'états-transitions.

Nous allons maintenant détailler la transformation de chaque composant de diagramme d'états-transitions comme ce suit.

3.1 Transformation des transitions

Les transitions représentent le passage instantané d'un état vers un autre. Ils sont modélisées en FoCaLiZe par nouveau type d'énumération "**trans**" en dehors l'espèce. Chaque transition correspond à une valeur d'énumération générée, comme suivant :

type trans = | T_0 | T_Init | T1 | T2 | ... | T_Final ;;

Tel que :

- **T_Init** modélise la transition sortante de l'état initiale.
- **T_Final** modélise la transition entrante dans l'état final.
- **T_1, T_2, …, T_n** modélisent les transitions intermédiaires.
- **T_0** est une transition supplémentaire introduite pour mettre fin à la fonction récursive **« diagSM »**.

3.2 Transformation des points de décision

L'expression suivante est la formalisation de point de décision :

```
|Ta -> if(is_even(e) && condition) then diagSM(Tb, even(e))
        else if(is_even(e) && non_ condition) then diagSM(Tc, even(e))
            else diagSM(Ta, e)
```

Tel que :

- **Ta:** transition courant.
- **Tb et Tc:** la prochaine transition selon la condition de garde de la décision (**Tb ou Tc**).
- **even:** événement qui déclencher la transition **Ta**.
- **is_even:** la fonction qui indique le déclenchement d'évènement.
- **Condition:** dans notre démarche nous utilisant le langage OCL (Object Constraint Langage) pour exprimer les conditions de gardes.

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

La règle de transformation est présentée dans le tableau suivant:

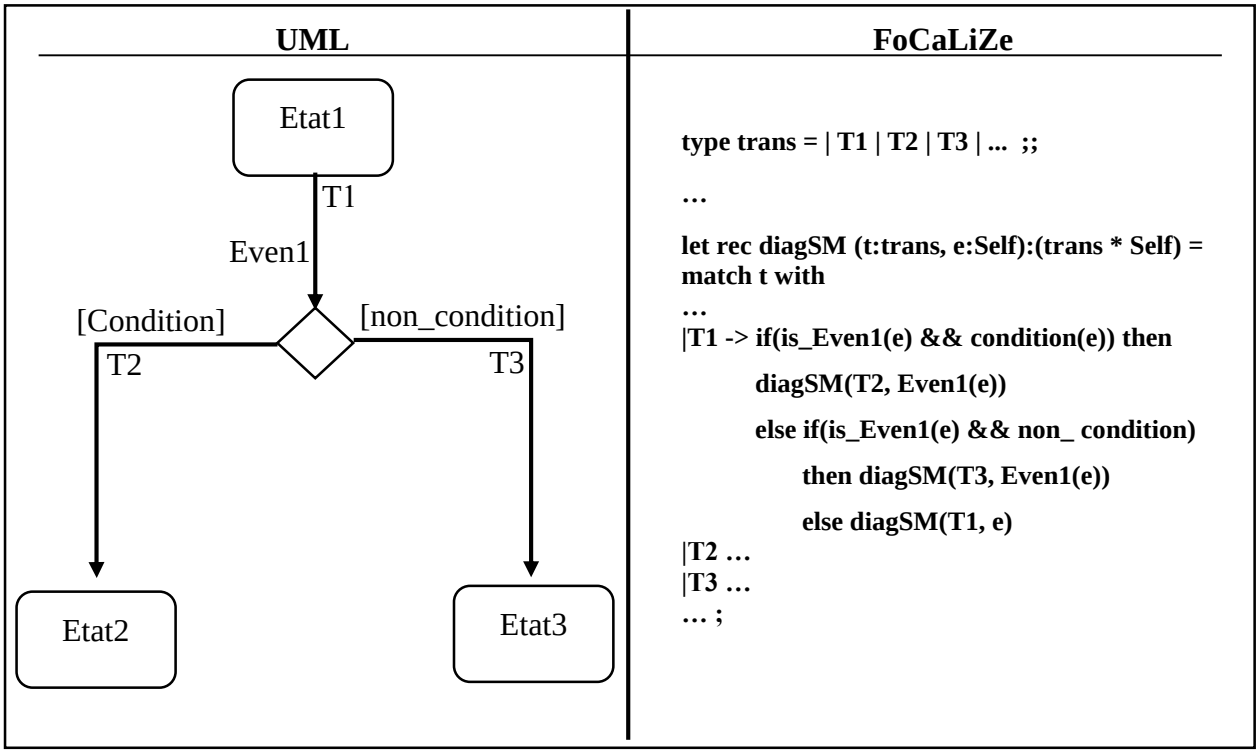


Tableau IV. 7: transformation de point de décision.

Un exemple de transformation d'un point de décision de notre diagramme d'états-transitions (voir Figure IV.1)**,** montré dans le tableau suivant :

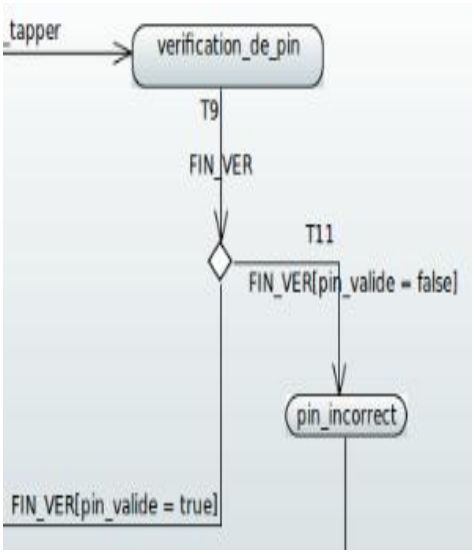
Point de décision	Code FoCaLiZe
	**type trans = | T_0 | T_Init |…| T9 | T10 | T11 ... | T_Final ;;** **signature FIN_VER : Self -> Self ;** **signature is_FIN_VER : Self -> bool;** **…** **Species Class_name=** **…** **| T9 ->** **if(is_FIN_VER(e) && [pin_valide(e) = true]) then** **diagSM(T10, FIN_VER(e))** **else if(is_FIN_VER(e) && [pin_valide(e) = false]) then** **diagSM(T11, FIN_VER(e))** **else diagSM(T9, e)** **|T10…** **|T11… ;**

Tableau IV. 8: Exemple de transformation d'un point de décision.

3.3 Transformation du débranchement et jointure (Fork / Join)

On ne peut pas gérer le débranchement et la jointure individuellement. Ils sont transformés en FoCaLiZe par l'utilisation de l'expression suivante:

let x1 = snd(diagSM(Ti, even(e))) and x2 = snd(diagSM(Tj, even(e)))… xn = snd(diagSM(Tk,even(e))) in diagSM(Tf,join_fun(x1,x2…xn))

Tel que:

- **Ti,Tj**,…et **Tk** est une transition sortant de pseudo état débranchement.
- **join_fun** est une fonction que nous définissons au niveau de l'espèce.
- **Tf** est la transition sortant de pseudo état jointure.

La règle de transformation du débranchement et jointure est donnée dans le tableau suivant:

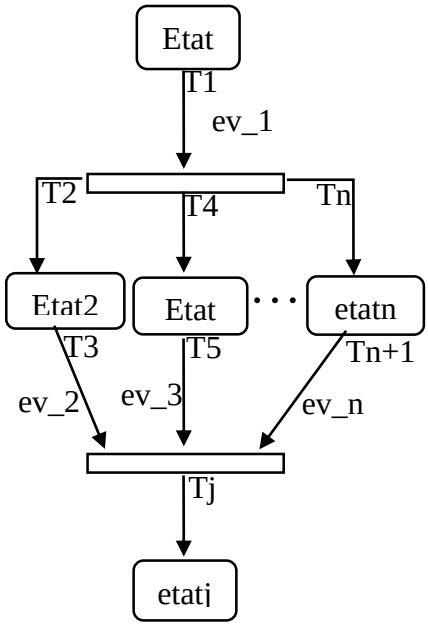
UML	FoCaLiZe
	**type trans = | T1 | T2 | T3 | T4 | T5 ... | Tn | Tn+1 | Tj … ;;** **Species Class_name=** **signature ev_1 : Self -> Self ;** **signature is_ev_1 : Self -> bool;** **signature ev_2 : Self -> Self ;** **signature is_ev_2 : Self -> bool;** **…** **signature ev_n : Self -> Self ;** **signature is_ev_n : Self -> bool;** **…** **signature join_fun : Self -> Self -> Self;** **let rec diagSM (t:trans, e:Self):(trans * Self) = match t with** **…** **| T1 ->** **let x1 = snd(diagSM(T2, ev_1(e))) and x2 =** **snd(diagSM(T4,ev_1(e))) and … xn =** **snd(diagSM(Tn, ev_1(e))) in diagSM(Tj,join_fun(x1,x2 … xn))** **| T2 -> …** **| T3 -> …** **| T4 -> …** **| T5 -> …** **…** **| Tn -> …** **…** **| Tj -> if (is_ev_2(e) && is_ev_3(e) && is_ev_n(e)) then …**

Tableau IV. 9: Transformation du débranchement et jointure.

On remarque dans la transition **T1** qu'il y a lancement de toutes les transitions sortantes de pseudo état du débranchement (Fork), et la condition de lancement de la transition **Tj** est l'arrivée de tous les événements des transitions entrants de pseudo état de jointure.

La transformation du débranchement et jointure de notre diagramme d'états-transitions de la figure IV.1 est indiqué dans le tableau suivant:

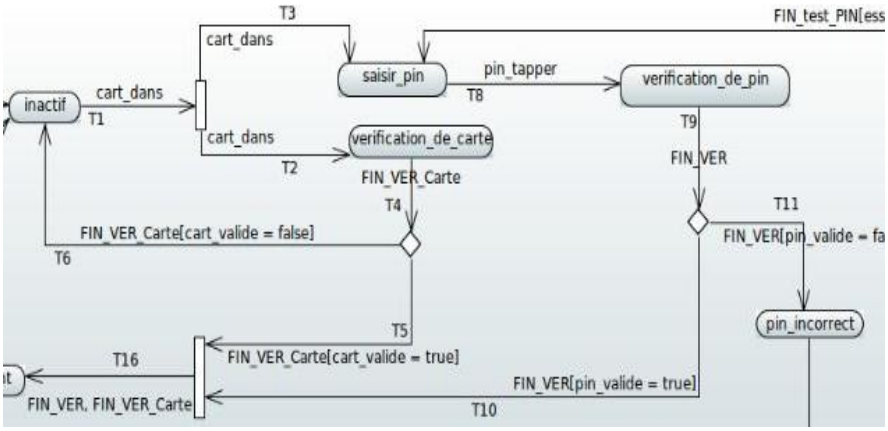
<table>
<tr><td>Débranchements et jointure</td><td> 
</td></tr>
<tr><td>Code FoCaLiZe</td><td>type trans = | T_0 | T_Init | T1 | T2 | T3 ... - | T_Final ;;
signature cart_dans : Self -> Self ;
signature is_ cart_dans : Self -> bool;
signature FIN_VER_cart : Self -> Self ;
signature is_ FIN_VER_cart : Self -> bool;
signature FIN_VER : Self -> Self ;
signature is_ FIN_VER : Self -> bool;
…
Species Class_name=
let rec diagSM (t:trans, e:Self):(trans * Self) = match t with
| T1 -> let x1 = snd(diagSM(T3, cart_dans(e))) and
x2 = snd(diagSM(T2, cart_dans(e))) in diagSM(T16,join_fun(x1,x2))
|T2 -> …
|T3 -> …
|T16 -> if(is_FIN_VER(e) && is_FIN_VER_Carte(e)) then diagSM(T_Final, e)
else diagSM(T16, e)
… ;</td></tr>
</table>

Tableau IV. 10: Exemple de transformation de l'état de débranchement et jointure.

3.4 Transformation des événements

Dans la transformation de classe, exactement dans la transformation des signaux, nous avons mentionné que les signaux peuvent être des événements qui déclenchent la transition entre les

états de diagramme d'états-transition, et pour ça nous formalisons les événements par l'expression suivante:

signature cart_dans : Self -> Self ;

signature is_ cart_dans : Self -> bool;

Aussi les opérations peuvent être des événements dans le diagramme d'états-transition, pour cela, nous avons ajouté une fonction booléenne à chaque opération transformée, pour connaitre quand l'événement se produit.

Par exemple: Si **Op_even()** est une opération dans quelque classe et est un événement de quelque transition de diagramme d'états-transitions alors sa transformation comme suit:

signature Op_even(): Self ->Self;

signature is_Op_even(): Self ->Self;

3.5 Transformation de l'état composite

Dans le chapitre I, nous avons dit que le diagramme d'états-transitions peut contenir des états composites soit orthogonaux ou non orthogonaux. Donc, avant de transformer ce type d'états vers FoCaLiZe, nous appliquons deux algorithmes proposés par **[21]**, pour transformer des diagrammes d'états transitions contiennent états composites (séquentiel/concurrent) vers diagrammes états transitions à plat (simple).

- L'algorithme A convertit d'états composite séquentiel vers un diagramme d'états-transitions plat, tel que l'entrée de l'algorithme est un diagramme d'états-transitions qui contient état composite séquentiel et la sortie est un diagramme d'états-transitions plat.

La figure IV.2 extraite de **[21]**, illustre un exemple de l'application de l'algorithme A.

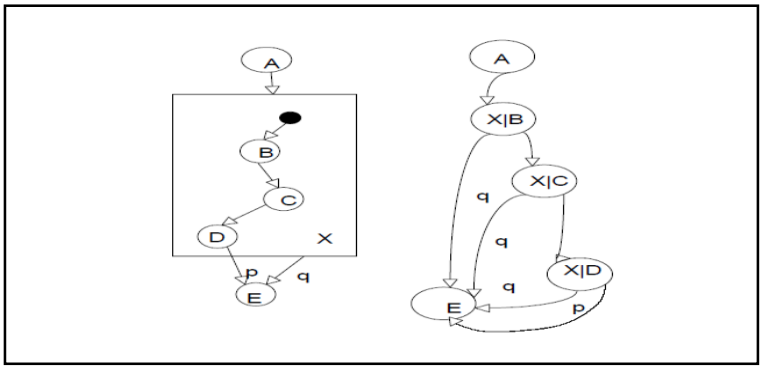


Figure IV. 2: Conversion d'états composite séquentiel vers un diagramme d'états-transitions plat **[21]**.

Algorithm A: Convert a sequential composite state into a flat state machine.

Input: A UML statechart diagram S containing a sequential composite state scs.

Output: Flat state machine O.

BEGIN

1. Let O be a flat state machine with zero states and zero transitions.

2. For all states s ∈ S and s ≠ scs, add state s and all its transition arcs into O.

3. If s ∈ S and s = scs,

- **Add each state sx ∈ scs and all its transition arcs into O.**
- **For each transition from a state s ≠ scs to a state sx ∈ scs, add a transition from s ∈ O to sx ∈ O.**
- **For each transition from a state sx ∈ scs to a state s ≠ scs, add a transition from sx ∈ O to s ∈ O.**
- **For an entry transition from a state s ∈ S and s ≠ scs to state scs, add a transition from s ∈ O to sx ∈ O such that sx corresponds to an initial state of state scs.**
- **For an exit transition from state scs to a state s ∈ S and s ≠ scs, if the transition is triggered by an event, add a transition from each state sx ∈ O to s ∈ O such that sx ∈ scs.**
- **For an exit transition from state scs to a state s ∈ S and s ≠ scs, if the transition is triggerless, add a transition from state sx ∈ O to s ∈ O such that sx corresponds to an ending state of state scs.**

END

Tableau IV. 11: L'algorithme A : convertit d'états composite séquentiel vers un diagramme d'états-transitions plat **[21]**.

- L'algorithme B convertit un état composite concurrent vers un diagramme d'états-transitions plat, tel que l'entrée de l'algorithme est un diagramme d'états-transitions qui contient état composite concurrent et la sortie est un diagramme d'états-transitions plat .la figureIV.3 extraite de **[21]**, illustre un exemple de l'application de l'algorithme B.

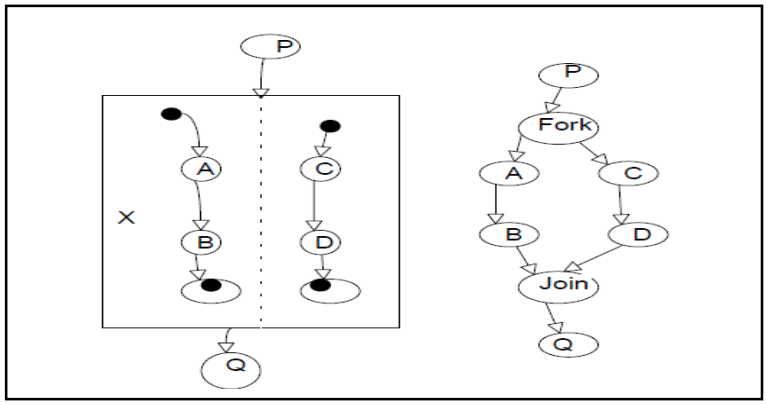


Figure IV. 3: Conversion d'états composite concurrent vers un diagramme d'états-transitions plat **[21]**.

Algorithm B: Convert a concurrent composite state into a flat state machine.
Input: A UML statechart diagram *S* containing a concurrent composite state *ccs*.
Output: Flat state machine *O*.
BEGIN
1. Let *O* be a flat state machine with zero states and zero transitions.
***2.* For all states $s \in S$ and $s \neq ccs$, add state *s* and all its transition arcs into *O*.**
***3.* If $s \in S$ and $s = ccs$,**
- **Add all the states along each distinct path in *s* into *O*.**
- **Add a state *fork* into *O* and add a transition from *fork* to each state in *O* that corresponds to an initial state on a distinct path in *s*.**
- **Add a state *join* into *O* and add a transition to *join* from each state in *O* that corresponds to an ending state on a distinct path in *s*.**

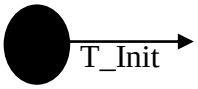
4. For all states $s \in S$ such that $s \neq ccs$ and there is a transition from *s* to *ccs*, add a transition from $s \in O$ to state $fork \in O$.
5. For all states $s \in S$ such that $s \neq ccs$ and there is a transition from *ccs* to *s*, if the transition is triggerless add a transition from state $join \in O$ to $s \in O$.
END

Tableau IV. 12: L'algorithme B: convertit un état composite concurrent vers un diagramme d'états-transitions plat **[21]**.

Le résultat des algorithmes est un diagramme d'états-transitions simple (pas des états composites), alors il est transformable par notre approche.

3.6 Transformation de l'état initial et final

Nous avons mentionné précédemment que **T_Init** est une formalisation de la transition qui sort de l'état initial et **T_Final** est une formalisation de la transition qui entre de l'état final en FoCaLiZe, et formalisé comme suit :

Etat initial	Code FoCaLiZe
	**type trans = T_Init | ... | T_Final ;;** **...** **Species Class_name=** **let rec diagSM (t:trans, e:Self):(trans * Self) = match t with** **| T_Init -> …**

Etat final	Code FoCaLiZe
	**type trans = T_Init | ... | T_Final ;;** **...** **Species Class_name=** **let rec diagSM (t:trans, e:Self):(trans * Self) = match t with** **…** **| T_Final -> diagSM(T_0, e)** **| T_0 -> focalize_error("Fin de traitement") ;**

Tableau IV. 13: Transformation de l'état initial et l'état final.

4. Conclusion

Le diagramme d'états-transitions nous donne des évolutions possibles des états d'une classe. Cette définition montre la relation entre le diagramme de classe et le diagramme d'états-transitions, en raison de cette relation, nous avons défini dans ce chapitre les règles de transformation de diagramme de classe et leur diagramme d'états-transitions en FoCaLiZe.

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

CHAPITRE V

Implémentation

1. Introduction.
2. L'environnement de travail.
3. Processus d'Implémentation.
4. L'outil de transformation.
5. Exemple de transformation.
6. Conclusion.

1. Introduction

Dans ce chapitre, nous décrivons les différentes étapes de notre processus de transformation afin de transformer le modèle d'états-transitions UML vers FoCaLiZe.

Notre processus de transformation se compose de deux étapes:

- utilisation d'environnement graphique UML qui prend en charge le méta-modèle OMG pour créer notre modèle d'UML (diagramme d'états-transition) et de générer son format XMI (XML Meta data Interchange) correspondant.
- l'application de nos règles de transformation pour générer le code FoCaLiZe.

Pour mettre en œuvre notre processus de transformation, nous avons utilisé:

- L'environnement Eclipse avec le plugin Papyrus.
- XSLT (eXtensible Stylesheet Language Transformation).

2. L'environnement de travail

2.1 Eclipse

L'outil principal utilisé durant ces travaux est Eclipse, le logiciel Eclipse est un environnement intégré de développement (IDE) pour le langage Java (et d'autres langages), il peut être téléchargé sur le site www.eclipse.org, Il contient un espace de travail de base et un système de plug-in extensible pour personnaliser l'environnement **[26]**.

2.2 Papyrus

Papyrus est un outil d'édition graphique pour UML2 tel que défini par OMG, papyrus vise à mettre en œuvre 100% de la spécification OMG, Il est écrit pour être intégré de manière transparente dans l'environnement eclipse et fournit des diagrammes puissants pour créer des modèles UML **[27]**.

2.3 XSLT

XSLT 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, ...) **[28].**

La figure suivante éclaircit notre schéma de transformation basé sur l'utilisation d'une feuille de style XSLT.

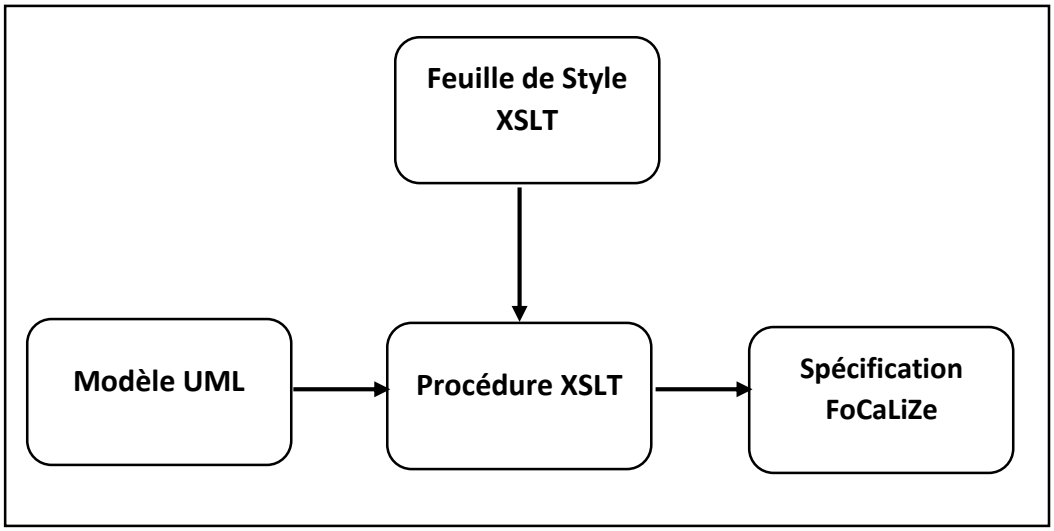


Figure V. 1: Représentation de rôle de feuille de style XSLT

3. Processus d'Implémentation

À l'aide de l'environnement eclipse et l'outil papyrus, nous avons développé une feuille de style XSLT qui permet de commander et de transformer le diagramme états-transitions UML en spécification FoCaLiZe, et nous avons utilisé un plugin qui permet de transformer le diagramme états-transitions UML (en utilisant la feuille de style XSLT) et compiler le code généré (en utilisant le compilateur FoCaLiZe), pour atteindre cet objectif, nous adoptons le processus d'implémentation suivant:

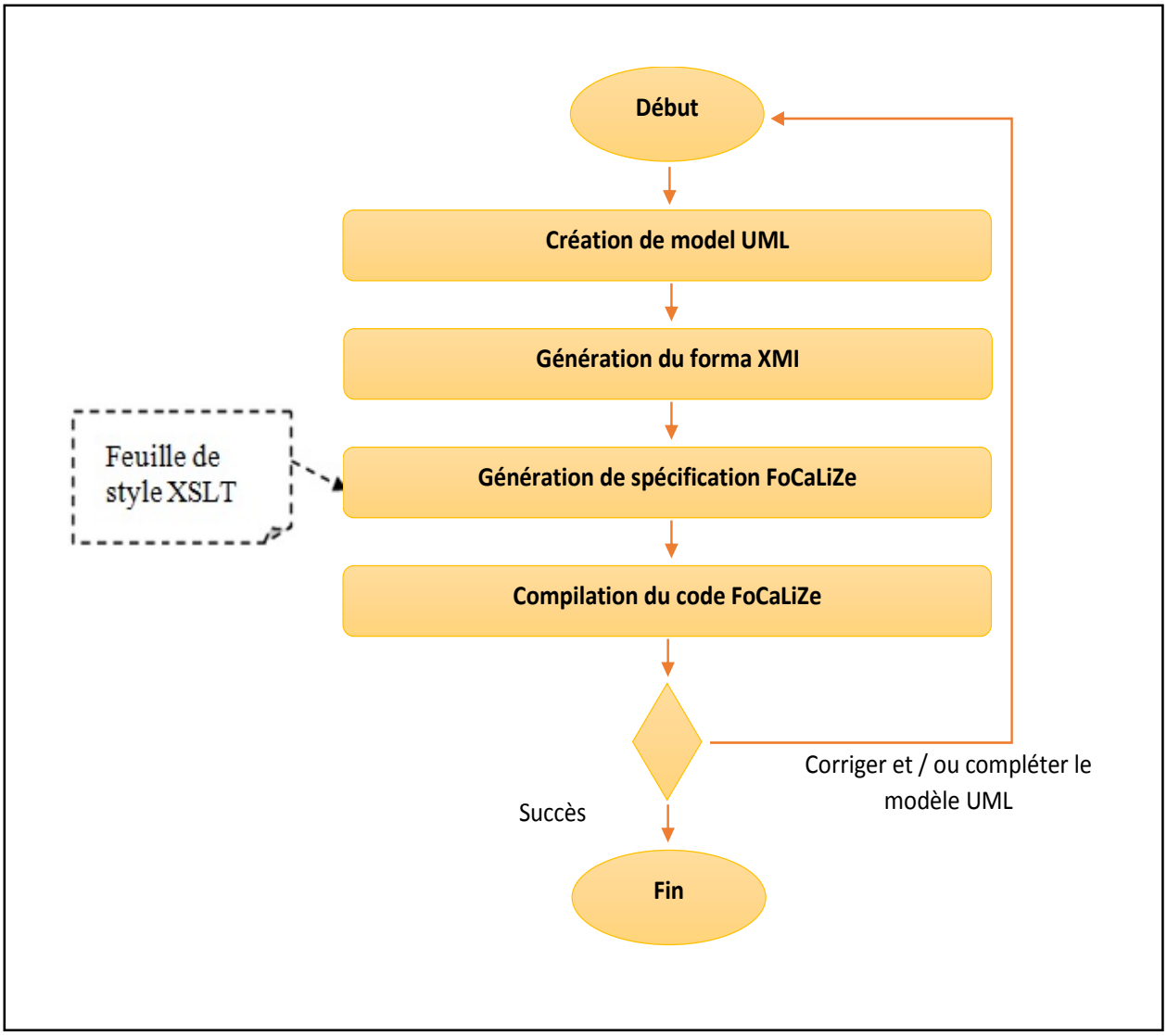


Figure V. 2: Transformation systématique de diagramme d'états-transitions UML en FoCaLiZe.

3.1 Création du modèle UML

Comme nous avons mentionné précédemment, à l'aide de l'environnement eclipse et des outils de papyrus, nous commençons par créer un diagramme de classes et diagramme d'états-transitions (La création de modèle UML se fait dans le même modèle **« model.di »**) (voir figure V.3), et génère son format XMI.

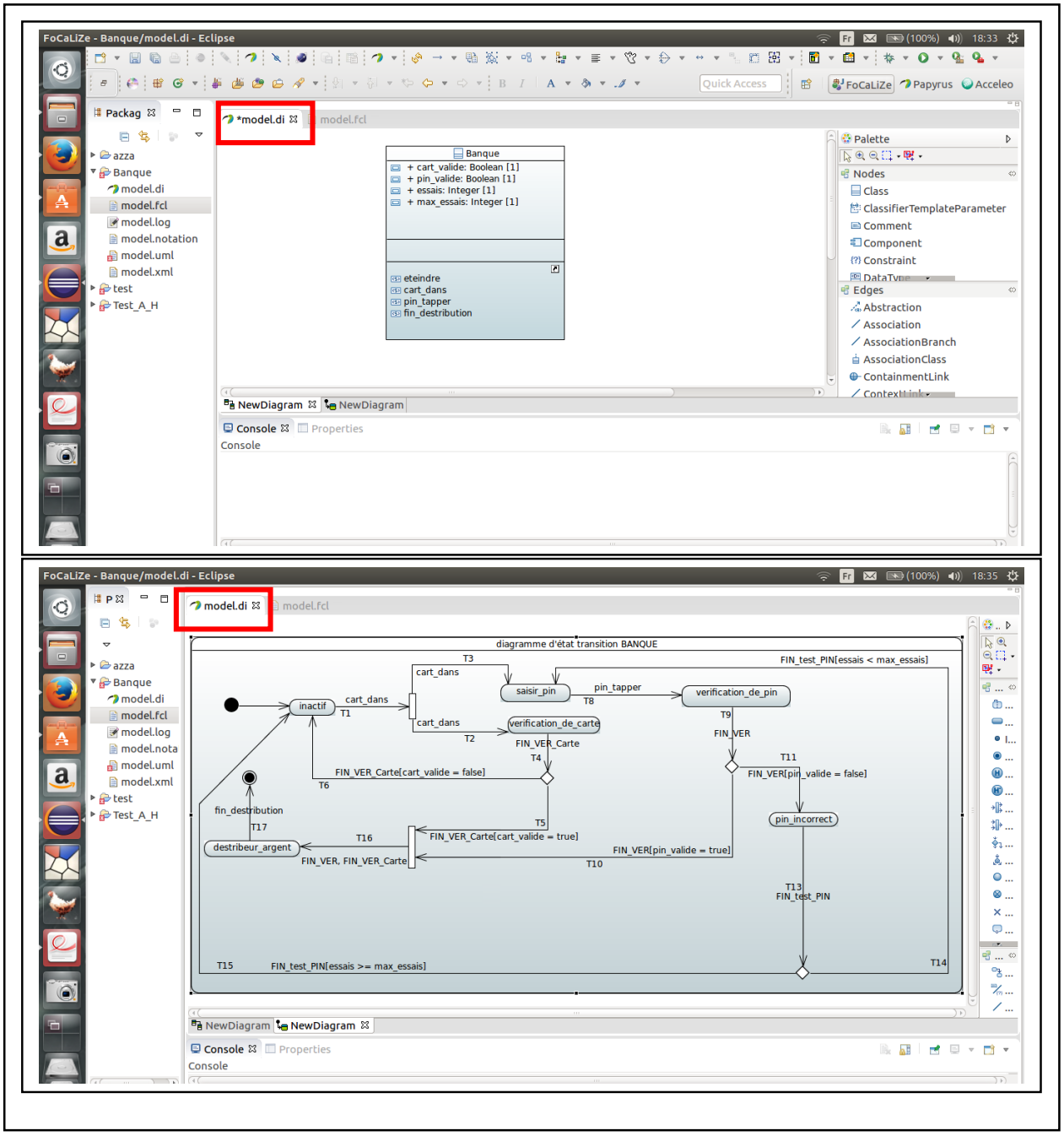


Figure V. 3: diagramme de classes et son diagramme d'états-transitions dans eclipse.

Pour générer le fichier XMI, nous sauvegardons le projet, le fichier XMI est enregistré automatiquement sous le nom **« model.uml »** (voir figure V.4).

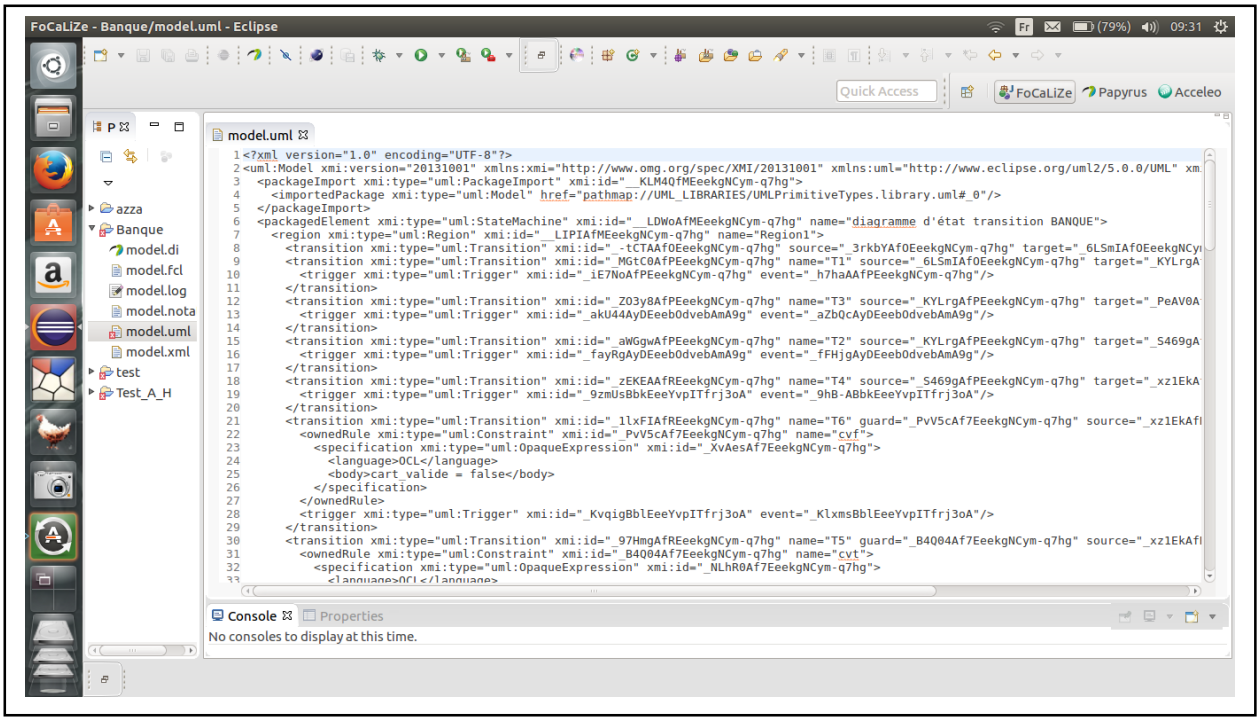


Figure V. 4: Exemple de format XMI.

3.2 Implémentation des règles de transformation

Dans cette étape, nous implémentons les règles de transformation (voir chapitre IV) pour générer un code FoCaLiZe. Pour terminer le processus d'implémentation, nous avons développé 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'états-transitions vers FoCaLiZe, basée sur le fichier XMI.

On résume le processus de transformation dans la figure V.5.

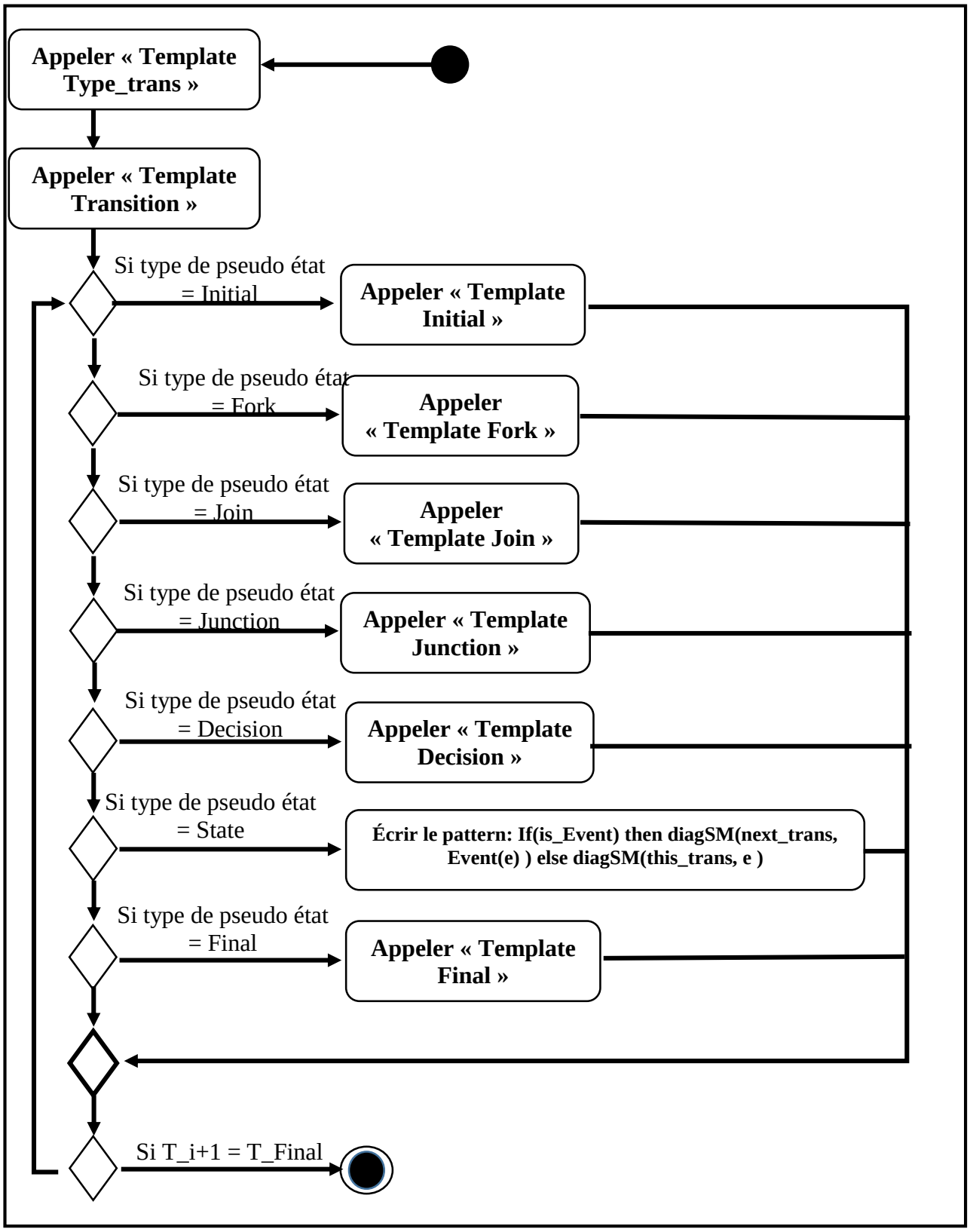


Figure V. 5: Implémentation des règles de transformation.

Les templates de notre feuille de style XSLT est expliqué dans le tableau V.1 :

Template	Description
« Type_trans »	Template définit la règle qui transformer les transitions à nouveaux type dans FoCaLiZe.
« Transition »	Template définit la règle de transformation de chaque pseudo état en UML vers FoCaLiZe selon les types existants : (décision, fork,…, etc).
« Initial »	Template définit la règle de transformation de 1ère transition de diagramme d'états-transitions UML vers FoCaLiZe
« Fork »	Template définit la règle de transformation de débranchement de diagramme d'états-transitions UML vers FoCaLiZe
« Join »	Template définit la règle de transformation de pseudo état de jointure de diagramme d'états-transitions UML vers FoCaLiZe
« Junction »	Template définit la règle de transformation de jonction de diagramme d'états-transitions UML vers FoCaLiZe
« Decision »	Template définit la règle de transformation de décision de diagramme d'états-transitions UML vers FoCaLiZe
« Final »	Template définit la règle de transformation de transition qui entrant à état final de diagramme d'états-transitions UML vers FoCaLiZe

Tableau V. 1: Les Templates de XSLT

4. L'outil de transformation

Cette étape contient des détails sur l'outil de transformation (l'installation et utilisation).

4.1 L'installation

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

- Environnement eclipse avec plugin papyrus.
- Compilateur FoCaLiZe.

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

4.1.1 Installation de plugin papyrus

Pour installer le plugin Papyrus voir le site Web suivant:

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

4.1.2 Installation de focalize

Pour installer FoCaLiZe, voir le lien :

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

Il requiert l'installation des outils externes suivants dans la racine de répertoire de FoCaLiZe (habituellement « Focalize »).

- **OCaml** (toute version récente => 3.11) :

Commande d'installation (sous ubunto): **#sudo apt-get install ocaml**

- **Coq** (toute version récente > = 8.1pl5):

Commande d'installation (sous ubunto): **#sudo apt-get install coq**

- **Zenon:** Il est disponible dans le site

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

4.1.3 Installation de plugin de transformation (UmlToFocalize)

Pour installer plugin « umlToFoCaLiZe », suivez les points suivants :

- Fermer l'environnement eclipse.
- Ajouter le plugin « umlToFoCaLiZe » dans le path « eclipse / plugins / ».
- Ajouter les fichiers XSLT (« TrieUMLOrginal.xsl » et « umlTofoclize.xsl ») dans le chemin « eclipse / ».
- Relancer l'eclipse.

4.2 Utilisation de l'outil

Dans cette étape, nous allons expliquer les techniques d'utilisation de notre outil de transformation.

- Premièrement: lancer Papyrus pour créer une modèle UML (voir figure V.6).

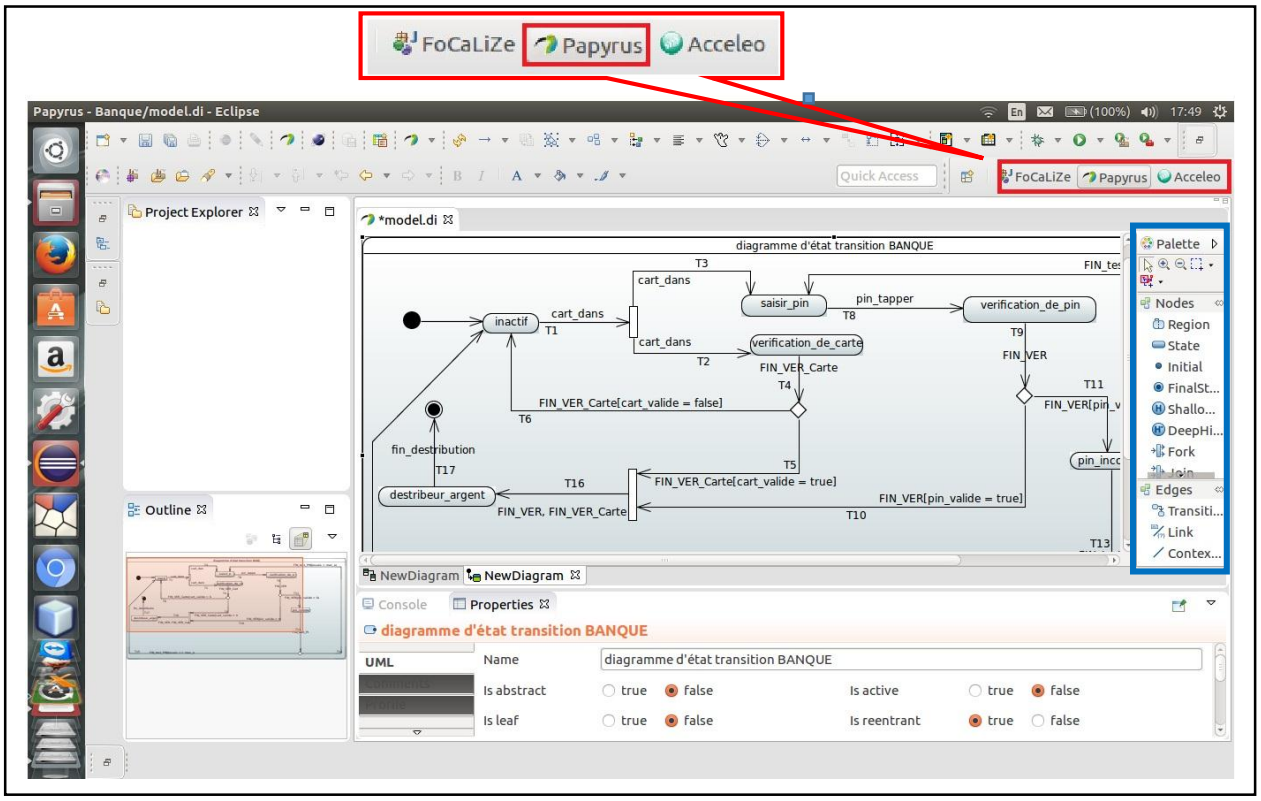


Figure V. 6: Modél UML créer par Papyrus.

- Deuxièmement: lancer FoCaLiZe en préparation à la transformation (voir figure V.7).

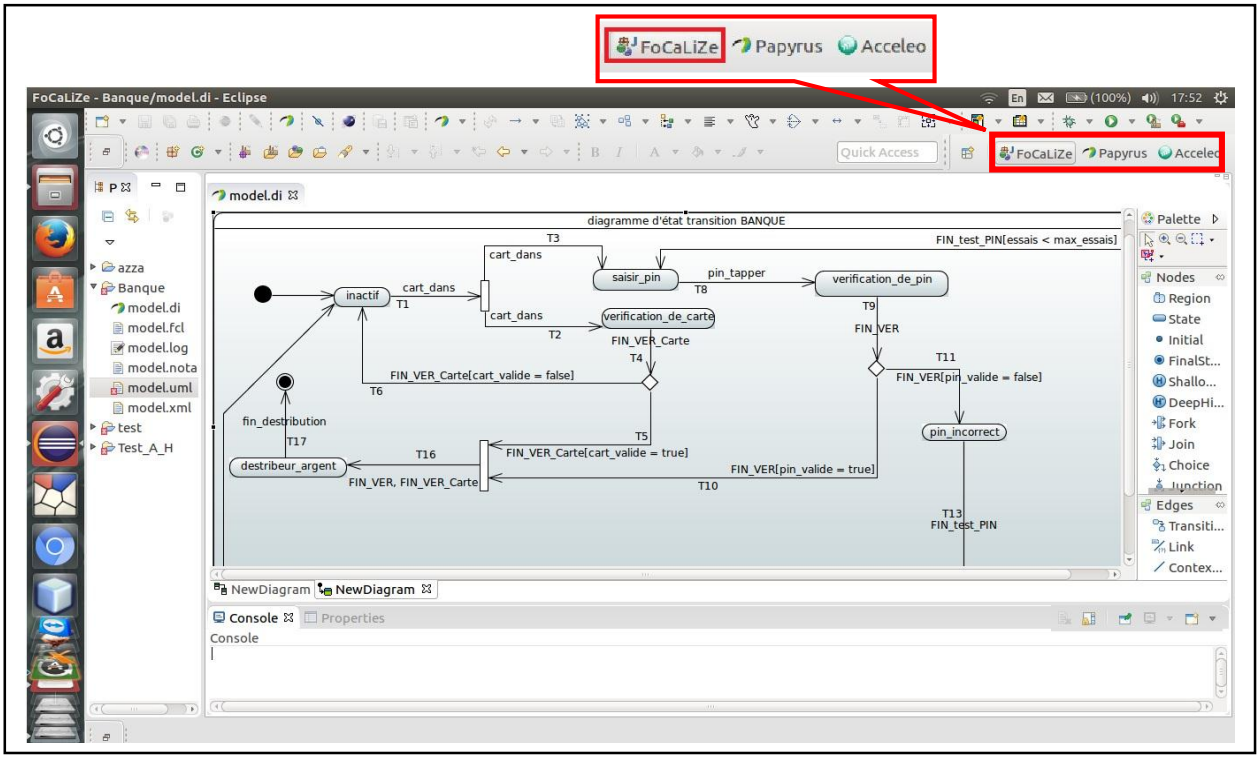


Figure V. 7: Lancement de FoCaLiZe.

Pour appliquer la transformation, on suit les étapes suivantes:

- une clique droite sur le fichier **model.uml**.
- 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.

La figure suivante montre cette étape:

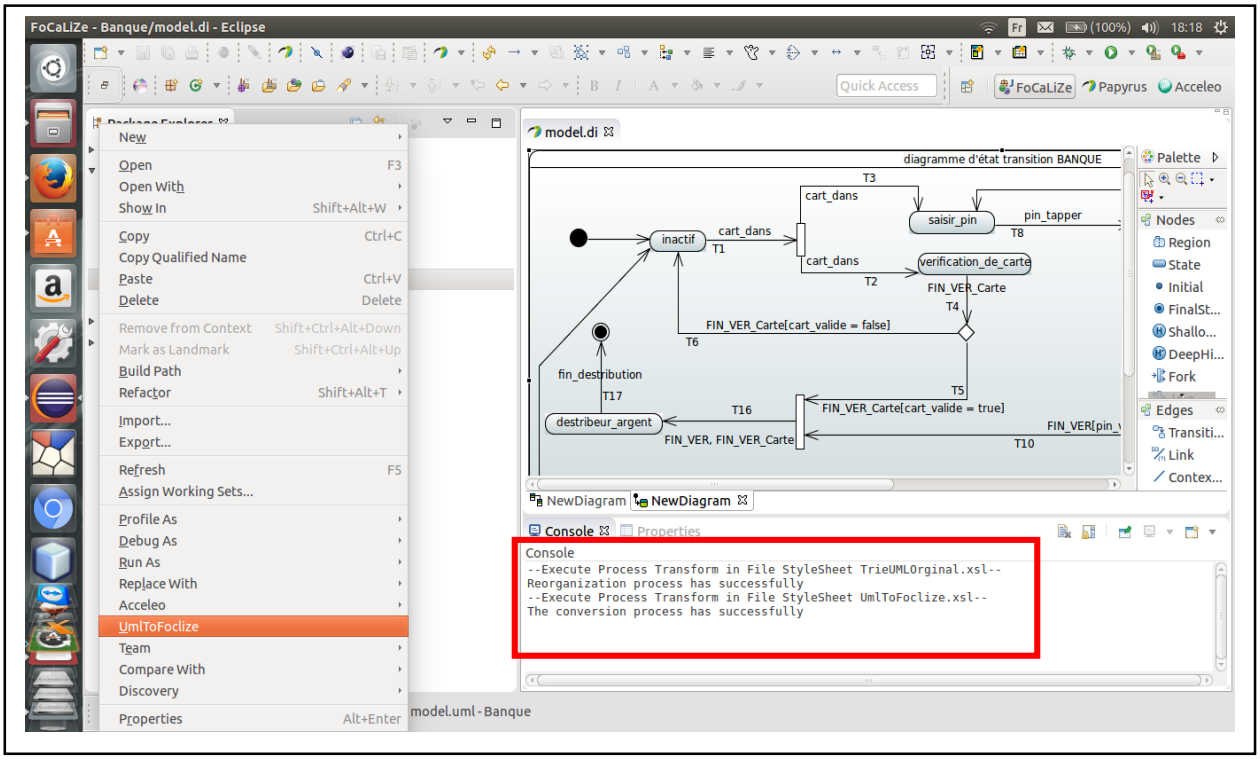


Figure V. 8: Génération de code FoCaLiZe.

Troisièmement: compilation de code FoCaLiZe.

Pour compiler le code FoCaLiZe, cliquer à 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.9).

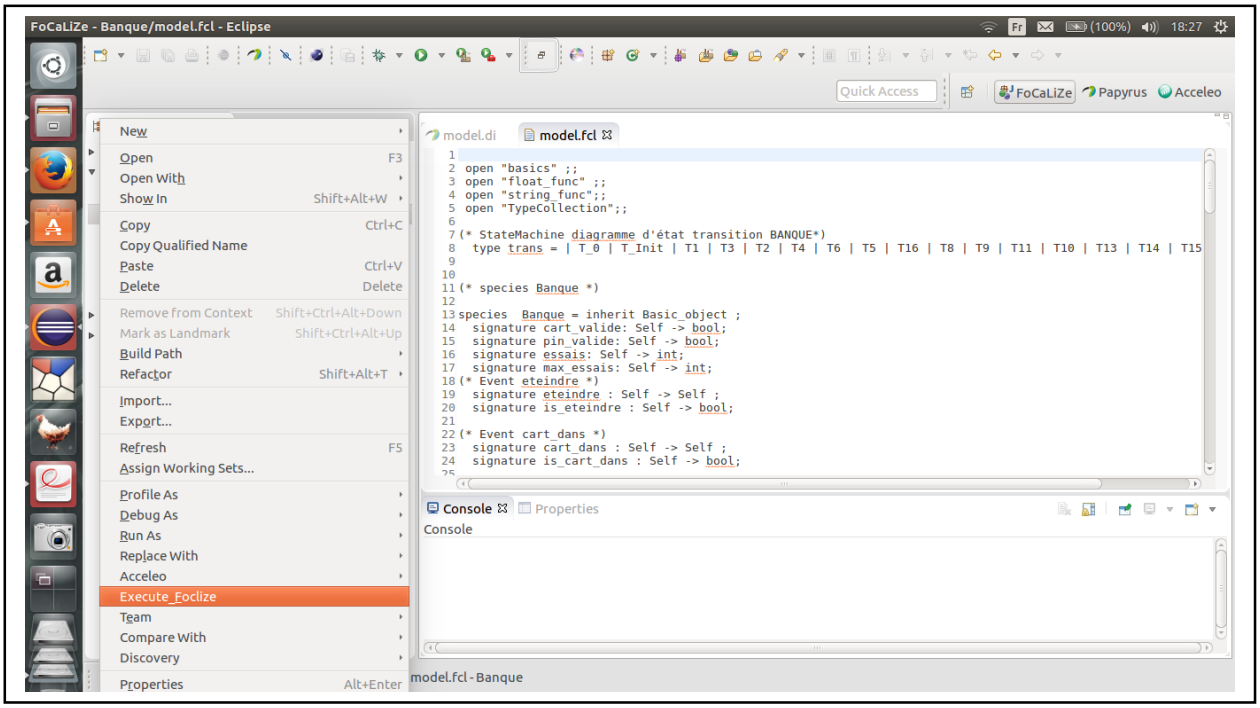


Figure V. 9: Compilation de code FoCaLiZe

5. Exemple de transformation de diagramme d'états-transitions de la classe « ATM »

Dans cette section, nous allons présenter un exemple qui décrit nos étapes de transformation en détail, pour illustrer notre transformation.

5.1 La création de modèle UML

Comme nous avons mentionné au début du chapitre V, la première étape de transformation est la création de modèle UML.

La figure V.10 présente la création de la classe « ATM », et la figure V.11 présente le diagramme d'états-transitions de la classe « ATM ».

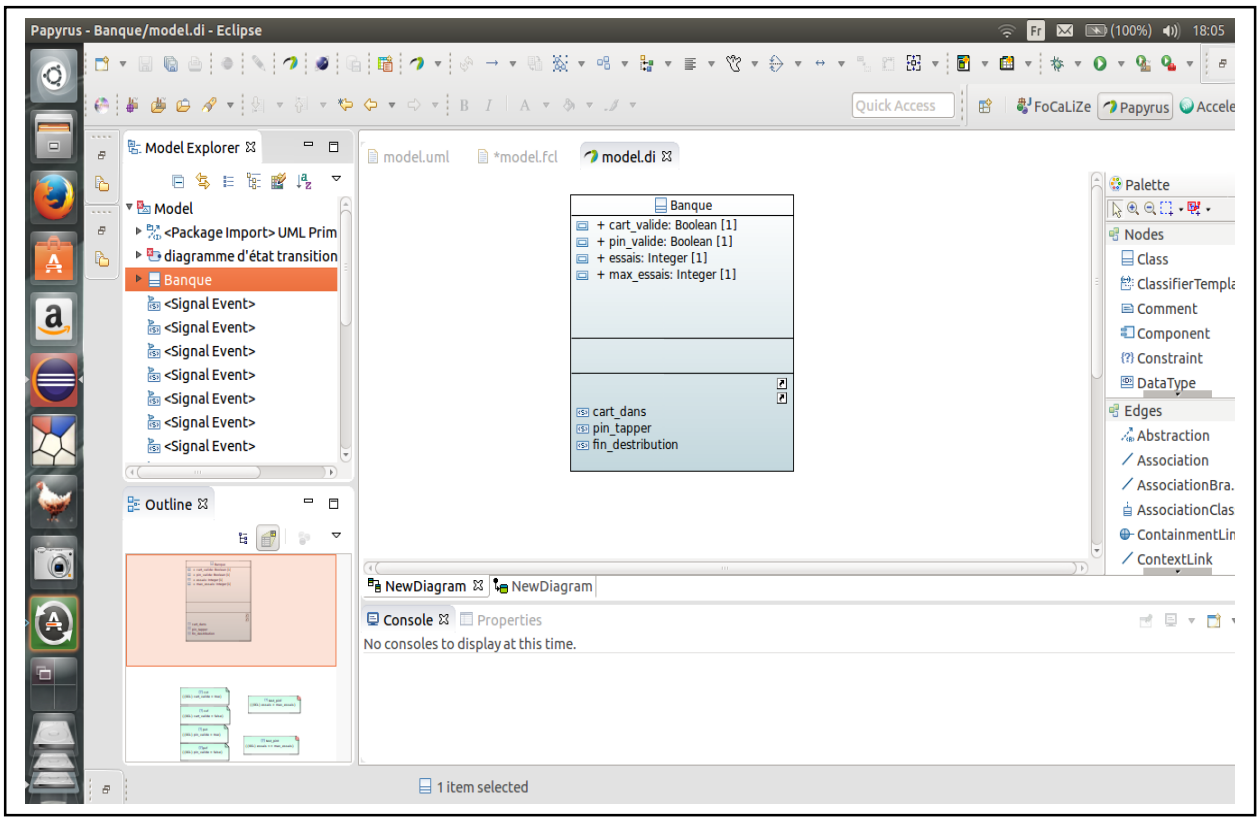


Figure V. 10: La classe ATM.

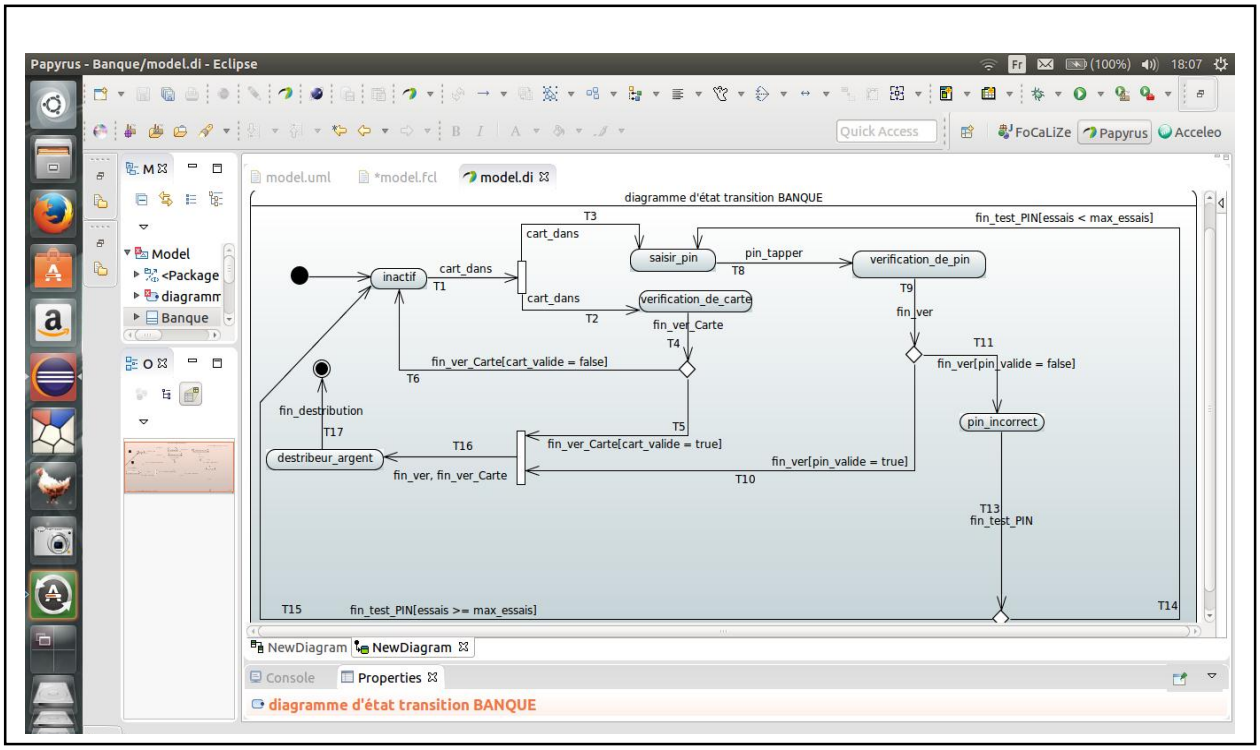


Figure V. 11: Diagramme d'états-transitions de la classe "ATM".

5.2 La génération de format XMI

Après l'enregistrement de notre modèle UML, le format XMI sera automatiquement enregistré dans le fichier « **model.uml** ». Voir la figure suivante (figure V.12).

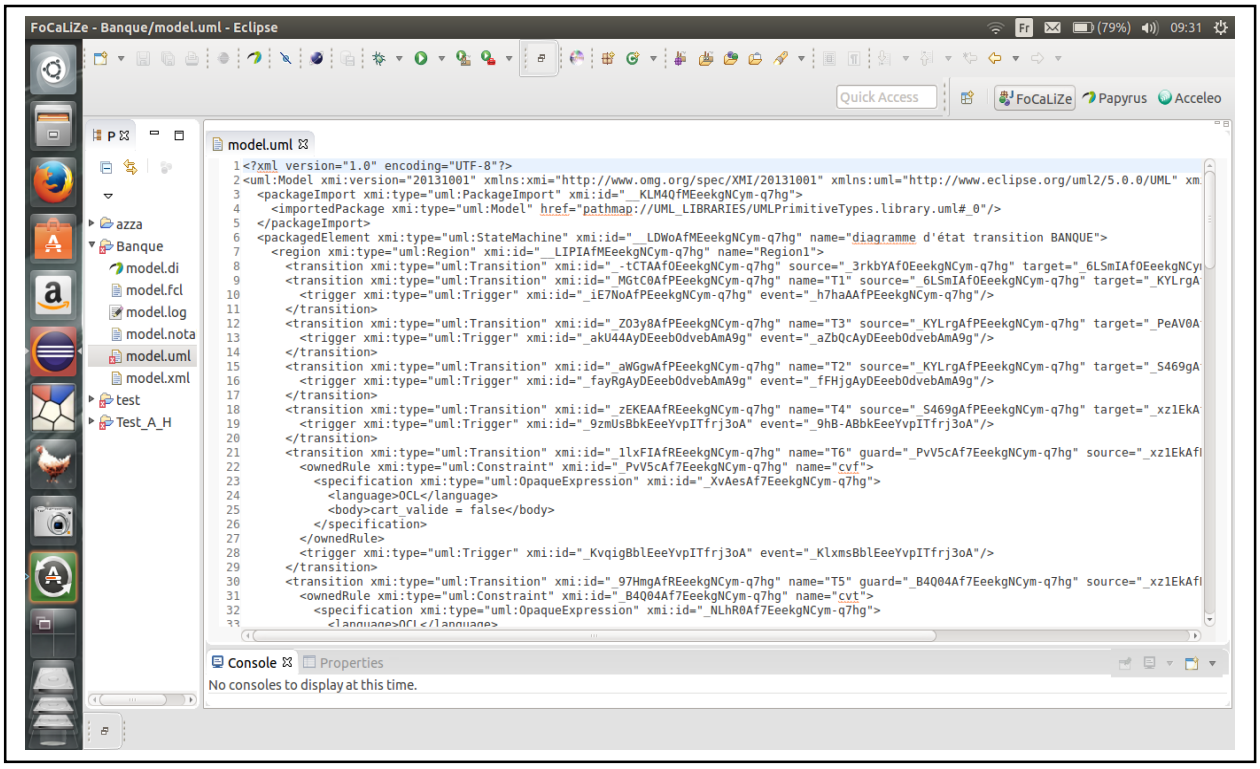


Figure V. 12: Le format XMI de diagramme de classe et d'états-transitions "ATM".

5.3 La génération de code FoCaLiZe

Dans le tableau V.2, le code FoCaLiZe est le résultat de transformation de diagramme d'état « ATM ».

```
open "basics" ;;

open "float_func" ;;

open "string_func";;

open "TypeCollection";;

(* StateMachine diagramme d'états-transitions ATM*)

type trans = | T_0 | T_Init | T1 | T3 | T2 | T4 | T6 | T5 | T16 | T8 | T9 | T11 | T10 | T13 | T14 | T15 |  T_Final ;;

(* species ATM *)
```

```
species  ATM = inherit Basic_object ;

signature cart_valide: Self -> bool;

signature pin_valide: Self -> bool;

signature essais: Self -> int;

signature max_essais: Self -> int;

(* Event cart_dans *)

signature cart_dans : Self -> Self ;

signature is_cart_dans : Self -> bool;

(* Event pin_tapper *)

signature pin_tapper : Self -> Self ;

signature is_pin_tapper : Self -> bool;

(* Event fin_destribution *)

signature fin_destribution : Self -> Self ;

signature is_fin_destribution : Self -> bool;

(* Event fin_ver *)

signature fin_ver : Self -> Self ;

signature is_fin_ver : Self -> bool;

(* Event fin_ver_Carte *)

signature fin_ver_Carte : Self -> Self ;

signature is_fin_ver_Carte : Self -> bool;

(* Event fin_test_PIN *)

signature fin_test_PIN : Self -> Self ;

signature is_fin_test_PIN : Self -> bool;

signature join_fun : Self -> Self -> Self;

let rec diagSM (t:trans, e:Self):(trans * Self) = match t with
```

```
| T_Init -> if(is_cart_dans(e)) then diagSM(T1, cart_dans(e))
            else diagSM(T1, e)
| T1 -> let x1 = snd(diagSM(T3, cart_dans(e))) and x2 = snd(diagSM(T2, cart_dans(e)) ) in
diagSM(T16,join_fun(x1,x2))
| T3 -> if(is_cart_dans(e)) then diagSM(T8, cart_dans(e))
        else diagSM(T3, e)
| T2 -> if(is_cart_dans(e)) then diagSM(T4, cart_dans(e))
        else diagSM(T2, e)
| T4 -> if(is_fin_ver_Carte(e)) then diagSM(T6, fin_ver_Carte(e))
        else if(is_fin_ver_Carte(e)) then diagSM(T5, fin_ver_Carte(e))
        else diagSM(T4, e)
| T6 -> if(is_fin_ver_Carte(e)) then diagSM(T1, fin_ver_Carte(e))
        else diagSM(T6, e)
| T5 -> if(is_fin_ver_Carte(e)) then diagSM(T_Final, fin_ver_Carte(e))
        else diagSM(T5, e)
| T16 -> if(is_fin_ver(e) && is_fin_ver_Carte(e)) then diagSM(T_Final, e)
          else diagSM(T16, e)
| T8 -> if(is_fin_ver(e)) then diagSM(T9, fin_ver(e))
        else diagSM(T8, e)
| T9 -> if(is_fin_ver(e)) then diagSM(T11, fin_ver(e))
        else if(is_fin_ver(e)) then diagSM(T10, fin_ver(e))
        else diagSM(T9, e)
| T11 -> if(is_fin_ver(e)) then diagSM(T13, fin_ver(e))
          else diagSM(T11, e)
| T10 -> if(is_fin_ver(e)) then diagSM(T_Final, fin_ver(e))
          else diagSM(T10, e)
```

```
| T13 -> if(is_fin_test_PIN(e)) then diagSM(T14, fin_test_PIN(e))
        else if(is_fin_test_PIN(e)) then diagSM(T15, fin_test_PIN(e))
        else diagSM(T13, e)
| T14 -> if(is_fin_test_PIN(e)) then diagSM(T8, fin_test_PIN(e))
        else diagSM(T14, e)
| T15 -> if(is_fin_test_PIN(e)) then diagSM(T1, fin_test_PIN(e))
        else diagSM(T15, e)
| T_Final -> diagSM(T_0, e)
| T_0 -> focalize_error("Fin de traitement") ;
signature new_ATM :bool -> bool -> int -> int -> Self ;
end;;
```

Tableau V. 2: Le code FoCaLiZe généré.

5.4 La compilation de code FoCaLiZe généré

Dans ce cas, le compilateur FoCaLiZe détecte la présence d'erreur.

- Si le compilateur FoCaLiZe trouve des erreurs dans le code, il indique les lignes responsables de ces erreurs.
- Si non, il nous donne ce message (voir tableau V.3).

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c /home/abdelaziz/workspace/ATM/model.ml
Invoking zvtov...
>> zvtov -zenon zenon -new  /home/abdelaziz/workspace/ATM/model.zv
Invoking coqc...
>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon /home/abdelaziz/workspace/ATM/model.v
```

Tableau V. 3: message de compilation de code FoCaLiZe

6. Conclusion

Dans ce chapitre, nous avons présenté un processus pour la transformation des modèles UML en FoCaLiZe. Le processus de transformation utilise un outil graphique UML, l'environnement FoCaLiZe et nos règles de transformation. L'approche proposée permet une transformation systématique du modèle UML en FoCaLiZe. Ensuite, un développeur FoCaLiZe peut compléter la spécification abstraite obtenue et fournir des preuves pour ses propriétés jusqu'à atteindre le code exécutable. Une implémentation du processus de transformation se fait par le développement d'une feuille de style XSLT décrivant les règles de transformation.

Conclusion générale

Conclusion général

Dans ce mémoire, nous avons établi une autre alternative pour la formalisation de diagramme d'états-transitions d'UML, en utilisant l'environnement FoCaLiZe.

Dans la formalisation proposée, le diagramme d'états-transitions d'une classe est formalisé par la définition d'une fonction récursive, les transitions sont modélisées par une énumération FoCaLiZe.

Pour mettre en œuvre la transformation proposée, nous avons utilisé une feuille de style XSLT spécifiant les règles de transformation d'un modèle UML exprimé dans le format d'échange XMI (généré par l'outil graphique Papyrus) dans FoCaLiZe.

Cette formalisation fonctionne naturellement avec la transformation des diagrammes de classes. Elle supporte des fonctionnalités UML comme l'héritage multiple et la paramétrisation. De cette manière, il est possible d'obtenir une transformation rigoureuse de diagramme d'états-transitions d'une classe, même si la classe est créée par héritage multiple ou créée par le paramétrage. Ce résultat est une amélioration significative par rapport aux transformations proposées par les autres méthodes formelles.

Comme perspective, nous envisageons de compléter notre transformation de diagramme d'états-transitions en considérons les contraintes OCL (permet la spécification des contraintes aux seins des diagrammes UML). Une contrainte OCL va correspondre à une propriété de l'espèce dérivée d'une classe UML. De cette manière, l'utilisation de prouveur automatique de théorèmes Zenon, permettre la détection automatiquement des éventuelles contradictions entre la spécification de classes et la spécification des diagrammes d'états-transitions.

Bibliographies

[1]: P. Roques, Les cahiers du programmeur UML 2.0 modéliser une application web, Eyrolles, 4e édition 2008.

[2]: OMG Unified Modeling Language Version 2.5, 2015. Disponible à : http://www.omg.org/spec/UML/2.5/PDF/

[3]: OMG Unified Modeling Languag: Superstructure, version 2.4,2011

[4]: P.A Muller, N. Geartner. Modélisation objet avec UML, Eyrolles, 2e édition 2000, Cinquième tirage 2004.

[5]: H.E. Erikson, M. Penker, B. Lyons, D. Fado, UML 2 ToolKit, Wiley publishing, USA, 2004.

[6]: James Rumbaugh, Ivar. Jacobson, Grady. Booch. The Unified Language Reference Manual UML,second edition, Addison-Wesley, USA, 2004.

[7]: James Rumbaugh,Ivar Jacobson,Grady Booch, Unified Modeling Language User Guide second edition, Addison-Wesley ,2005.

[8]: Pierre-Alain Muller Modélisation objet avec UML, Eyrolles 1997.

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

[10]: 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. PLAS - Seventh Workshop on Programming Languages and Analysis for Security, Jun 2012, Beijin, China. <hal 0077365>.

[11]: 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.

[12]: 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). Actes de la conférence.

[13]: 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). Publié dans LNCS.

[14]: Équipe de développement FoCaLiZe : Tutorial and reference manual, version 0.9.1. CNAM/INRIA/LIP6 (2016). Disponible à: http://focalize.inria.fr.

[15]: Kyriakos A, Behzad B, Geri G, et Indrakshi R, "UML2Alloy: A Challenging Model Transformation", article, 2007.

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

[17]: Idani Akram, B/UML : Mise en relation de spéciation B et de descriptions UML pour l'aide à la validation externe de développements formels en B, thèse de doctorat, Université Joseph-Fourier - Grenoble I, 2006.

[18]: Colin Snook et Michael Butler, Formal modelling and design aided by UML, Journal ACM Transactions on Software Engineering and Methodology (TOSEM), 2006.

[19]: Manuel C, Francisco D, Steven E, Santiago E, Patrick L, Narciso M, José M, Carolyn T, "Maude Manual", Version 2.7, 2015.

[20]: BOUDIA MALIKA, « Transformation des diagrammes d'états-transitions vers Maude », thèse de magistère, UNIVERSITE DE M'SILA, 2011.

[21]: John Anil Saldhana, Sol M. Shatz, Zhaoxia Hu. .Formalization of Object Behavior and Interactions from UML Models. Concurrent Software Systems Laboratory. Department of Computer Science .University of Illinois at Chicago. Disponible sur: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.3900&rep=rep1&type=pdf Ou http://www.cs.uic.edu/~shatz/papers/ijseke01.pd.

[22]: H Vangheluwe, J de Lara, "META-MODELS ARE MODELS TOO" Proceedings of the 2002 Winter Simulation Conference, Disponible sur: http://www.informs-sim.org/wsc02papers/076.pdf.

[23]: GHENDIR MABROUK Nacira et MENACEUR Khadija, "Automatic transformation tool of UML classe diagrams into FoCaLiZe", Thése de Master, UNIVERSITE ECHAHID HAMMA LAKHDAR EL OUED, 2015.

[24]: FERHAT HAMIDA Imane et SALHI Noura, « Transformation de diagramme d'activité UML vers FoCaLiZe », Thése de Master, UNIVERSITE ECHAHID HAMMA LAKHDAR EL OUED, 2016.

[25]: LEMMOUCHI Nacer-Eddine et GHERAISSA Tarek, « Transformation of OCL Constraints to FoCaLiZe Specifications », Thése de Master, UNIVERSITE ECHAHID HAMMA LAKHDAR EL OUED, 2016.

[26]: Tutorials Point, « eclipse tutorial », 2015, Disponible sur https://www.tutorialspoint.com/eclipse/eclipse_tutorial.pdf.

[27]: Sébastien Gérard, « PAPYRUS USER GUIDE SERIES », 2011, Disponible sur http://www.eclipse.org/papyrus/resources/PapyrusUserGuideSeries_AboutUMLProfile_v1.0.0_d20120606.pdf.

[28]: Matthias Biehl, "Literature Study on Model Transformations", 2010, Disponible sur https://pdfs.semanticscholar.org/3d6c/6bfc306c29a16afd09bb649bd21e178cd008.pdf.