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
Faculté des Sciences Exactes et Informatique
Département d'Informatique

Mémoire de fin d'études

Présenté pour l'obtention du Diplôme de MASTER ACADEMIQUE

Option : Systèmes Distribués et Intelligence Artificielle

Thème

Réalisation d'un outil de transformation automatique du diagramme d'activité UML vers Maude

Présenté par :
Nouir Mohamed
Bennouna Ahmed Oualid

Encadré par :
Dr. Boucherit Ammar

Soutenu le ... Septembre 2022, Devant le jury composé de :

Mr. Abbas Messaoud	MCA Univ El-Oued	Président
Mr. Bali Ahmad	MCA Univ El-Oued	Examinateur
Mr. Boucherit Ammar	MCA Univ El-Oued	Rapporteur

Année Universitaire : 2021/2022

إهداء

إلى من علمنا العطاء دون انتظار
إلى من نحمل اسمه بكل افتخار
إلى من تحمل وكدّ من أجل مستقبلنا
إليك مصدر همتنا -أبي الغالي-

إلى من أوصى بها الرحمن
إلى من منحتنا العون والدفء والحنان
إلى من كان دعاؤها لنا سرا وجهرا دعما وسندا لنا
إليك شمعتي المضيئة -أمي الغالية-

إلى من حبهم يجري في عروقنا ويبتهج عند ذكراهم فؤادنا
إلى مشاطري أفراحنا وأحزاننا
إليكم رياحين حياتنا -إخوتي الأعزاء-

إلى من ضاقت السطور عن ذكرهم فوسعهم قلبنا ... إليكم يا من لاتفارقون الوجدان -أصدقائي-

إلى أستاذنا و مشرف مذكرتنا ... إليك كل الإحترام و التقدير و الشكر و العرفان على جميع مجهوداتك في اتمام هذا العمل -الأستاذ الكريم بوشريط عمار-

محمد - وليد

Résumé

Le langage UML est un langage destiné à la modélisation et la conception des applications Informatiques, mais son principal critique est l'absence de bases formelles permettant l'application des techniques de vérifications formelles, car il n'a pas d'outils intégrés ou dédiés pour la vérification des propriétés de ses diagrammes.

Dans ce contexte, Maude est un langage déclaratif et formel de hautes performances basé sur la logique de réécriture, et qui peut jouer ce rôle de manière satisfaisante à cause de sa base purement algébrique très puissante pour décrire le comportement de systèmes concurrents ainsi que la batterie des outils de vérification formelle qu'il possède tel que son model-checker (Vérificateur de modèle).

Dans notre mémoire nous avons réalisé un outil de transformation automatique de diagramme d'activité UML vers la logique de réécriture exprimé sous Maude, pour servir une aide à la vérification et la validité des modèles proposés pour le développement des systèmes Informatique.

Mots clés : Diagramme d'activité, Maude, Transformation automatique, Logique de réécriture.

Abstract

The UML language is a language intended for the modeling and design of computer applications, but its main critic is the absence of formal bases allowing the application of formal verification techniques, because it has no integrated or dedicated tools for checking the properties of its diagrams.

In this context, Maude is a high performance declarative and formal language based on rewriting logic that can play this role satisfactorily because of its very powerful algebraic basis to describe the behavior of concurrent systems as well as the several verification tools it owns such as its LTL model-checker and Resolution / Inductive Theorem prover.

In our thesis, we have implemented a tool for the automatic transformation of UML activity diagrams into the rewriting logic expressed under Maude, to serve as an aid to the verification and validity of the models proposed for the development of computer systems.

Keywords : Activity diagram, Maude, Transformation, Rewriting logic.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

UML est un langage de modélisation de développement à usage général dans le domaine du génie logiciel qui fournit un moyen standard de visualisation de la conception d'un système. La création d'UML a été motivée à l'origine par le désir de normaliser les systèmes de notation et les approches disparates de la conception de logiciels afin d'offrir une vue synthétique, structurante et intuitive du système.

Autrement dit, UML offre un moyen de visualiser les vues architecturales et dynamiques d'un système dans l'ensemble des diagrammes, y compris : toutes activités, composants individuels du système, et leurs interactions avec d'autres composants logiciels, ... etc.

En plus, la modélisation et l'analyse de ces systèmes dans les premières étapes de la conception peuvent grandement améliorer leur développement. En revanche, le manque d'une sémantique précise limite l'emploi des techniques de validation et de vérification. Par conséquent, Il est fortement nécessaire de chercher une alternative pour compléter ce manque afin de puisse analyser formellement sur les systèmes étudiés.

D'après notre point de vue dans ce travail, nous pensons que la transformation des diagrammes en une méthode formelle adéquate peut résoudre le problème de la vérification et de la validation. Pratiquement, le système de Maude peut jouer ce rôle de manière satisfaisante. C'est un langage algébrique très puissant pour décrire le comportement de systèmes concurrents. Nous l'avons volontairement choisi car il offre les mécanismes appropriés pour décrire les composants ainsi que tous les types d'associations de diagrammes UML.
Mais, il n'est pas aussi facile dans notre cas d'effectuer la transformation pour tous les diagrammes UML, et c'est pourquoi nous nous limitons au diagramme d'activité.

Notre approche de transformation est basée principalement sur l'approche objet sous Maude où on a remarqué la facilité de spécification des constructeurs (composants) du diagramme d'activité.

Notre mémoire est composé de quatre chapitres. Le premier a été consacré pour introduire les diagrammes d'activités et leurs intérêts dans le processus de développement. Dans le deuxième chapitre, on a présenté les concepts de base de la logique de réécriture et le système Maude. Le troisième chapitre présente notre approche de transformation des diagrammes d'activités vers Maude. Enfin, un chapitre a été consacré pour la présentation des aspects d'implémentation et des outils utilisés durant le processus de développement.

Chapitre 1

Les diagrammes d'Activités

1.1 Introduction

UML (Unified Modeling Langage, traduisez "langage de modélisation objet unifié") est né de la fusion des trois méthodes qui ont le plus influencé la modélisation objet au milieu des années 90 : OMT, Booch et OOSE.

Uml (Unified Modeling Langage) est un langage de modélisation graphique (n'est pas une méthode) qui a été publié par l'OMG (Objet management groupe). UML comporte 14 types de diagrammes. Il permet de représenter, d'illustrer et de communiquer différents aspects d'un système d'information.

Dans ce chapitre, nous exposons les concepts fondamentaux de l'UML en nous concentrant sur le diagramme d'activité.

1.2 Modélisation UML

1.2.1 Introduction

La modélisation UML fournit un ensemble d'outils permettant de représenter l'ensemble des éléments du monde objet (classes, objets, ...) ainsi que les liens qui les relie.

Toutefois, étant donné qu'une seule représentation est trop subjective, UML fournit un moyen astucieux permettant de représenter diverses projections d'une même représentation grâce aux vues.

Comme UML est un langage de modélisation objet, nous allons pour la suite dire ce que c'est la modélisation et à quoi il sert.

1.2.2 UML et la modélisation

1.2.2.1 Qu'est-ce que la modélisation

Modéliser est une représentation abstraite et simplifiée (i.e. qui exclut certains détails), d'une entité (phénomène, processus, système, etc.) du monde réel en vue de le décrire, de l'expliquer ou de le prévoir. Modéliser est synonyme de théorie, mais avec une connotation pratique : Modéliser c'est une théorie orientée vers l'action qu'elle doit servir.
Concrètement, modéliser permet de réduire la complexité d'un phénomène en éliminant les détails qui n'influencent pas son comportement de manière significative.

1.2.2.2 But de la modélisation

La modélisation a plusieurs buts, dont certains sont les suivants :

- Modéliser un système avant sa réalisation permet de mieux comprendre le fonctionnement du système. C'est également un bon moyen de maîtriser sa complexité et d'assurer sa cohérence. Un modèle est un langage commun, précis, qui est connu par tous les membres de l'équipe et il est donc, à ce titre, un vecteur privilégié pour communiquer. Cette communication est essentielle pour aboutir à une compréhension commune aux différentes parties prenantes (notamment entre la maîtrise d'ouvrage et la maîtrise d'œuvre informatique) et précise d'un problème donné.
- La modélisation permet de réduire la complexité du système en ignorant les détails qui n'influencent pas son comportement de manière significative.
- Un modèle est un langage commun, précis qui est connu par tous les membres de l'équipe, donc c'est un vecteur privilégié pour communiquer.
- La modélisation permet de décomposer les tâches d'un système et de les automatiser, c'est également un facteur de minimisation et réduction des coûts et des délais.

1.2.2.3 Différent types de la modélisation

La classification de la modélisation peut se faire selon le degré du formalisme des langages ou des méthodes impliquées dans le processus de la modélisation. Ainsi, la modélisation peut être considérée comme étant formelle, semi-formelle ou informelle . La table de le tableau 1.1 ci-dessous présente une définition des catégories de langages ainsi que des exemples de langages ou de méthodes qui l'utilisent.

1.2.2.3.1 Modélisation Informelle : Le processus de modélisation informelle à base de langages informels, se justifie selon pour [1] plusieurs raisons :

- La facilité de compréhension du langage permet des consensus entre les personnes qui spécifient et celles qui commandent un logiciel.
- elle représente une manière familière de communication entre personnes.

Par ailleurs, l'utilisation d'un langage informel rend la modélisation imprécise et parfois ambigüe. Le caractère informel de cette approche rend difficile toute tentative de standardisation.

1.2.2.3.2 Modélisation Semi-Formelle : Le processus de modélisation semi-formelle est basé sur un langage textuel ou graphique pour lequel une syntaxe précise est définie [1]. La sémantique d'un tel langage et souvent assez faible. Néanmoins, ce type de modélisation permet d'effectuer des contrôles et de réaliser des automatisations pour certaines tâches.

La plupart des méthodes de modélisation semi-formelles s'appuient fortement sur des langages graphiques. Ceci se justifie par la puissance expressive du modèle graphique. Par ailleurs, l'appui de la modélisation semi-formelle (tels que : UML) sur des langages graphiques, permet la production de modèles assez faciles à interpréter. Cependant, cette modélisation soufre de la déficience des aspects sémantiques impliqués dans l'approche. Afin de palier aux insuffisances de cette approche, l'utilisation de contraintes a été introduite. Dans ce cadre, nous citons à titre d'exemple les travaux de S. Cook et J. Daniels [2].

1.2.2.3.3 Modélisation Formelle : Une méthode formelle est un processus de développement rigoureux basé sur des notations formelles avec une sémantique définie. Le principal avantage des aspects formelles est leur capacités à exprimer une signification précise, permettant ainsi des vérifications de la cohérence et de la complétude d'un système. Par exemple J. P. Bowen et M. C. Hinchey [3] montrent qu'avec une traduction appropriée, les méthodes formelles peuvent aider à la compréhension d'un système par un utilisateur.

1.3 Langage UML

1.3.1 Définition

Ce n'est pas vraiment un langage, plutôt une trousse à outils qui offre des diagrammes normalisés pour décrire (modéliser) un système. Ils sont très utiles dans la phase de conception, pour produire un système cohérent, et servent ensuite de documentation de référence pour ceux qui vont réaliser le système, ou étudier des impacts d'évolution du système. Les différentes "vues" UML permettent de lister les besoins fonctionnels jusqu'à leur traduction en solution logicielle (architecture, objets et propriétés, infrastructure, échanges de données, etc.), sans limite en terme de niveau de détails. Malheureusement les méthodes "agiles" ont tendance à sacrifier cette étape de réflexion pourtant indispensable à la qualité d'un projet.

UML unifie également les natations et les concepts orientés objet (voir Historique d'UML sur la figure 1.1).Il ne s'agit pas d'une simple notation graphique, car les concepts transmis par un diagramme

Catégories de Langages			
Langage Informel		**Langage Semi-Formel**	**Langage Formel**
Simple	*Standardisé*		
Langage qui n'a pas un ensemble complet de règles pour restreindre une construction	Langage avec une structure, un format et des règles pour la composition d'une construction.	Langage qui a une syntaxe définie pour spécifier les conditions sur lesquelles les constructions sont permises.	Langage qui possède une syntaxe et une sémantique définies rigoureusement. Il existe un modèle théorique qui peut être utilisé pour valider une construction.
Exemples de Langages ou Méthodes			
Langage Naturel.	Texte Structuré en Langage Naturel.	Diagramme Entité-Relation, Diagramme à Objets.	Réseaux de Petri, Machines à états finis, VDM, Z.

TAB. 1.1 : Les types de modélisation.

ont une sémantique précise et sont porteurs de sens au même titre que les mots d'un langage.

La figure ci-dessous, présente l'évolution des versions d'UML.

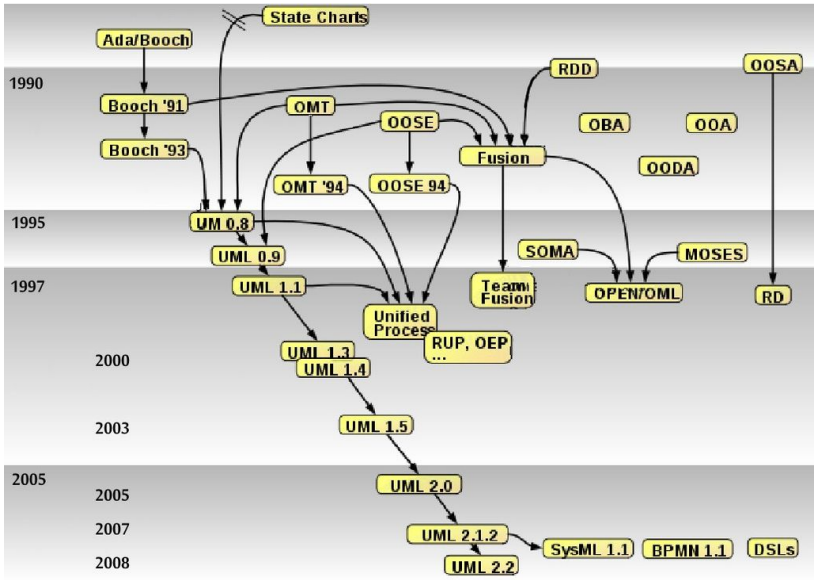


FIG. 1.1 : Historique d' UML.

1.3.2 Utilisation

UML est destiné à faciliter la conception des documents nécessaires au développement d'un logiciel orienté objet, comme standard de modélisation de l'architecture logicielle. Les différents éléments représentables sont :

- Activité d'un objet/logiciel
- Acteurs

- Processus
- Schéma de base de données
- Composants logiciels
- Réutilisation de composants.

Il est également possible de générer automatiquement tout ou partie du code, par exemple en langage Java, à partir des documents réalisés.

1.3.3 Diagrammes UML

Les 14 diagrammes UML sont dépendants hiérarchiquement et se complètent, de façon à permettre la modélisation d'un projet tout au long de son cycle de vie (voir la figure 1.2).

La figure 1.2 représente les 14 diagrammes d'UML.

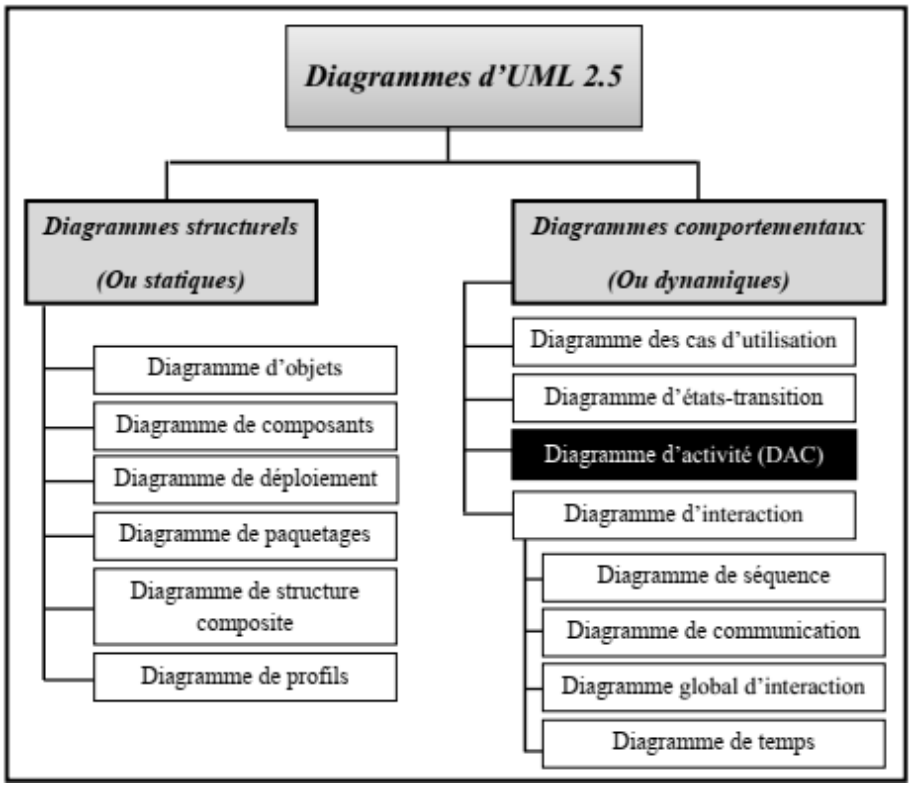


Fig. 1.2 : La hiérarchie des diagrammes UML 2.0.

1.3.3.1 Diagrammes structurels

Les diagrammes structurels ou statiques rassemblent [4] [5] :

- **Diagramme de classes (Class diagram) :** Ce diagramme est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que leurs relations. Ce diagramme fait partie de la partie statique d'UML, ne s'intéressant pas aux aspects temporels et dynamiques.
- **Diagramme d'objets (Object diagram) :** Le diagramme d'objet permet de représenter les instances des classes, c'est-à-dire des objets. Comme le diagramme de classes, il exprime les relations qui existent entre les objets, mais aussi l'état des objets, ce qui permet d'exprimer des contextes d'exécution. En ce sens, ce diagramme est moins général que le diagramme de classes.

- **Diagramme de composants :** Le diagramme de composants décrit l'organisation du système du point de vue des éléments logiciels comme les modules (paquetages, fichiers sources, bibliothèques, exécutables), des données (fichiers, bases de données) ou encore d'éléments de configuration (paramètres, scripts, fichiers de commandes). Ce diagramme permet de mettre en évidence les dépendances entre les composants (qui utilise quoi).
- **Diagramme de déploiement :** un diagramme de déploiement est une vue statique qui sert à représenter l'utilisation de l'infrastructure physique par le système et la manière dont les composants du système sont répartis ainsi que leurs relations entre eux. Les éléments utilisés par un diagramme de déploiement sont principalement les nœuds, les composants, les associations et les artefacts. Les caractéristiques des ressources matérielles physiques et des supports de communication peuvent être précisées par stéréotype.
- **Diagramme de paquetage :** Le paquetage représente un ensemble homogène d'élément de système (classe, composant, etc.) permettant de regrouper et d'organiser les éléments dans le modèle UML.
- **Diagramme de structure composite :** Le diagramme de structure composite permet de décrire sous forme de boîte blanche les relations entre les composants d'une seule classe (depuis UML 2.x).
- **Diagramme de profils :** permet de spécialiser, de personnaliser pour un domaine particulier un Meta-modèle de référence d'UML (depuis UML 2.2).

1.3.3.2 Diagrammes comportementaux

Les diagrammes de comportement représentent l'aspect dynamique du système. Il met l'accent sur ce qui doit se passer dans le système modélisé. Étant donné que les diagrammes de comportement illustrent le comportement d'un système, ils sont largement utilisés pour décrire la fonctionnalité des systèmes logiciels [7] [6].

- **Diagramme de cas d'utilisations :** Les cas d'utilisation sont une technique de description du système étudié selon le point de vue de l'utilisateur. Ils décrivent sous la forme d'actions et de réactions le comportement d'un système. Donc, le diagramme des cas d'utilisation, permet d'identifier les possibilités d'interaction entre le système et les acteurs. Il permet de clarifier, filtrer et organiser les besoins.
- **Diagramme d'activités :** Le diagramme d'activité est un diagramme comportemental d'UML, permettant de représenter le déclenchement d'événements en fonction des états du système et de modéliser des comportements parallélisables (multi-threads ou multi-processus). Le diagramme d'activité est également utilisé pour décrire un flux de travail (workflow).
- **Diagramme d'états-transitions :** Diagramme d'états ou machine à états Consiste à déerire l'évolution d'un système, ou par extension de tout ce qu UML peut définir comme objet : classe, composant, package, noeud, collaboration et cas d'utilisation. Cette évolution est rythmée par les états, qui représentent la configuration d'un système pendant laquelle il réagira toujours de la même façon aux sollicitations externes, et par les transitions qui concement les instants fugag de changements d'états [8].
- **Diagramme de séquences :** Ce diagramme permet de montrer les interactions d'objets dans le cadre d'un scénario d'un diagramme des cas d'utilisation. Dans un souci de simplification, on représente l'acteur principal à gauche du diagramme, et les acteurs secondaires éventuels à droite du système. Le but est de décrire comment se déroulent les interactions entre les acteurs ou objets.
- **Diagramme de communication :** C'est une représentation simplifiée d'un diagramme de séquence, en se concentrant sur les échanges de messages entre les objets.
- **Diagramme global d'interaction :** permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences (variante du diagramme d'activité).

- **Diagramme de temps :** un diagramme de temps est un type de diagrammes d'interaction dédié aux contraintes temporelles prises en compte dans l'écriture et la structure d'un logiciel. ils sont utilisés pour expliciter visuellement les divers comportements des objets d'un système au cours d'une période donnée.

1.4 Diagramme d'activité

1.4.1 Présntation

Les diagrammes d'activités permettent de mettre l'accent sur les traitements. Ils sont donc particulièrement adaptés à la modélisation du cheminement de flots de contrôle et de flots de données. Ils permettent ainsi de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.

Les diagrammes d'activités sont relativement proches des diagrammes d'états-transitions dans leur présentation, mais leur interprétation est sensiblement différente. Les diagrammes d'états-transitions sont orientés vers des systèmes réactifs, mais ils ne donnent pas une vision satisfaisante d'un traitement faisant intervenir plusieurs classeurs et doivent être complétés, par exemple, par des diagrammes de séquence. Au contraire, les diagrammes d'activités ne sont pas spécifiquement rattachés à un classeur particulier. On peut attacher un diagramme d'activités à n'importe quel élément de modélisation afin de visualiser, spécifier, construire ou documenter le comportement de cet élément.

1.4.2 Intérêts des diagrammes d'activités

(1) Utiliser le mécanisme de synchronisation pour représenter les successions d'états synchrones, alors que les diagrammes d'états transitions sont utilisés principalement pour représenter les suites d'états asynchrones.

(2) Utiliser des transitions automatiques évite la nécessité d'existence d'évènement de transition pour avoir un changement d'états.

(3) Modéliser un workflow dans un cas d'utilisation, ou entre plusieurs cas d'utilisations.

(4) Définir avec précision les traitements qui ont cours au sein du système, Certains algorithmes ou calculs nécessitent de la part du modélisateur une description poussée.

(5) Spécifier une opération (décrire la logique d'une opération).

(6) Le diagramme d'activité est le plus approprié pour modéliser la dynamique d'une tâche ou d'un cas d'utilisation, lorsque le diagramme de classe n'est pas encore stabilisé.

(7) Représenter graphiquement le comportement interne d'une opération, d'une classe ou d'un cas d'utilisation sous forme d'une suite d'actions.[9]

1.4.3 Composition

1.4.3.1 Action

Une action est le plus petit traitement qui puisse être exprimé en UML. Elle a une incidence sur l'état du système ou en extrait une information.

Les actions sont des étapes discrètes à partir desquelles se construise les comportements :

- une affectation de valeur à des attributs ;
- un accès à la valeur d'une propriété structurelle (attribut ou terminaison d'association) ;
- la création d'un nouvel objet ou lien ;
- un calcul arithmétique simple ;
- l'émission d'un signal ;
- la réception d'un signal .

Nous décrivons ci-dessous les types d'actions les plus courants prédéfinis dans la notation UML. (Voir le tableau 1.2).

Type d'action	Description
Action appeler (Call Operation).	invocation d'une opération sur un objet de manière synchrone ou asynchrone.
Action comportement (Call Behaviour).	variante de l'action call opération invoquant directement une activité plutôt qu'une opération.
Action envoyer (Send).	appel asynchrone bien adapté à l'envoi de signaux.
Action accepter événement (Accept Event).	attend la réception d'un signal (asynchrones).
Action accepter appel (Accept Call).	variante de l'action accept event pour les appels synchrones.
Action répondre (Reply).	transmet un message en réponse à un accept call.
Action créer (Create).	instancier un objet.
Action détruire (destroy).	détruire un objet.
Action lever exception (raise exception).	lever explicitement une exception.

TAB. 1.2 : Description des types d'actions

1.4.3.2 Activité

Une activité définit un comportement décrit par un séquencement organisé d'actions.

- Le flot d'exécution est modélisé par des nœuds reliés par des arcs (transitions).
- Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.

1.4.3.3 Groupe d'activités :

Un groupe d'activités est une activité regroupant des nœuds et des arcs.

- Les nœuds et les arcs peuvent appartenir à plus d'un groupe.
- Un diagramme d'activités est lui même un groupe d'activités.

1.4.3.4 Nœud d'activité :

Un nœud d'activité est une étape le long du flot d'une activité. Il existe trois familles de nœuds d'activités :

- les nœuds d'exécutions (executable node en anglais) ;
- les nœuds objets (object node en anglais) ;
- les nœuds de contrôle (control nodes en anglais).

La figure 1.3 représente types de nœuds d'activités :

1.4.3.4.1 Nœud exécutable :

C'est une classe abstraite pour les nœuds d'activités qui peuvent être exécutés. Il possède un gestionnaire d'exception qui peut capturer les exceptions levées par le nœud ou par l'un des nœuds imbriqués [10] .

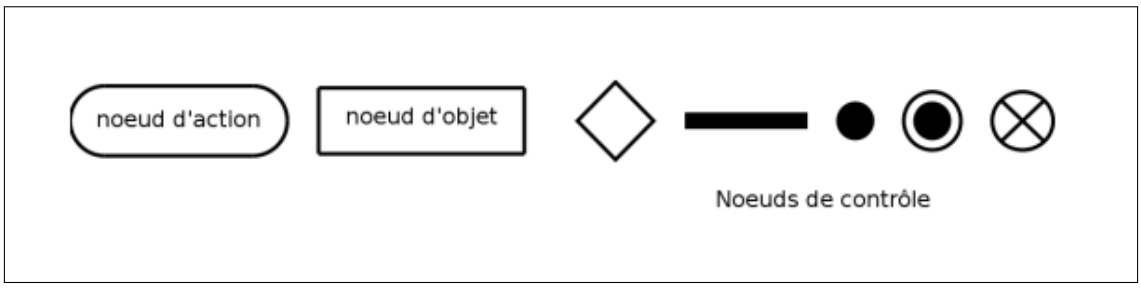


FIG. 1.3 : Notation nœuds d'activités.

1.4.3.4.2 Nœud d'objet :

C'est une méta-classe abstraite permettant de définir les flux d'objets dans les diagrammes d'activités. Il représente l'existence d'un objet généré par une action dans une activité et utilisé par d'autres actions. La figure 1.4 donne deux représentations équivalentes de flux d'objets entre deux actions. La première utilise des pins et l'autre utilise la notation d'un noeud objet [10] .

La figure 1.4 représente Flot d'objets :

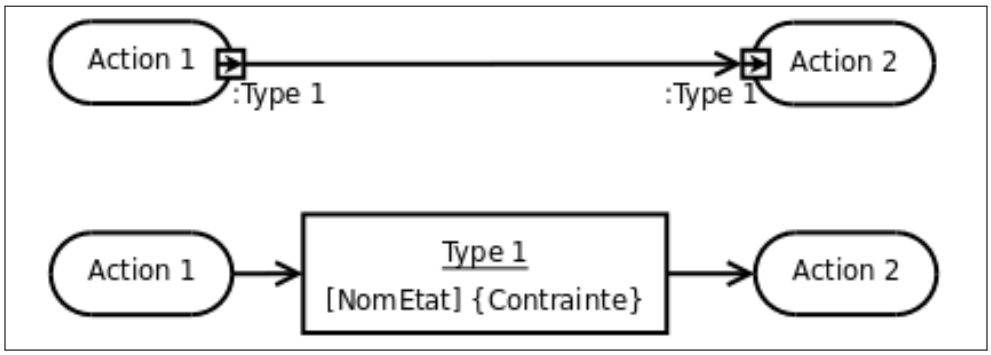


FIG. 1.4 : Deux notations possibles pour modéliser un flot de données.

1.4.3.4.3 Nœuds de contrôle :

C'est un nœud d'activité abstrait utilisé pour coordonner les flots entre les nœuds d'une activité. Il existe plusieurs types de nœuds de contrôle, la figure 1.5 présente ces types [10] :

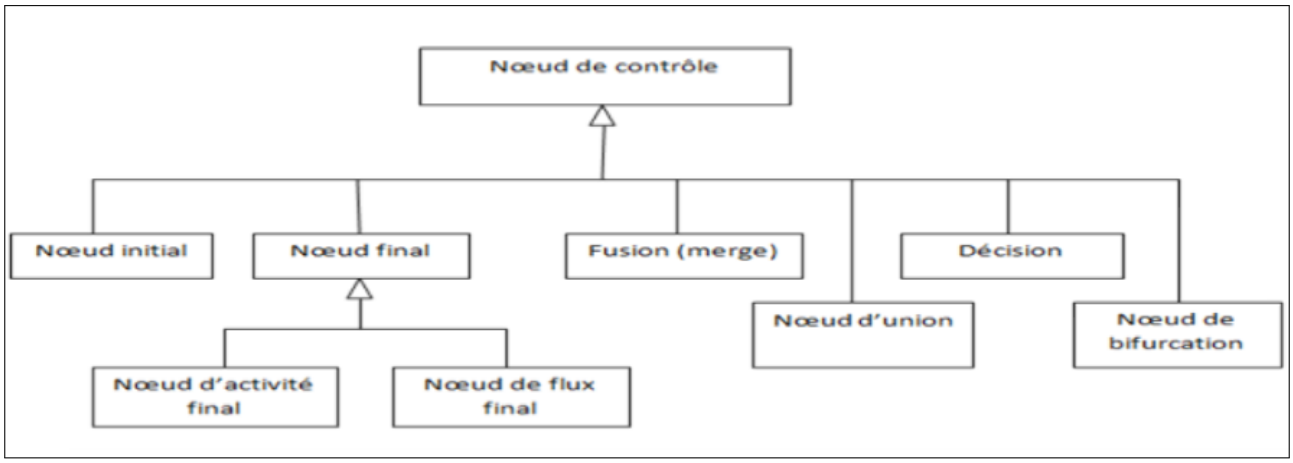
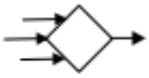
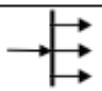
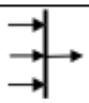


FIG. 1.5 : Les types des nœuds de contrôle.

Dans le tableau 1.3 on a la description des nœuds de contrôle :

Elément	Notation	Description
Nœud initial		C'est un nœud de contrôle à partir le duquel le flot débute. Il possède un arc sortant et pas d'arc entrant. Dans une activité, on a seulement un nœud initial.
Nœud d'activité final		Dans un nœud final d'activité, lorsque l'un de ses arcs entrants est activé, l'exécution de l'activité en cours s'achève, et tout nœud ou flux actif au sein de cette activité est abandonné.
Nœud de flux final		L'arrivé du flux d'exécution à un nœud final de flux, implique la terminaison du flux de ce dernier. Mais cette fin n'a aucun effet sur les autres flux actifs de l'activité.
Décision		Il permet de faire le choix entre plusieurs flux sortants. Ces flux sont sélectionnés en fonction de la condition de garde qui est associé à chaque arc sortant. Si aucun arc en sortie n'est franchissable, le modèle est mal formé, et l'utilisation d'une [else] garde est recommandée.
Fusion (Merge)		C'est un nœud de contrôle pour accepter un flot en sortie parmi plusieurs flots en entrées, l'utilité de ce nœud n'est pas pour synchroniser des flux concurrents.
Nœud de bifurcation (fork)		C'est le nœud de contrôle qui joue le rôle de séparation de flux d'entrée en plusieurs flots concurrents en sortie.
Nœud d'union (Join)		Il synchronise des flots multiples. Il possède plusieurs arcs entrants et un seul arc sortant. Ce dernier ne peut être activé que lorsque tous les arcs entrants sont activés.

TAB. 1.3 : Description des nœuds de contrôle

1.5 Conclusion

Dans ce chapitre nous avons présenté brièvement le langage de modélisation unifié UML, son évolution et ses diagrammes , Ensuite nous avons détaillé les diagrammes d'activités d'UML et leur composition puisqu'ils constituent la base de notre travail.

Les diagrammes d'activité ont des mécanismes pour éxprimer le séquencement, le choix et le parallélisme. Mais ils souffrent du manque des outils de vérification de comportement, qui sont disponible dans un langage comme langage Maude.

Chapitre 2

Le Système Maude

2.1 Introduction

La spécification des logiciels est une activité importante dans le développement de logiciels de qualité. Cette activité peut faite par des approches informelles ,semi-formelles ou formelles. les deux dernières sont les plus utilisés dans le cas des systèmes critiques. L'approche formelle nous permet à travers des spécifications formelles d'automatiser les activités de vérifications et de validation des systèmes. Plusieurs langages de spécification formelle ont été proposés dans la littérature comme : Object-Z, LOTOS etc. Maude est un langage basé sur une logique saine et complète dite la logique de réécriture. Maude supporte la spécification et la programmation des systèmes concurrents ,il a la puissance de description, la possibilité de simulation pour validation et la vérification formelle basée sur les techniques de model-checking.

2.2 La logique de réécriture

Les techniques de réécriture ont été développées depuis les années 1970 mais les premières publications sur la logique de réécriture c'étaient en 1990. La logique de réécriture est une logique solide et complète, elle donne un raisonnement mathématique au niveau de la sémantique. La logique de réécriture est appliquée dans plusieurs domaines comme : la spécification et la vérification des logiciels et des matériels, la sécurité et la déduction automatique. La logique de réécriture est en fait une amélioration de la logique équationnelle par l'ajout des règles de réécritures pour bien décrire la concurrence. Autrement dit, les règles de réécriture sont utilisées pour modéliser l'aspect dynamique du système c'est-à-dire qu'elles décrivent les transitions entre les états, alors que les équations sont utilisées pour réduire les termes à leur forme normale avant d'être réécrits en utilisant des règles de réécriture [11].

(1) **Systèmes de réécriture :** Un système de réécriture est un ensemble de règles de réécriture de la forme r $\rightarrow$ r'. Une telle règle s'applique à l'objet syntaxique t si celui ci contient une instance du membre gauche r, c'est à dire un sousobjet que l'on peut identifier à r. L'objet t se réécrit alors en un nouvel objet t', obtenu en remplaçant l'instance de r par l'instance du membre droit r' correspondante.
Notation : t $\rightarrow$ t' [13].

(2) **Réécriture de mots :**

- les objets sont des mots construits à partir d'un alphabet Σ = a, b, c, … ;
- une règle r $\rightarrow$ r' est constituée de deux mots r et r'. En général, on exige que le membre gauche r soit non vide.
Exemple : ab $\rightarrow$ caa .

  - le système formé des deux règles ab $\rightarrow$ a et ba $\rightarrow$ b n'est pas confluent, car on a aba $\rightarrow$ aa et aba $\rightarrow$ ab $\rightarrow$ a, et les mots aa et a sont en forme normale ;
  - le système formé des deux règles ab $\rightarrow \varepsilon$ et ba $\rightarrow \varepsilon$ (où ε est le mot vide) est confluent [13].

(3) **Réécriture de termes :**

- les objets sont des termes du premier ordre, c'est-à-dire des expressions construites à partir d'une signature constituée de constantes et de symboles d'opérations.
Exemple : 0,1,+,$*$ (deux constantes et deux opérations binaires, avec la convention habituelle de précédence du produit $*$ sur la somme +). On utilise aussi des variables x, y, z, … ;
- une règle r $\rightarrow$ r' est constituée de deux termes r et r'. En général, on exige que le membre gauche r ne soit pas une variable et que toutes les variables de r' apparaissent déjà dans r.
Exemple : x $*$ (y + z) $\rightarrow$ x $*$ y + x $*$ z.
Une instance du membre gauche est un sous-terme s obtenu en remplaçant les variables de r par des termes quelconques. L'instance correspondante du membre droit s'obtient en remplaçant les variables de r' par ces même termes.

Exemple : (x + 0) ∗ (x + 1) est une instance de x ∗ (y + z) et l'instance correspondante de x ∗ y + x ∗ z est
(x + 0) ∗ x + (x + 0) ∗ 1.
On a donc (x + 0) ∗ (x + 1) → (x + 0) ∗ x + (x + 0) ∗ 1,
et aussi
((x + 0) ∗ (x + 1)) + y → ((x + 0) ∗ x + (x + 0) ∗ 1) + y [13].

Confluence :

La nature non-déterministe de la réécriture fait qu'on peut appliquer plusieurs règles au même objet, obtenant ainsi plusieurs résultats différents.
Pour un système de réécriture donné, la propriété de confluence s'énonce ainsi : si t →* u et t →* v, alors il existe w tel que
u →* w et v →* w. Elle équivaut à la propriété de Church-Rosser [13].

2.3 Théorie de réécriture

La logique de réécriture est représentée par une théorie de réécriture T= (Σ, E, L,R), tel que [12] :

- La structure statique du système est décrite par la signature (Σ, E) qui représente les états d'un système.
- La structure dynamique est décrite par les règles de réécriture de la forme :
  rl [l] : t → t', qui représentent les transitions. Dans la logique de réécriture, les formules logiques sont dites règles de réécriture.
  Une règlede réécriture peut aussi être conditionnelle de la forme :
  crl [l] : t → t' if C , Ce qui indique que le terme t devient (se transforme en) t' si une certaine condition C'est vérifiée. Le terme t représente un état partiel d'un état global du système décrit.

2.3.1 Théorie de réécriture étiquetée

Une théorie de réécriture ℜ est un quadruplet ℜ = (Σ , E, L, R) tel que :

- Σ est un ensemble de symboles de fonctions, et de sortes.
- E un ensemble de Σ-équations (l'ensemble des les Σ-équations entre termes).
- L est un ensemble d'étiquettes.
- R est un ensemble de paires $\Re \subseteq L \times (T_{\Sigma,E}(X))^2$ tel que le premier composant est une étiquette,et le second est une paire de classes d'équivalences de termes modulo les équations E avec $X = \{x_1, ...x_n...\}$ un ensemble infini et dénombrable de variables[12].

2.3.2 Les systèmes de réécriture conditionnels

Un système de réécriture conditionnel naturel à des règles de réécriture de la forme
$R([t],[t'],C_1,...,C_k)$ la notation suivante est → utilisé : $R : [t] \to [t']$ if $C_1 \wedge ... \wedge C_k$. Où une règle R exprime que la classe d'équivalence contenant le terme t peut se réécrire en la classe d'équivalence contenant le terme t' si la condition de la règle $C_1 \wedge ... \wedge C_k$ Est vérifiée. Cette dernière est appelée condition de la règle et peut être abrégée pa r la lettre C,et la règle de réécriture,dans ce cas,est dite conditionnelle. La partie conditionnelle d'une règle peut être vide,dans ce cas les règles sont appelées règles de réécriture inconditionnelles et son notés par $R : [t] \to [t']$ [12].

2.4 Règles de déduction

Dans un système, la séquence des transitions exécutées se commence à partir d'un état initial. Les règles formalisant l'opération de réécriture des termes sont appelées des règles de déductions.
Les règles de déduction sont :

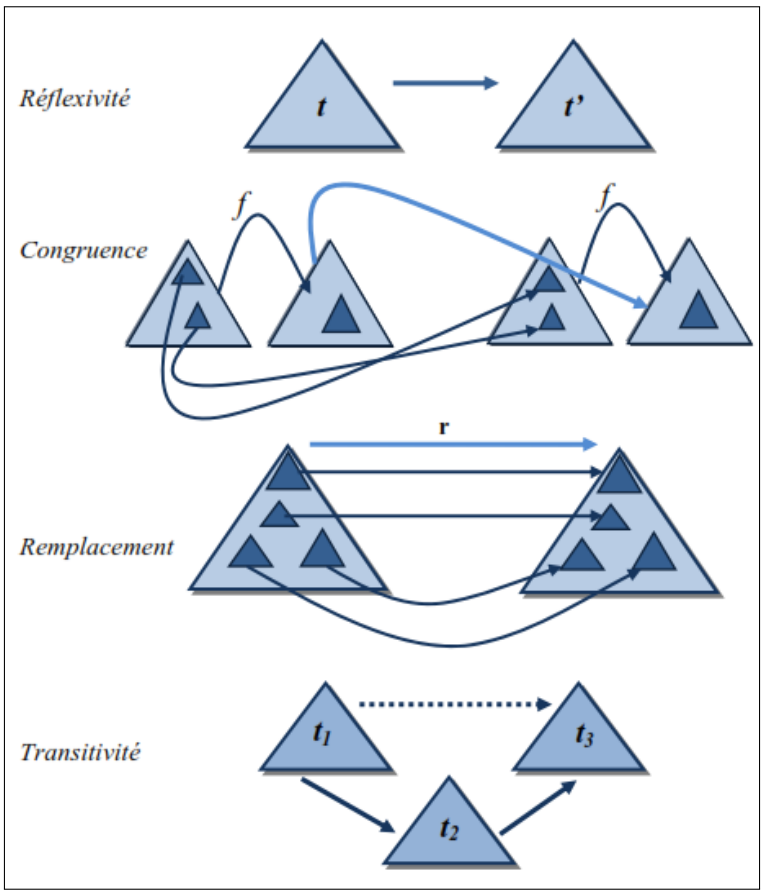


FIG. 2.1 : Représentation graphique des règles de déduction.

- **La réflexivité :**
  Pour chaque terme $[t] \in T_{\Sigma,E}(X)$; $\overline{[t] \rightarrow [t]}$ est l'ensemble des Σ termes avec variables construits sur la signature Σ et les équations[12].

- **La congruence :**
  est une forme générale de « parallélisme latéral », de sorte que chaque opérateur f peut être vu comme un constructeur d'états parallèles, permettant à ses arguments non figés d'évoluer en parallèle[12].
  Pour chaque fonction $f \in \Sigma n, n \in N$.

$$\frac{[t_1]\rightarrow[t_1^{'}].....[t]\rightarrow[t_n^{'}]}{[f(t_1.....t_n)]\rightarrow[f(t_1^{'}.....t_n^{'})]}$$

- **Le remplacement :** prend en charge une forme différente de parallélisme.
  Pour chaque règle r : $[t(x_1...X_n)] \rightarrow [t^{'}(x_1...X_n)]$ dans R :

$$\frac{[w_1]\rightarrow[w_1^{'}].....[w]\rightarrow[w_n^{'}]}{[t(\overline{w}/\overline{X})]\rightarrow[t^{'}(\overline{w'}/\overline{X})]}$$

  Sachant que $t(\overline{w}/\overline{X})$ dénote la substitution simultanée de x_i par w_i dans avec $\overline{x}$ représentant x_1 ... ... x_n[12].
- **La transitivité :**
  elle nous permet de construire des calculs simultanés plus longs en les composants séquentiellement[12].

$$\frac{[t_1]\rightarrow[t_2][t_2]\rightarrow[t_3]}{[t_1]\rightarrow[t_3]}$$

2.5 Maude : Langage Basé sur la Logique de Réécriture

Il y a beaucoup de langages qui sont basés sur la logique de la réécriture dans la modélisation du système, parmi lesquels (CafeOBJ, ELAN et Maude ... etc) :

Dans le cadre de notre étude, nous utilisons le langage Maude qui est basé sur la logique de réécriture pour produire un code exécutable pour un système.

2.5.1 Le système Maude

Maude est un langage déclaratif et formel de haute performance, basé sur la logique de réécriture, il permet la modélisation des systèmes et aussi des actions au sein de ces systèmes.
Grâce à la puissance de sa logique, Maude offre une collection puissante d'outils formels supportant différentes formes de raisonnement logique pour vérifier des propriétés de programme comprenant [24].

2.5.1.1 Les caractéristiques du système Maude

Le langage Maude, basé sur la logique de réécriture, où les programmes sont des théories, et les règles de déduction logique de réécriture correspond exactement au calcul concurrent, a été développé en fonction de trois grands objectifs : la simplicité, l'expressivité et la performance.

- **La simplicité** : Maude est un langage déclaratif qui offre des notions de programmation simples et faciles à comprendre. En effet, les expressions de base de ce langage sont soit des équations soit des règles de réécriture.
- **L'expressivité** : Le langage Maude permet d'exprimer naturellement une vaste gamme d'applications, en commençant par les programmes déterministes arrivant aux programmes hautement concurrentiels. En outre, la sémantique rigoureuse de Maude permet non seulement d'exprimer des applications, mais aussi des formalismes entiers (d'autres langages, d'autres logiques). De ce fait, Maude est vue comme étant un « métalangage » avec lequel il est facile de développer un langage spécifique.
- **La puissance et la performance** : En plus d'être un langage de spécification formelle, Maude est un véritable langage de programmation compétitif. Il permet l'exécution de millions de règles de réécriture par seconde [12].

2.5.1.2 Les modules d'une spécification Maude

Le développement Maude se fait par l'écriture de divers modules décrivant le système que l'on écrit. Ainsi, l'unité de base pour le développement Maude est le module.

Pour une bonne description modulaire, trois types de modules sont définis dans Maude. Les modules fonctionnels permettent de définir les types de données. Les modules systèmes permettent de définir le comportement dynamique d'un système. Enfin, les modules orientés objet qui peuvent, en fait, être réduits à des modules systèmes offrent explicitement les avantages du paradigme objet.

2.5.1.2.1 Les modules fonctionnels :

Les modules fonctionnels sont des théories fonctionnelles [19] Que nous avons présentées précédemment. Ces modules définissent les types de données et les opérations qui sont utilisés par les équations. L'algèbre initiale sous jacente est un modèle mathématique dénotationel pour les sortes et les opérations. Les éléments de cette algèbre sont des classes d'équivalence des termes sans variables (ground terms [15]) modulo les équations. Si deux termes sans variables sont égaux par une équation, on dit qu'ils appartiennent à la même classe d'équivalence [16]. Les équations sont utilisées comme des règles de réductions. À la fin du calcul, chaque règles est évaluée à sa forme réduite dite représentation canonique [16]. La représentation canonique est unique et elle représente tous les termes de la même classe d'équivalence.
Les équations dans un module fonctionnel sont orientées, elles sont utilisées de gauche à droite, le

résultat de réduction est unique comme nous avons dit quelques soit l'ordre dans lequel les équations sont appliquées.
Les modules fonctionnels supportent aussi les axiomes d'appartenance (membership axioms) [19] ces axiomes précisent l'appartenance d'un terme à un type. Les axiomes peuvent être conditionnels ou inconditionnels. En cas des axiomes conditionnelles, les conditions sont des jonctions des équations et des testes d'appartenance inconditionnels [14].
En Maude, une spécification équationnelle est un module fonctionnel qui est représenté par la syntaxe suivante [18].

fmod MODULE-NAME *is*
BODY
endfm

Où MODULE-NAME est le nom de module introduit, et BODY est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance et des commentaires. Les commentaires commencent par $***$ ou —- et se termine par la fin de la ligne courante ou elles commencent par $***$ (ou —(et se termine par l'occurrence de ")" [17].
pour montrer l'utilisation de module fonctionnel, où donne l'exemple dû le réseau de Petri (RdP) ordinaire suivant :

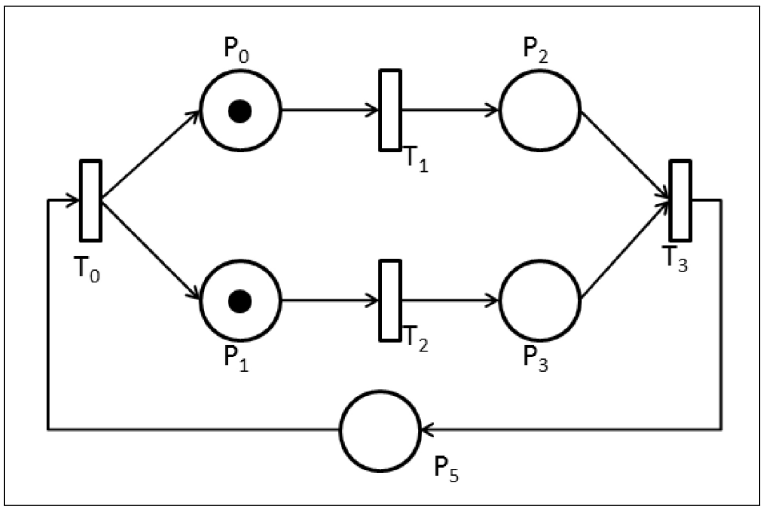


Fig. 2.2 : Exemple de réseaux de Petri.

La spécification correspondant est comme suit :

fmod PETRI-NET-SIGNATURE is
sorts Place Marking .
subsorts Place < Marking .
ops P0 P1 P2 P3 P5 : -> Place .
op null : -> Marking .
op _ _ : Marking Marking -> Marking [ctor assoc comm id : null] .
op initial : -> Marking.
eq initial = P0 P1 .
endfm

Tab. 2.1 : Module fonctionnel.

2.5.1.2.2 Les modules systèmes :

Les modules systèmes sont des théories de réécriture [19]. Ils permettent de spécifier le comportement d'un système concurrent. Les modules systèmes ajoutent à la définition des modules fonctionnels les règles de réécriture qui peuvent être conditionnels et inconditionnels. L'introduction des règles de réécriture permet d'exprimer la concurrence dans les systèmes.
Un module système décrive une théorie de réécriture qui inclue des sortes, des opérations, des variables, des équations, des axiomes d'appartenances (conditionnelles et inconditionnelles) et des règles de réécriture conditionnelles et inconditionnelles.
Une règle de réécriture s'exécute quand sa partie gauche correspond (match) une portion dans l'état global du système et avec la satisfaction de la condition en cas d'une règle conditionnelle [16].
En Maude, une spécification est un module système qui est représenté par la syntaxe suivante [82] :

mod MODULE-NAME *is*
BODY
endfm.

Où MODULE-NAME est le nom du module introduit, et BODY est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance, et des règles de réécriture [14].
Reprenons l'exemple des réseaux de Petri précédent, la spécification correspondante est comme suit :

```
mod PETRI-NET is
protecting PETRI-NET-SIGNATURE .
rl [T0] : P5 => P0 P1 .
rl [T1] : P0 => P2 .
rl [T2] : P1 => P3 .
rl [T3] : P2 P3 => P5 .
endm
```

Tab. 2.2 : Module système.

2.5.1.2.3 Les modules orientés-objet :

Dans le langage Maude, les systèmes basés sur le paradigme orienté objet peuvent être définis à travers des modules orientés objet. Les modules orientés objet sont supportés par le système FullMaude [20]. Les modules orientés objet sont déclarés avec la syntaxe suivante :

omod < nom-module > is < déclarations et expressions > *endom* .

Les modules orientés objet offrent une syntaxe appropriée qui permet de déclarer les concepts de base du paradigme orienté objet :

- Oid pour identifier les objets,
- Cid pour identifier les classes,
- Objects pour définir les objets
- Msg pour déclarer les messages.

Ils supportent la spécification et la manipulation des objets, des messages, des classes et de l'héritage. Un système orienté objet concurrent dans ce cas est modélisé par un multi-ensemble d'objets et de messages juxtaposés, où les interactions concurrentes entre les objets sont régies par des règles de réécriture. Dans les modules orientés objet, un objet est représenté par un terme :

$$< \text{O} : \text{C} \mid \text{a1} : \text{v1} , \ldots , \text{an} : \text{vn} >.$$

Où **O** est l'identificateur de l'objet, **C** est l'identificateur de sa classe, les **ai** sont les noms des attributs de l'objet et enfin les **vi** correspondent aux valeurs des attributs. Les messages dans les modules orientés objet n'ont pas de forme syntaxique prédéfinie, leur structure est définie par l'utilisateur. La seule contrainte est que le premier argument doit être l'identifiant de l'objet destination. La déclaration des classes suit la syntaxe :

$$class \text{ C} \mid \text{a1} : \text{s1} , \ldots , \text{an} : \text{sn} .$$

Où **C** est le nom de la classe et **si** est la sorte de l'attribut **ai**. Il est aussi possible de déclarer des sous classes et bénéficier ainsi de la notion d'héritage. Les messages sont déclarés en utilisant le mot clé msg. La forme générale d'une règle de réécriture dans la syntaxe orientée objet de Maude est :

crl [r] : M_1 … M_n $< O_1 : F_1 \mid a_1 >$ … $< O_m : F_m \mid a_m > \Rightarrow < O_{i1} : F'_{i1} \mid a'_{i1} >$ … $< O_{ik} : F'_{ik} \mid a'_{ik} >$ M'_1 … M'_p if Cond .

Où r est l'étiquette de la règle, M_s, s ∈ 1 ... n, et M'_u, u ∈ 1 ... p sont des messages, O_i, i ∈ 1 ... m, et O_{il}, l ∈ 1 ... k, sont des objets, et Cond est la condition de la règle. Si la règle est non conditionnelle, nous remplaçons le mot clé **crl** par **rl** et nous enlevons la clause if Cond.
Notons que le système Full-Maude offre un support additionnel pour la programmation orientée objet avec les notions de classes, sous-classes et une syntaxe plus conviviale des règles de réécriture. Ainsi, il permet au système Maude de supporter la modélisation orientés objet en fournissant le module prédéfini CONFIGURATION. Dans ce module, les sortes représentant les concepts essentiels des objets, classes, messages et configurations, sont déclarés.

2.6 Conclusion

Nous avons présenté, dans ce chapitre, les notions relatives à la compréhension des concepts de base de la logique de réécriture. Dans un premier temps, nous avons présenté l'aspect théorique de la logique de réécriture, qui est basée sur deux concepts théoriques très connus depuis bien longtemps : les systèmes de réécriture et les spécifications équationnelles. Avec ces concepts, la logique de réécriture possède un pouvoir d'abstraction et de description de tout type de système. Ensuite, nous avons introduit le système Maude qui un langage caractérisé par la simplicité, la performance et l'expressivité. Maude est un candidat idéal pour la spécification de système car il est basé sur une logique très mathématique et dispose d'une forte sémantique et des techniques d'analyses formelles.

Chapitre 3

Transformation de diagramme d'activité vers Maude

3.1 Introduction

Dans ce chapitre, Nous présenterons quelques travaux connexes au domaine des transformations du diagramme UML, Nous proposerons également une approche qui transforme de diagramme d'activité en Maude sur la base de quelques règles simples de transformations d'une classe UML vers Maude.

3.2 Travaux connexes

3.2.1 Introduction

Uml est un langage de représentation destiné en particulier à la modélisation objet. Il utilise pour la conception des applications informatiques, uml est un langage semi-formel le point faible d'uml l'absence de bases formelles pour les vérifications formelles , pour cela il y a des travaux qui ont transféré les modèles UML aux plusieurs langages formelles comme : FoCaLiZe, Alloy, B et Maud.

Dans ce chapitre, nous allons se concentrer sur les travaux liés avec notre mémoire, qui transforme des diagrammes UML aux langages formels.

3.2.2 Transformation vers B

la méthode B est un langage de programmation formelle basée sur la théorie des ensembles,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.
Un travail a été fait par Clin Snook et Michael Butler ce le titre de "Formal modelling and design aided by UML"publié dans le journal "journal ACM Transactions on Software Engineering and Methodology" au 2006, ils proposent des règles de la transformation des diagrammes UML(diagramme de classe et d'état)vers la méthode B [21].

3.2.3 Transformation vers FoCaLiZe

Dans leur mémoire[22], les étudiants proposent une approche de la transformation du diagramme de classes vers Focalize. Dans ce mémoire les étudiants réalisent un outil de transformation automatique de diagramme de classes UML en FoCaLiZe. L'implémentation proposée supporte la plupart des fonctionnalités d'UML telles que l'héritage, l'héritage multiple, la dépendance, le template UML et le template binding. Le but de leur transformation est de tirer une partie des outils de vérification de FoCaLiZe pour vérifier les propriétés d'un modèle UML.

3.2.4 Transforme vers alloy

Alloy est un langage open source et un analyseur pour la modélisation de logiciels. il est utilisé dans une large gamme d'applications, Alloy fournit un outil de modélisation structurelle simple basé sur la logique du premier ordre.

Il y a un travail de Ana CR Paiva, Alcino Cunha et Daniel Riesco Dans la transformation de l'aspect structurel d'UML vers Alloy, il y a beaucoup de travaux dans ce domaine , nous trouvons une étude de l'université de Birmingham sous le titre de "On Challenges of Model Transformation from UML to Alloy" [23].

3.2.5 Transformation vers les réseaux de Petri

Un travail a été réalisée on 2009 par deux étudiants (Université de Constantine) sous le titre de : "De M-UML vers les réseaux de Petri« Nested Nets » : Une approche basée transformation de graphes", dans ce travail ils présentèrent une transformation de diagramme d'état transition mobile de M-UML vers les RDPs Nested Nets, ils se basant sur le paradigme de la transformation de graphe,

l'approche consiste à proposer une grammaire de graphe et des règles pour la transformation entre deux formalismes différents. La transformation est implémentée à l'aide de l'outil AToM3 [24].

3.2.6 Transformation vers Maude

Le travail [26] effectue une transformation du diagramme d'activité vers le système Maude. Ce dernier est très proche du nôtre sauf qu'il est basé sur une approche de transformation de graphe qui aplatit la transformation, c'est-à-dire que les nœuds ne peuvent pas avoir suffisamment de détails (propriétés, variables) comme leurs cas réels dans un diagramme d'activité. En d'autres termes, dans leur travail, ils effectuent une transformation des nœuds comme s'il s'agissait de boîtes noires.

3.2.7 Synthèse et motivation

La majorité des travaux précédents (B, FoCaLiZe, Alloy et les réseaux de Petri) ont prouvé leur capacité à supporter plusieurs fonctionnalités UML, mais quelques uns eux mêmes notent qu'ils ignorent certains fonctionnalités et d'autres ont basés sur l'utilisation des prouveurs de théorèmes pour la vérification ce qui nécessite un expert des concepts mathématiques et parfois le prouveur lui même nécessite une intervention de la part du développeur ce qui rends la tâche de vérification n'est pas completement automatique.

Dans notre mémoire, on va proposer un travail qui réalise une transformation de diagramme d'activité vers Maude du fait qu'il possède un prouveur de théorème et un vérificateur de modèle (model-checker). Cette caractéristique offre plus de choix au développeur pour le type d'outil de vérification à utiliser.

3.3 Approche de Transformation Proposée

Dans le cadre de proposer une approche de transformation, on a remarqué la facilité de spécification des composants (noeuds) du diagramme d'activité à travers l'approche objet sous Maude. Dans ce qui suit, on donne une description de l'approche proposée.

3.3.1 Description générale des règles de Transformation

Nous avons remarqué q'une activité (diagramme d'activité) se compose de deux groupes de nœuds selon notre point de vue pour la transformation vers Maude :

Le premier groupe :

Ce groupe contient trois nœuds qui sont : nœud Action, nœud Initial et nœud Final (fin d'activité/flot). Le nœud « Action » représente une incidence sur l'état du système ou en extrait une information. L'exécution d'une action représente une transformation ou un calcul quelconque dans le système modélisé. Les deux autres nœuds (initial et final) peuvent aussi contenir des propriétés caractérisant l'état du système ou d'un de ses objets.

Par conséquent, on voit clairement que ce groupe de nœuds doit contenir quelques propriétés similaires à l'objet comme : Nom (libellé), type (classe), et quelques attributs qui seront utilisés pour représenter un état ou une valeur. Afin de simplifier le processus de transformation, on va utiliser des attributs abstraits où le développeur peut les modifier selon le cas de son système.

Cette similarité entre les noeuds du premier groupe du diagramme d'activité et l'approche objet en Maude peut être résumée comme suit :

Noeud	Object
Nom	Object Id
Propriété/Variable	Attribut
Type	Classe

Le deuxième groupe :
Ce groupe contient la transition ainsi que le l'ensemble de quatre nœuds suivant : Décision, Fusion, Union, Bifurcation. Les noeuds de ce groupe reflète la dynamicité et/ou le changement dans le diagramme d'activité. Autrement dit, un nœud de cet groupe décrit le déclenchement et/ou le contrôle des transformations dans le système avec leurs conditions. Par conséquent, les règles de réécriture seront le meilleur moyen pour les présenter en Maude.

3.3.2 Description détaillée des règles de Transformation

Dans cette section, nous allons présenter les détails de spécification des noeuds du diagramme d'activité. En commençant par un rappel sur la spécification des objets et des classe sous Maude. En suite, nous donnons le code Maude correspondant à chaque noeud du diagramme d'activité.

3.3.2.1 Spécification d'une classe et d'un objet

La spécification d'une classe en Maude est donnée comme suit :

class C | a1 : s1 , . . . , an : sn .

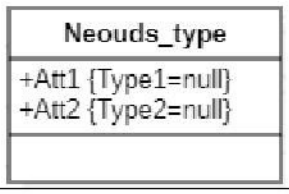
Où C est le nom de la classe , Si est la sorte de l'attribut et a_i est le nom de l'attribut.

Un objet dans un état donné est spécifié en Maude comme suit :

< O : C | a1 : v1 , . . . , an : vn >.

Où O est le nom de l'objet ou l'identificateur, C est sa classe, a_i sont les attributs de l'objet et les v_i sont les valeurs correspondantes.

Le tableau 3.1 montre la spécification d'une classe avec ses attributs en Maude :

Classe UML	Code Maude
	*class* Nœuds_type | $Att_1 : Type_1$, $Att_2 : Type_2$.

Tab. 3.1 : Transformation de classe UML et objet.

3.3.2.2 Transformation des noeuds du diagramme d'activité

Comme nous avons cité précédemment, nous allons présenter implicitement la transformation pour les deux ensemble de noeuds du diagramme d'activité sans faire la décomposition.

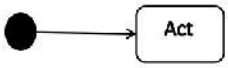
3.3.2.2.1 Transformation d'une Action :
Une action est le plus petit traitement qui puisse être exprimé en UML. Elle a une incidence sur l'état du système ou en extrait une information. Nous considérons l'action dans cette approche comme un objet qui a un nom et des attributs. Une action doit apprtient à une classe des actions. Par conséquent, Une action (Act) est représentée en Maude comme suit :

Action	Code Maude
	$< \text{Act} : \text{Action} \mid Att_1 : Val_1, ..., Att_n : Val_n >.$

TAB. 3.2 : Transformation d'une Action.

3.3.2.2.2 Transformation du Nœud initial :

Un nœud initial est un nœud de contrôle à partir duquel le flot débute lorsque l'activité enveloppante est invoquée, Un nœud initial possède un arc sortant et pas d'arc entrant. Dans une activité, on a seulement un nœud initial.
Dans le tableau suivant, nous illustrons la spécification sous Maude du noeud initial lié à une action :

Nœud initial	Code Maude
	rl [Initial] : $< \text{Initial} : \text{InitialNode} \mid Att_1 : Val_1, ..., Att_n : Val_n >$ $\Rightarrow < \text{Act} : \text{Action} \mid Att_1 : Val_1, ..., Att_n : Val_n >.$

TAB. 3.3 : Transformation du nœud initial.

3.3.2.2.3 Transformation du Nœud final :

Le noeud final d'un diagramme d'activité dénote la fin de tous les flux de contrôle dans l'activité et/ou le diagramme complet, il est représenté comme un cercle avec un point (ou un symbole de multiplication) à l'intérieur.
Dans le tableau suivant, un exemple de la transformation du noeud final vers Maude est présenté :

Nœud Final	Code Maude
	rl [FinalAction] : $< \text{Act} : \text{Action} \mid Att_1 : Val_1, ..., Att_n : Val_n > \Rightarrow$ $< \text{FinalAction} : \text{FinalActivityNode} \mid Att_1 : Val_1, ..., Att_n : Val_n >.$

TAB. 3.4 : Transformation du nœud Final.

3.3.2.2.4 Transformation d'une Transition :

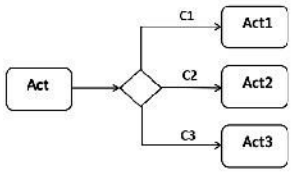
Ces nœuds matérialisent le passage d une activité vers une autre. ils sont représentés par des flèches en traits pleins qui connectent les activités entre elles .
Dans le tableau suivant, nous donnons la spécification corespondante d'une Transition (ControlFlow) sous Maude :

Nœud Transition	Code Maude
	rl [Transition] : $< \text{Act} : \text{Action} \mid Att_1 : Val_1, ..., Att_n : Val_n > \Rightarrow <$ $\text{Act} : \text{Action} \mid Att_1 : Val_1, ..., Att_n : Val_n >.$

TAB. 3.5 : Transformation d'une Transition.

3.3.2.2.5 Transformation du Nœud de Décision :

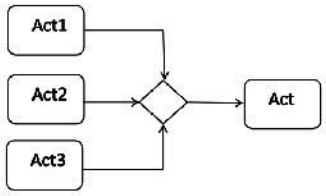
Ce nœud représente un point d'une activité où un bord entrant unique débouche sur plusieurs bords sortants. Le tableau suivant montre la spécification du noeud de Décision sous Maude :

Nœud de Décision	Code Maude
	crl [DecisionC1] : $<$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $\Rightarrow <$ Act1 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ if (C1). crl [DecisionC2] : $<$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $\Rightarrow <$ Act2 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ if (C2). crl [DecisionC3] : $<$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n > \Rightarrow$ $<$ Act3 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ if (C3).

TAB. 3.6 : Transformation du nœud de Décision.

3.3.2.2.6 Transformation du Nœud de Fusion :

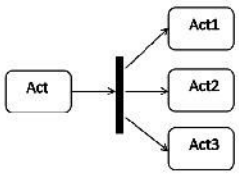
Ce nœud est un nœud de contrôle qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents mais pour n'accepter un flot parmi plusieurs. Le tableau suivant montre la transformation du noeud de Fusion vers Maude :

Nœud de Fusion	Code Maude
	rl [Merge1-1] : $<$ Act1 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $\Rightarrow <$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$. rl [Merge1-2] : $<$ Act2 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $\Rightarrow <$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$. rl [Merge1-3] : $<$ Act3 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $\Rightarrow <$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$.

TAB. 3.7 : Transformation du nœud de Fusion.

3.3.2.2.7 Transformation du Nœud de Bifurcation :

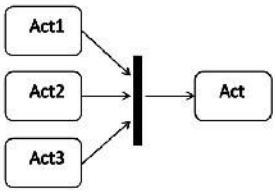
Un nœud de bifurcation, également appelé nœud de débranchement est un nœud de contrôle qui sépare un flot en plusieurs flots concurrents. Un tel nœud possède donc un arc entrant et plusieurs arcs sortants. Le tableau suivant montre la transformation du noeud de Fusion vers Maude :

Nœud de Bifurcation	Code Maude
	rl [Fork] : $<$ Act : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n > \Rightarrow <$ Act1 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $<$ Act2 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$ $<$ Act3 : Action $\mid$ $Att_1 : Val_1, ..., Att_n : Val_n >$.

TAB. 3.8 : Transformation du Nœud de Bifurcation

3.3.2.2.8 Transformation du Nœud d'union :

Un nœud d'union, également appelé nœud de jointure est un nœud de contrôle qui synchronise des flots multiples. Un tel nœud possède donc plusieurs arcs entrants et un seul arc sortant. Lorsque tous les arcs entrants sont activés, l'arc sortant l'est également. Le tableau suivant montre la transformation du noeud d'union vers Maude :

Nœud d'union	Code Maude
	rl [Join] : < Act1 : Action | $Att_1 : Val_1, ..., Att_n : Val_n$ > < Act2 : Action | $Att_1 : Val_1, ..., Att_n : Val_n$ > < Act3 : Action | $Att_1 : Val_1, ..., Att_n : Val_n$ > ⇒ < Act : Action | $Att_1 : Val_1, ..., Att_n : Val_n$ >.

TAB. 3.9 : Transformation du nœud d'union

3.3.3 Un exemple complet

Pour évaluer l'utilité pratique de l'approche proposée, nous considérons un exemple simple d'application de traitement de commandes. Dans ce diagramme, la première action est de recevoir la commande demandée. Une fois la commande acceptée et toutes les informations requises renseignées, le paiement est accepté et la commande est expédiée. Nous notons que cet exemple permet l'expédition de la commande avant l'envoi de la facture ou la confirmation du paiement. La figure (3.1) présente le diagramme d'activité UML de l'ordre de traitement créé avec l'outil StarUML.

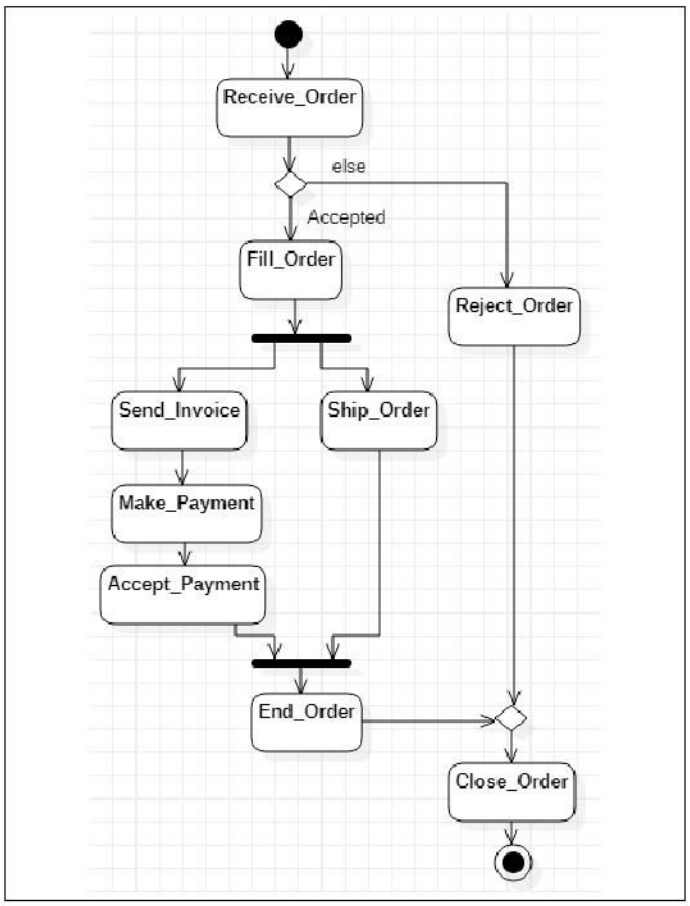


FIG. 3.1 : Diagramme d'activité "Traitement des commandes"

Ensuite, et par l'application des règles de transformation citées précédemment sur ce diagramme, nous obtenons la spécification Maude correspondante comme suit (Figure 3.2) :

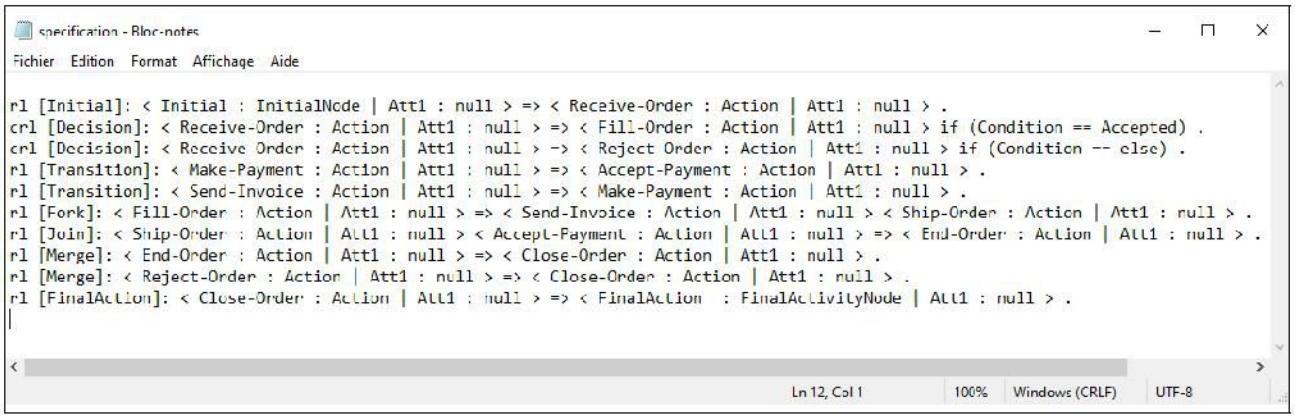


FIG. 3.2 : Spécification Maude du diagramme d'activité "Traitement des commandes"

3.3.4 Description générale de l'approche de transformation

Notre approche de transformation automatique des diagrammes d'activités vers sa spécification correspondante sous Maude se compose de quatre étapes importante comme suit :

1. **Création du modèle :** Cette étape consiste à préparer le diagramme d'activité en utilisant un des outils UML. Dans notre cas, nous avons choisis StarUML.
2. **Exportation du modèle en XMI :** Dans cette étape, on doit exporter le modèle préparé sous format XMI. Les représentations XMI sont exploitées pour migrer des données d'un modèle à un autre. Pratiquement, le choix de l'outil UML ne nécessite aucun changement dans le code de l'étape de génération de la spécification du fait de la standardisation des représentations XML.
3. **Traitement du fichier XMI :** Dans cette étape, nous avons utilisé une bibliothèque du langage Python qui nous permet de manipuler les fichiers XMI. Le traitement de cette étape fait une partie intègrée dans notre code, c-à-d, nous avons pas utilisé un fichier de commande externe pour parser/détecter et extraire les composants du diagramme original.
4. **Génération de la spécification Maude :** Cette étape est le plus importante dans le processus de transformation du diagramme d'activité vers la spécification Maude. Elle est basée sur le traitement effectué dans l'étape précédente. En plus, la génération de la spécification est basée sur une Template prédéfinit et un ensemble de règles de transformation intégrées dans notre code.

Enfin, ces étapes peuvent être résumées graphiquement comme le montre le schéma de la figure suivante.

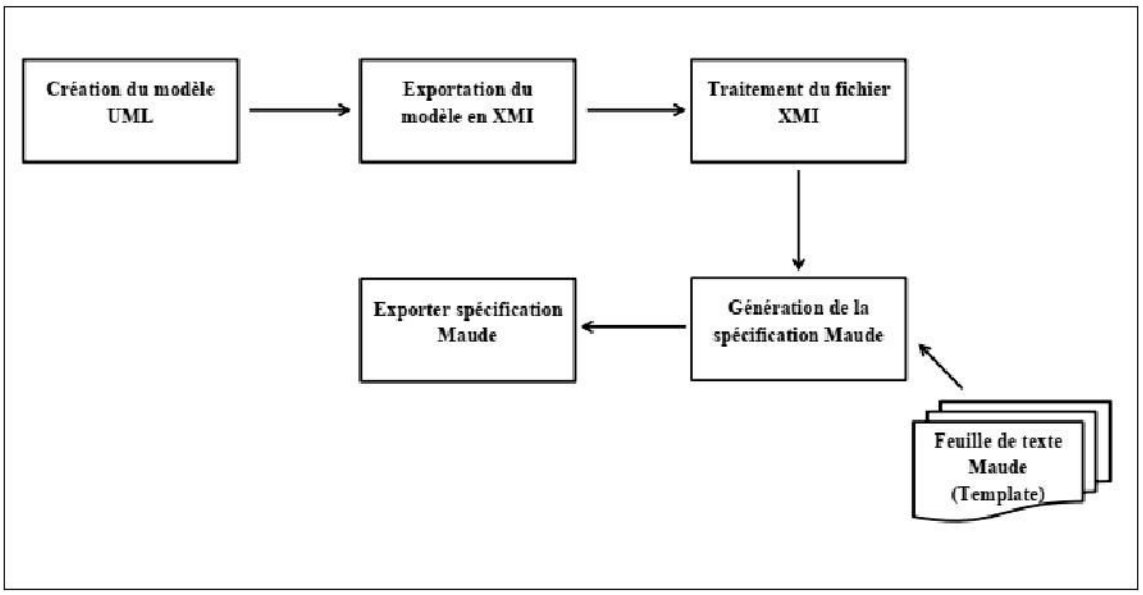


FIG. 3.3 : Schéma général de l'approche de transformation.

3.4 Conclusion

Nous avons présenté dans ce chapitre les travaux réalisés dans le contexte de transformation de diagrammes uml vers autres langages pour des fins de vérification comme FoCaLiZe, B, alloy, réseaux de Petri et Maude. Ensuite, nous avons présenté notre approche de transformation des nœuds de diagramme d'activité vers le langage Maude avec les détails nécessaires.

Chapitre 4

Conception et aspect d'implémentation

4.1 Introduction

Les diagrammes d'activité permettent la modélisation des systèmes workflows, des modèles orientés service et des processus métiers. Ils sont adaptés à la modélisation du cheminement de flux de données et de flux de contrôle. Ils incluent des mécanismes pour exprimer le parallélisme, le séquencement, le choix et les événements. Cependant, ils souffrent de l'inconvénient du manque des outils de vérification du comportement. Les diagrammes d'activité doivent être fournies avec une sémantique formelle, afin de pouvoir vérifier n'importe quel aspect du comportement. Pour pallier ce problème le langage Maude peut être utilisé, en définissant une transformation des diagrammes d'activité vers Maude.

Le but de ce chapitre est de présenter d'un outil de transformation automatique des diagrammes d'activité d'UML vers Maude .

Le chapitre est organisé en trois grandes parties, la première partie est consacré à l'explication L'environnement de travail utilisé (StarUml et Python), la deuxième partie contient sur d'explication détaillée la Processus de génération de spécifications Maude. Ensuite, la troisième partie qui contient d'exemple d'un outil de transformation (les différentes étapes de transformation).

4.2 L'environnement et les outils utilisés

4.2.1 StarUml

StarUML est un logiciel de modélisation UML disponible en Open Source. Via cette plateforme, vous serez en mesure de concevoir une dizaine de types de diagrammes. Il vous sera notamment possible de créer de classes, d'objets, d'activités ou bien de séquences compatibles avec le standard UML 2.0. StarUML est écrit technologies Web (HTML5, CSS, JavaScript). Il est facile d'accès. ce logiciel constitue une excellente option pour une familiarisation à la modélisation. (Editeur : Plastic Software – http ://staruml.io)

La figure 4.1 représente l'interface de StarUml.

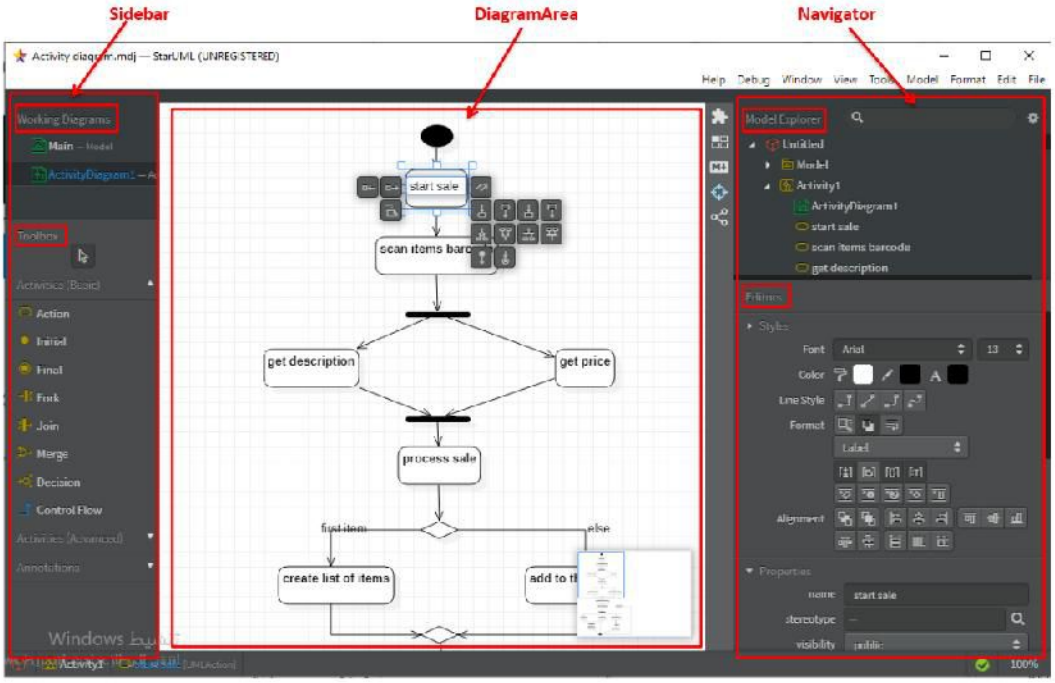


FIG. 4.1 : L'interface de StarUml.

Les principales fonctionnalités de StarUML sont :

* Prise en charge multiplateforme (MacOS , Windows et Linux).
* Conforme à la norme UML 2.x ;
* Exporter vers le fichier XMI ;
* ... etc .

4.2.2 VS Code (Visual Studio Code)

C'est un éditeur du code source développé par Microsoft pour Windows, Linux et MacOs, qui prend immédiatement en charge presque tous les principaux langages de programmation. Plusieurs d'entre eux sont inclus par défaut, par exemple Python et XML ...etc, mais d'autres extensions de langage peuvent être trouvées et téléchargées gratuitement à partir de VS Code Marketplace.
Il a été présenté lors de la conférence des développeurs Build d'avril 2015 comme un éditeur de code multiplateforme [27].

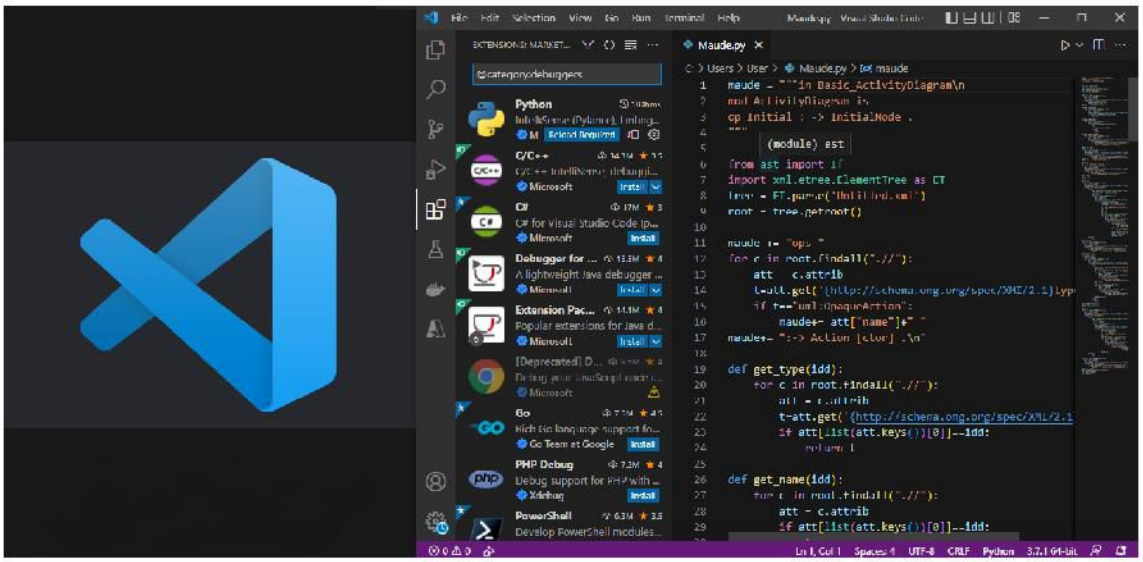


FIG. 4.2 : L'interface Visual Studio Code sur Windows 10.

4.2.3 Python

Python est un langage développé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles. Il permet -sans l'imposer-une approche modulaire et orientée objet de la programmation [28].
Le langage Python est un langage de programmation de haut niveau, facile à apprendre, orienté objet, totalement libre et terriblement efficace. Il est conçu pour produire du code de qualité, portable et facile à intégrer. Ainsi la conception d'un programme Python est très rapide et offre au développeur une bonne productivité. En tant que langage dynamique, il est très souple d'utilisation et constitue un complément idéal à des langages compilés.
Python est universel. Il peut être utilisé dans un grand nombre de contextes. Un autre avantage de Python est la richesse de ses bibliothèques. C'est toutes ses nombreuses et importantes caractéristiques qui font la force de ce langage. Contrairement à certains langages comme Java, Python est facile à manipuler et peut s'apprendre sans formation. Il est aussi portable et plus général que Java ; il est utilisé dans une grande variété de domaines. Il possède aussi des bibliothèques beaucoup plus riches que celle de Java.

4.2.4 La bibliothèque xml.etree

La bibliothèque etree (ElementTree) fait partie de la bibliothèque standard Python et contient de nombreux outils qui simplifient l'analyse et l'extraction des informations d'un document XML. Il

existe d'autres bibliothèques qui peuvent analyser des documents XML, mais etree est couramment utilisé et très facile à utiliser.

```
# La bibliothèque doit d'abord être importée, quelle que soit la manière dont nous extrayons les données.
import xml.etree.ElementTree as ET
```

4.3 Processus de génération des spécifications Maude

4.3.1 Création du diagramme d'activité

Comme nous avons mentionné précédemment, à l'aide de l'environnement de travail de StarUml, nous commençons par créer un diagramme d'activité (Voir la figure 4.4), et pour générer le fichier XMI :
« L'installation du package XMI se fait en suivant les étapes (Voir la figure 4.3) :
Installer l'extension
Pour installer une extension à partir du registre d'extension :

1- affichez **Extension Manager** en sélectionnant **Tools | Expansion Manager**...
2- sélectionnez le bouton **Registry** .
3- trouvez une extension à installer par recherche.
4- appuyez sur le bouton **install** de l'extension. »

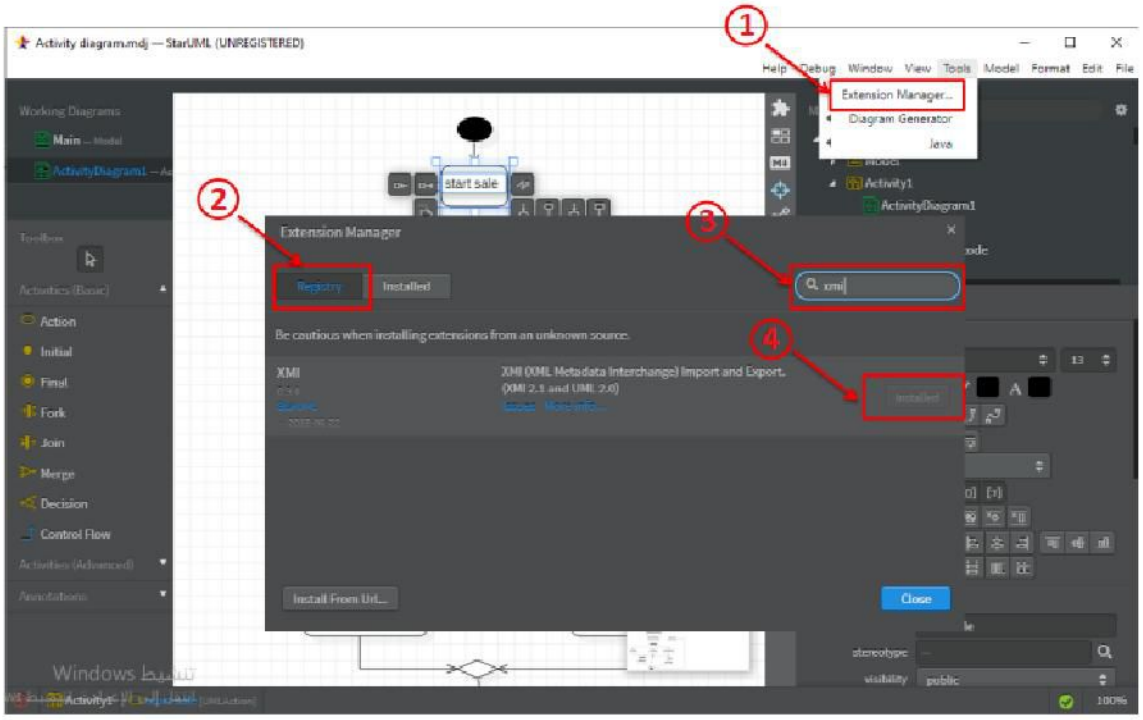


FIG. 4.3 : Étapes pour installer XMI dans StarUml.

- **XMI (XML Meta data Interchange)**
  C'est un standard pour l'échange d'informations de métadonnées UML basé sur XML, qui offre une représentation concrète des modèles sous forme de documents XML.

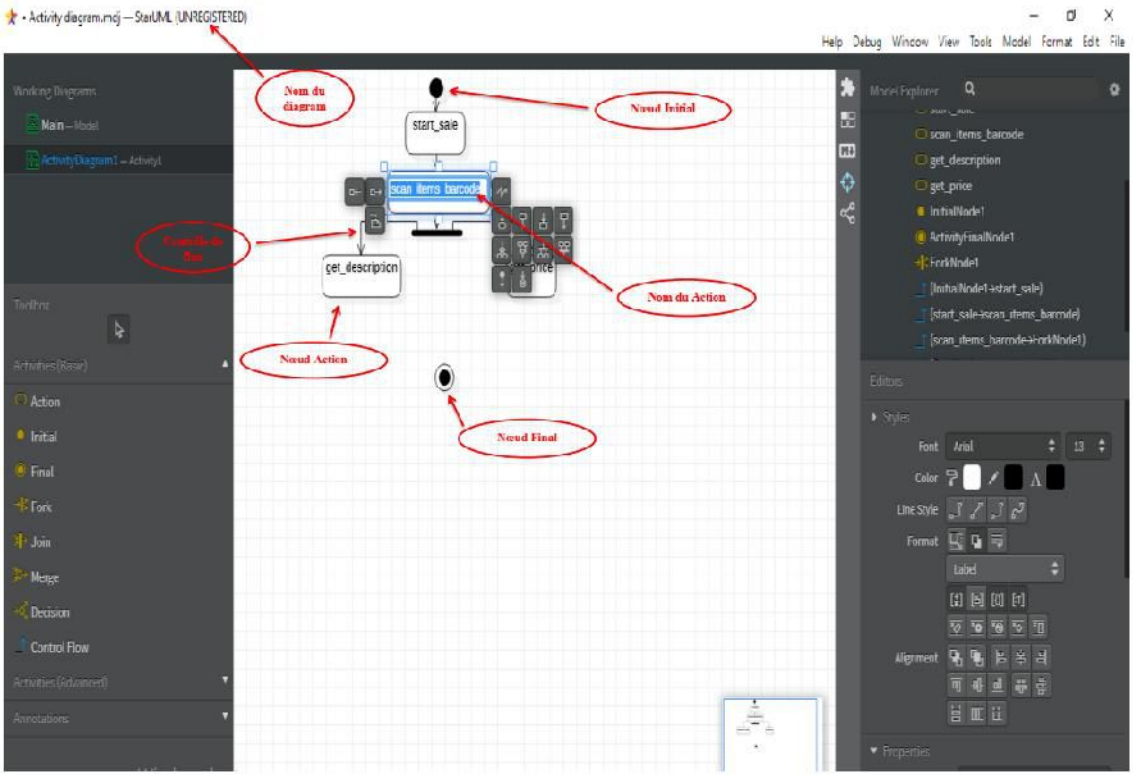


FIG. 4.4 : Création du diagramme d'activité.

4.3.2 Le Schéma global

Ce schéma nous montre une vue globale du modèle de transformation automatique utilisé lors de l'implémentation :

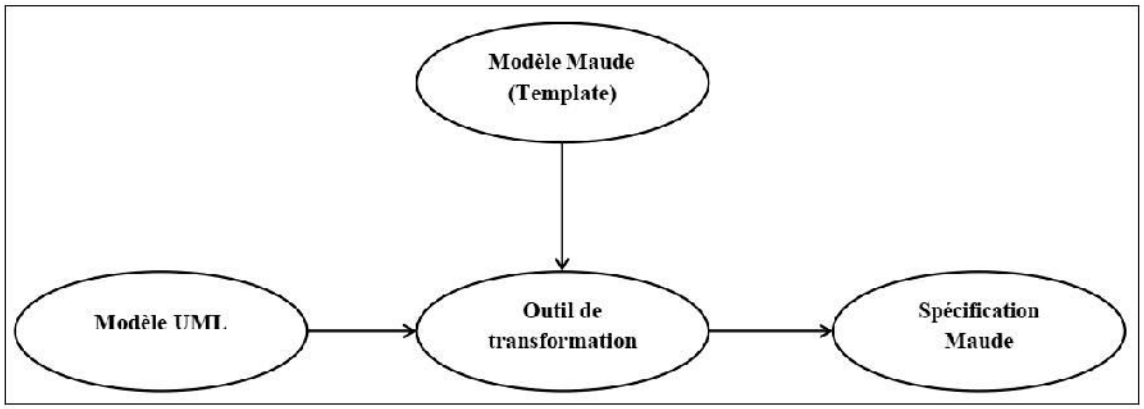


FIG. 4.5 : schéma global de modèle de transformation

4.3.3 Implémentation des règles de transformation

Dans cette étape, nous implémentons les règles de transformation pour générer un code Maude. pour terminer le processus d'implémentation, nous avons développé une feuille de texte (le fichier.txt) pour les modules Maude(système et fonctionnel) qui contient un template pour représenter les règles de transformation d'un diagramme d'activité vers Maude, basée sur le fichier XMI.
On résume le processus de transformation dans la figure 4.6 .

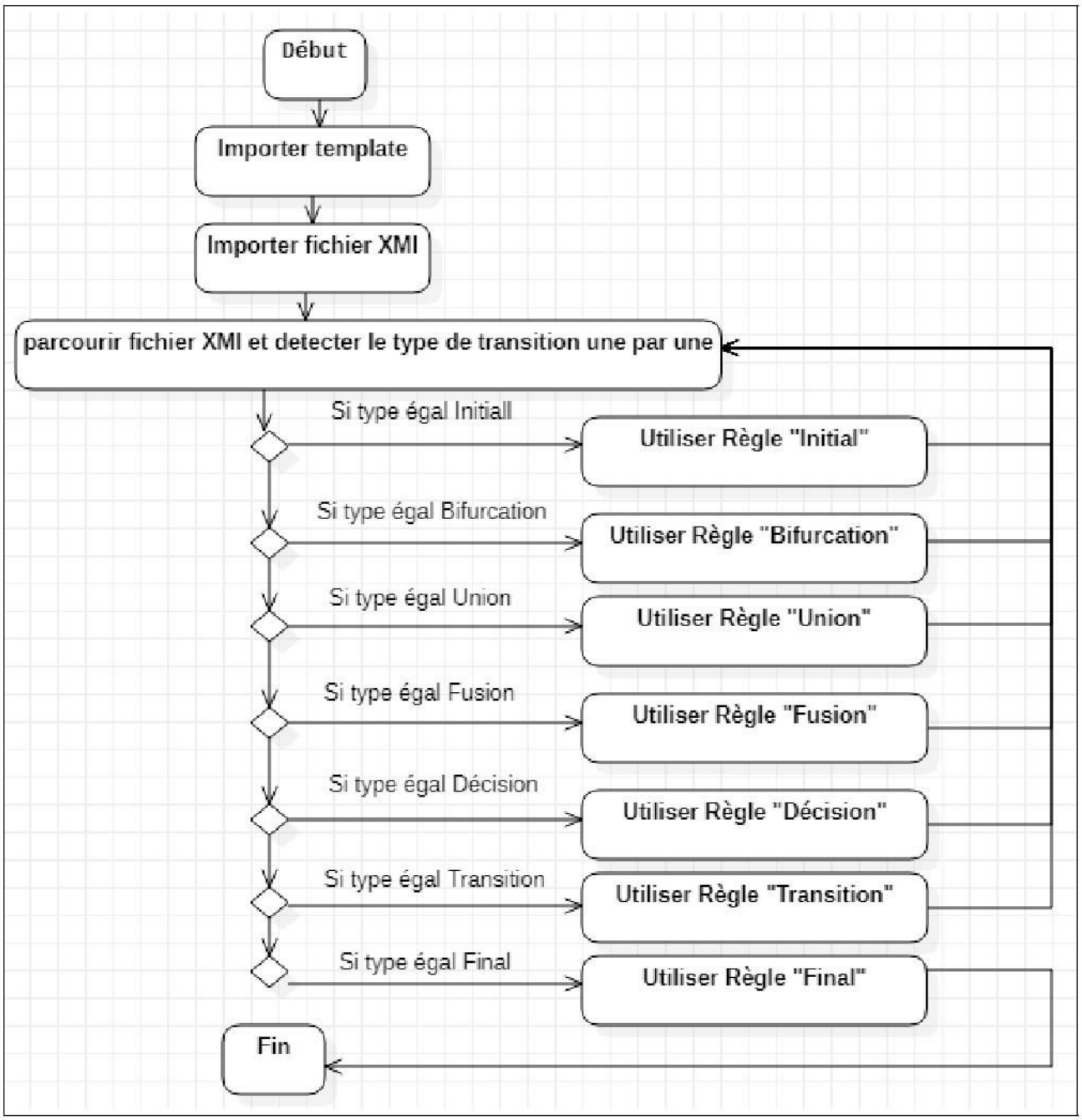


FIG. 4.6 : Le processus d'implémentation des règles de transformation

4.4 Exemple de transformation

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

4.4.1 Création du modèle UML(diagramme d'activité) :

Comme nous avons mentionné du chapitre 3, la première étape de transformation est la création de modèle UML.
La figure 4.7 présente la création de diagramme d'activité « StarUml ».

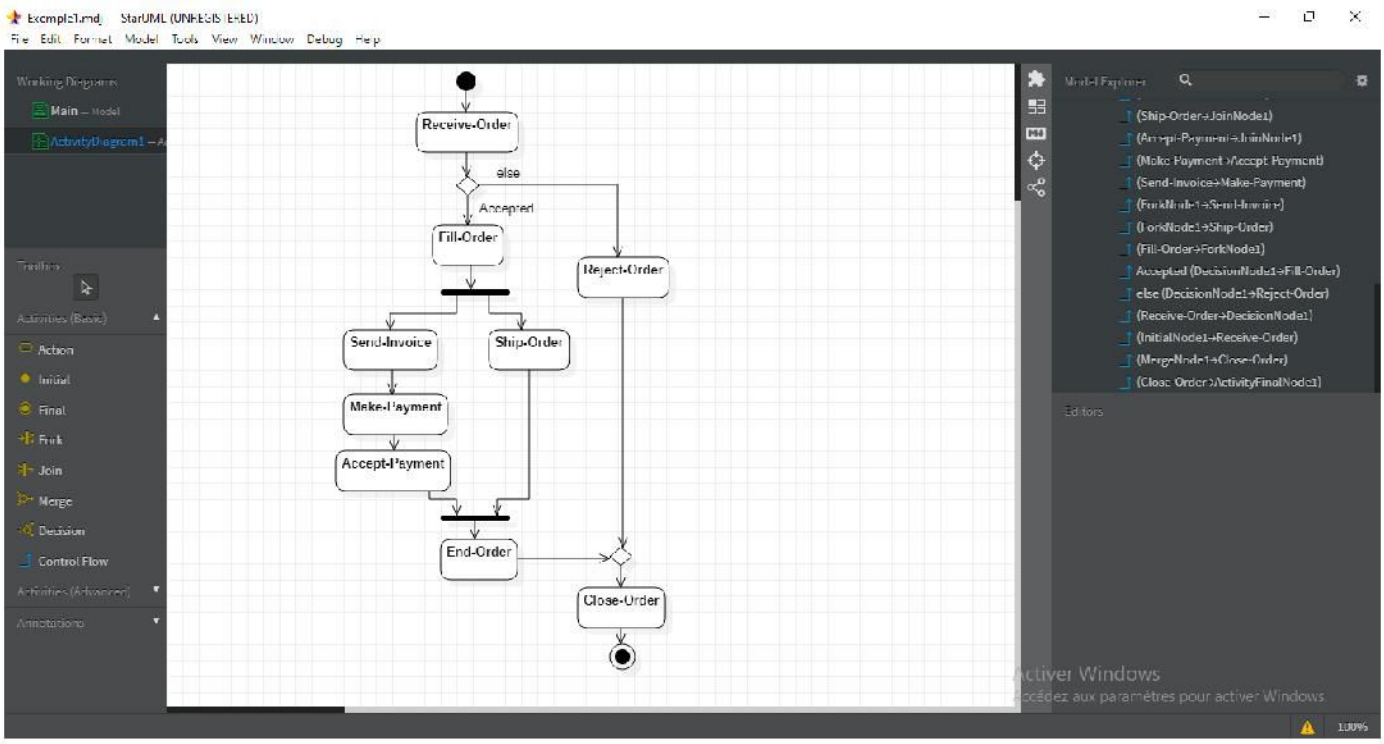


FIG. 4.7 : Diagramme d'activité.

4.4.2 Exportation du modèle en XMI :

Après l'enregistrement de notre modèle UML sur StarUml, on doit exporter le fichier XMI de ce diagramme, voir la figure suivante (figure 4.8).

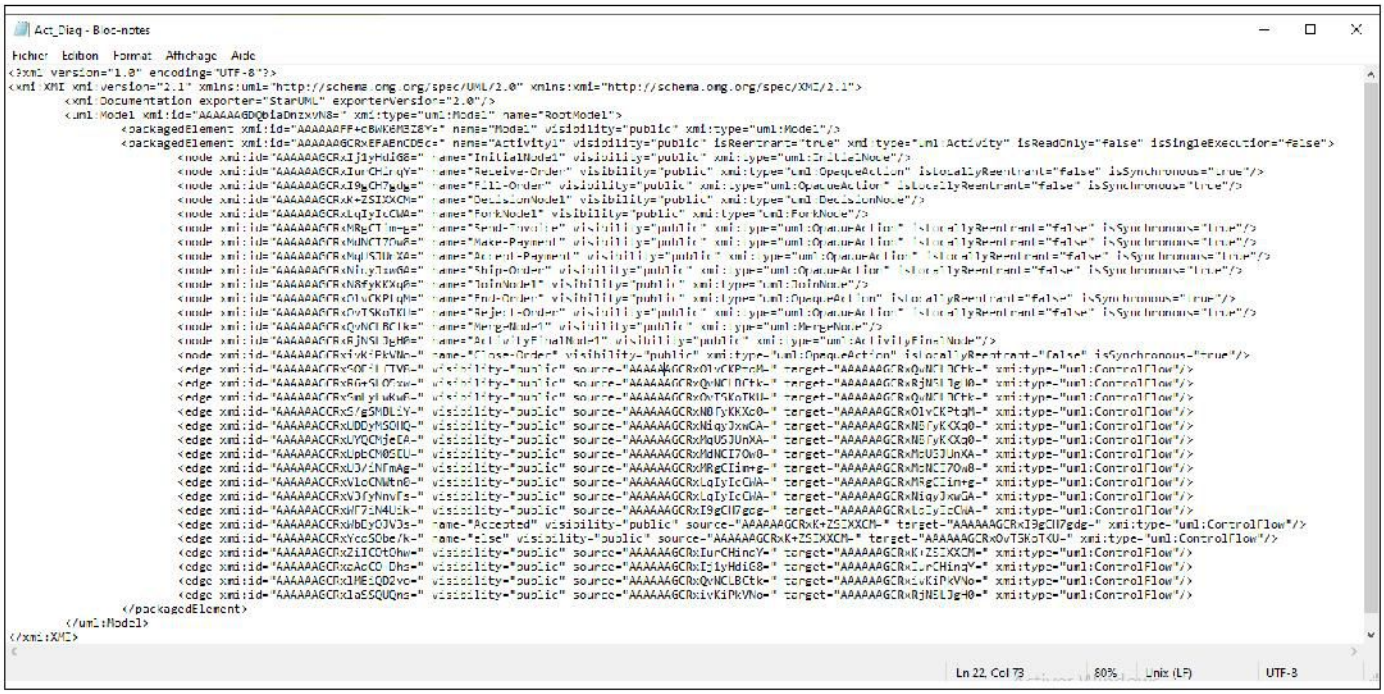


FIG. 4.8 : Le format XMI de diagramme d'activité.

4.4.3 Génération du Spécification Maude :

Dans la figure 4.9, le code Maude est le résultat de transformation de notre exemple de diagramme d'activité.

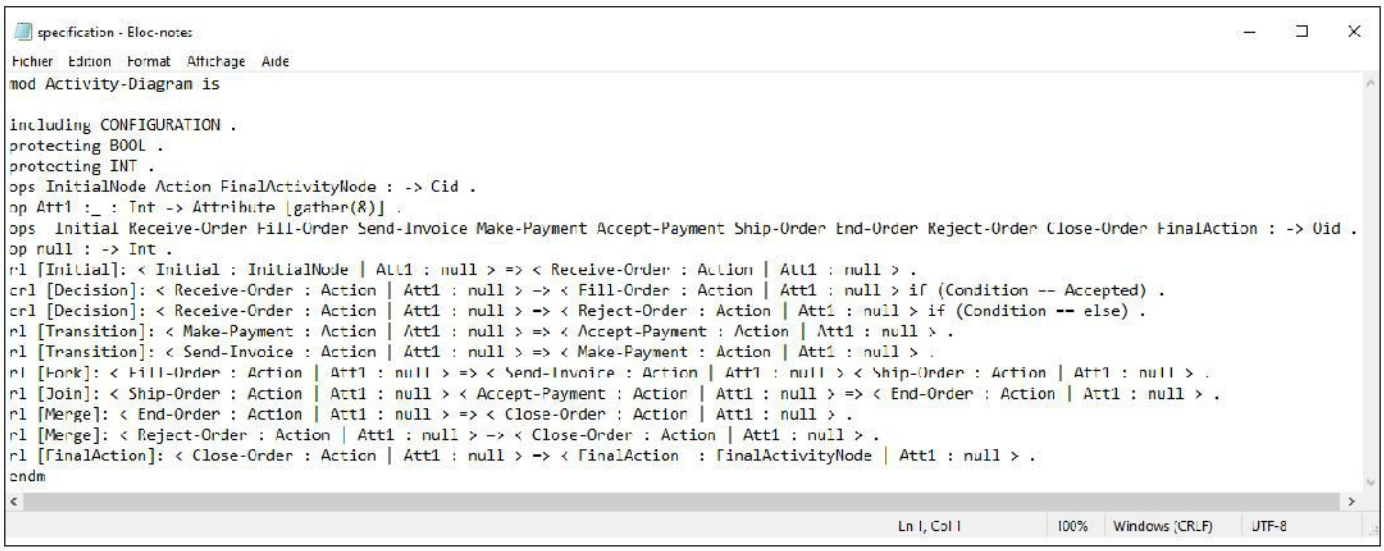


FIG. 4.9 : Le code Maude généré.

4.4.4 Exécution sous Maude pour vérification :

Afin d'assurer l'absence des erreurs syntaxiques et sémantiques du code généré, on a chargé ce code sur Maude en utilisant la command `load specification.maude`, et il nous a conffirmé l'absence d'érreurs. Seulement, on note que nous avons utilisé des mots abstraits — dans le code généré — tels que "Att1" "Condition" et "Accepted" car les développeurs généralement ne citent pas tous les attributs et/ou les détails des condition dans le diagramme UML. Par conséquent, le développeur doit les adaptés selon le cas qu'il traite effectivement.

4.5 Conclusion

Dans ce chapitre, nous avons présenté le processus complet pour la transformation des modèles UML vers Maude. On a mentionné l'environnement et les outils utilisés pour réaliser notre outil de transformation automatique du diagramme d'activité vers Maude. Ensuite, nous avons donné une explication du processus de génération des spécifications Maude. Eenfin nous avons donné un exemple de ce qui se passe pendant le processus de génération et les résultats que nous obtenons lors de l'exécution de notre outil.

Conclusion générale

Dans notre travail, nous avons proposé une approche de transformation de diagramme UML (diagramme d'activité) vers une spécification Maude. La méthode repose sur l'exploitation de la représentation XMI des diagrammes d'activités. Ensuite, la transforme en une spécification formelle écrite en Maude où les noeuds d'action, initiale et finale sont traités en tant que des objets et les autres noeuds sont représenté à l'aide des règles de réécritures.

Pour mettre en œuvre la transformation proposée, nous avons récupérer le fichier XMI du diagrameme d'activité puis on a l'a transformé vers la spécification correspondante Maude.

Pratiquement, la transformation des diagrammes UML vers la logique des réécriture sont fait généralement pour des fins de vérification du diagramme avant de passer à la phase d'implémentaion pour confirmer l'absence des erreurs dans les modèles proposés.

Enfin, notre prototype est limité avec les diagrammes d'activités et nous avons l'intention de continuer le travail sur cet axe de recherche et développer un outil qui permet de transformer les autre diagrammes UML, supporter l'OCL (Object Constraint Language) qu'UML utilise pour bien formaliser l'expression des contraintes, et même développer une version web de notre outil de transformation .

Bibliographie

[1] Celso, J., Freire, J., Giraudin, J. Agnès Front Atelier MODSI : Un Outil de Meta-Modelisation et de Multi-Modélisation. Laboratoire de Logiciels et Systèmes Réseaux, IMAG B.P. 72 - 38402 - Saint Martin d'H'eres Cedex – France.

[2] S. Cook, J. Daniels, Designing Object Systems - Object-Oriented Modelling with Syntropy. Prentice-Hall, 1994.

[3] Bowen, J. P., Hinchey, M. G. (1995). Seven more myths of formal methods. IEEE software, 12(4), 34-41.

[4] Asma, K., Meryem, B. (2017). Une approche dirigée par les modèles basée sur UML pour la mise en œuvre de services web composés.

[5] Joseph, G. David, G. (2008). UML 2 analyse et conception. Dunod, Paris.

[6] Audibert, L. (2007). UML 2. Institut Universitaire de Technologie de Villetaneuse–Département Informatique.

[7] Debrauwer, L., Van der Heyde, F. (2010). "UML2", ENI 2éme éditions.

[8] Vallée, F. (2005). UML pour les décideurs - Eyrolles.

[9] Hamrouche, H. (2010). Une approche de transformation des diagrammes d'activité d'uml vers csp basée sur la transformation de graphes. Mémoire de DEA, Ecole Doctorale en Informatique de l'Est.

[10] BOUDRIA, Y., CHIAL, R. (2020). Une Approche Automatique de Transformation des Diagrammes d'activité UML vers OWL-S (Doctoral dissertation, University of Jijel).

[11] Madani, K., Medjouri, S. (2020). Réalisation d'un outil de génération automatique de spécification Maude (nouvelle sémantique) des réseaux de Petri (Master dissertation, University of EL oued).

[12] Bendiaf, M. (2018). Spécification et vérification des systèmes embarqués temps réel en utilisant la logique de réécriture (Doctoral dissertation, UNIVERSITE MOHAMED KHIDER BISKRA).

[13] Djelloul, M., Mérouani, H. (2019). Un Profil UML/Maude pour les systèmes embarqués.

[14] Ölveczky, P. C. (2005). Formal modeling and analysis of distributed systems in Maude. Lecture Notes INF3230/INF4230, Department of Informatics, UNIVERSITY OF OSLO.

[15] Bruni, R., Meseguer, J. (2006). Semantic foundations for generalized rewrite theories. Theoretical Computer Science, 360(1-3), 386-414.

[16] Martí-Oliet, N., Meseguer, J. (1996). Rewriting logic as a logical and semantic framework. En first international workshop on rewriting logique and its application, volume4 of Electronic Notes in Theoritical Computer science, Elsevier.

[17] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (2007). All About Maude-A High-Performance Logical Framework : How to Specify, Program, and Verify Systems in Rewriting Logic (Vol. 4350). Springer.

[18] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (2007). MAUDE MANUAL (Version 2.3), SRI Inter, web http ://maude.cs.uiuc.edu/maude1/manual/.

[19] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Quesada, J. F. (2002). Maude : Specification and programming in rewriting logic. SRI International, http ://maude.cs.uiuc.edu/maude1/manual/.

[20] Duran, F., Meseguer, J. (1999, February). The Maude specification of Full-Maude. Technical report, SRI International, Computer Science Laboratory.

[21] Snook, C., Butler, M. (2006). Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology (TOSEM).

[22] BOUSBIA, L. A., TERCHA, A. (2021). Réalisation d'un outil de transformation automatique de diagramme de classes UML en FoCaLiZe (Master dissertation, UNIVERSITE Eloued).

[23] Anastasakis, K., Bordbar, B., Georg, G., Ray, I. (2010). On challenges of model transformation from UML to Alloy. Software Systems Modeling.

[24] Hettab, A. (2009). De M-UML vers les réseaux de Petri « Nested Nets » : Une approche basée transformation de graphes (Magistère dissertation, UNIVERSITE Constantine).

[25] Nouri, Z., Marir, T. (2020). Spécification formelle des systèmes multi-agents-Une approche basée sur le langage MAUDE.

[26] Kerkouche, E., Khalfaoui, K., Chaoui, A., Aldahoud, A. (2015). UML activity diagrams and maude integrated modeling and analysis approach using graph transformation. In Proceedings of the 7th International Conference on Information Technology (ICIT 2015) doi (Vol. 10).

[27] "Visual Studio Code - Code Editing. Redefined." https ://code.visualstudio.com/ (accessed Oct. 15, 2020).

[28] "cours-python pdf " https ://python.sdv.univ-paris-diderot.fr/cours-python.pdf