République algérienne démocratique et populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Abbes Laghrour Khenchela
Faculté des Sciences et de la Technologie
Département des Mathématiques et d'Informatique

Année : 2020

Mémoire de fin d'étude

Pour obtenir le diplôme
de Master
Option : **G**énie **L**ogiciel et **S**ystèmes **D**istribués

Vérification et Validation des diagrammes comportementaux d'UML2.0 basées sur OCL

.

Présenté par : Akachat brahim
Benabess rafik

Encadreur de mémoire :

Dr. Nabil Messaoudi - Enseignant à l'université Abbes Laghrour Khenchela

Devant le jury

- Ben attallah hassiba	Président
- Saadi souad	Examinatrice

Remerciements

Akachat Brahim,

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon étude et qui m'ont aidée lors de la rédaction de ce mémoire.
Je voudrais dans un premier temps remercier, mon encadreur de mémoire
Dr. Nabil Messaoudi, professeur de l'informatique à l'université Abbès Laghrour Khenchela, pour sa patience, sa disponibilité et surtout ses judicieux conseils, qui ont contribué à alimenter ma réflexion.

Je remercie également toute l'équipe pédagogique de l'université de khenchela et les intervenants professionnels responsables de ma formation, pour avoir assuré la partie théorique de celle-ci.
Je voudrais exprimer ma reconnaissance envers les amis et collègues qui m'ont apporté leur soutien moral et intellectuel tout au long de ma démarche.
Je tiens à témoigner toute ma reconnaissance aux personnes, pour leur aide dans la réalisation de ce mémoire.

Un grand merci à ma femme pour ses conseils concernant mon style d'écriture, elle a grandement facilité mon travail.
Je remercie mes très chers parents, Lahcen et Khadidja, qui ont toujours été là pour moi. Je remercie mes filles Aya et youssra, et mon fils Youcef, pour leurs encouragements.

À tous ces intervenants, je présente mes remerciements, mon respect et ma gratitude.

Remerciements

Benabesse rafik,

La réalisation de ce mémoire a été possible grâce au concours de plusieurs personnes à qui je voudrais témoigner toute ma gratitude.

Je voudrais tout d'abord adresser toute ma reconnaissance à mon encadreur le Dr Nabil Messaoudi, pour sa patience, sa disponibilité et surtout ses judicieux conseils, qui ont contribué à alimenter ma réflexion.

Je désire aussi remercier les enseignants de l'université Abbas Laghrour, qui m'ont fourni les outils nécessaires à la réussite durant toute mon étude Master 2.

Aussi je voudrais remercier mon binôme pour les efforts qu'il a déployé pendant toute cette période

Un très grand merci à ma femme pour son soutien moral et psychologique ainsi que sa patience inestimable.

Résumé

La représentation des différentes vues du système, accentue la difficulté du processus de vérification et de validation. Les différents diagrammes dans un modèle UML se complètent et présentent plusieurs chevauchements sémantiques et syntaxiques entre eux. Dans ce mémoire, nous proposons une vérification et validation des modèles UML à base de relations de dépendance inter-diagrammes. Notre travail consiste à assurer la cohérence inter-diagrammes par la formalisation des règles de cohérences utilise le langage de spécification formel OCL (Object Constraint Language). La formalisation des règles est à base du méta-modèle d'UML.

Abstract

T*he representation of the different views of the system accentuates the difficulty of the verification and validation process.*

The different diagrams in a UML model complement each other and have several semantic and syntactic overlaps between them.

In this thesis, we propose a verification and validation of UML models based on inter-diagram dependency relations.

Our work consists in ensuring inter-diagram consistency by formalizing consistency rules using the formal specification language OCL (Object Constraint Language). The formalization of the rules is based on the UML meta-model.

ملخص

تمثيل وجهات النظر المختلفة للنظام تزيد من صعوبة عمليات التحقيق. مختلف المخططات الموجودة في النموذج تكمل بعضها البعض ولها العديد من التداخلات الدلالية والنحوية فيما بينها.

في هذه المذكرة إقترحنا طريقة عمل للتحقيق من صحة تناسق النماذج أومل 2.0 فيما بينها بناءا على العلاقات المرتبطة بين المخططات.

يتمثل عملنا في ضمان التنسيق بين المخططات من خلال تكوين مجموعة من القواعد وذلك بإستخدام لغة المواصفات أوسل. إضفاء الطابع الرسمي على القواعد يستند على نموذج التعريف أومل.

Table des matières

Liste des figures

Liste des tables

Chapitre 1

Introduction générale

1. Introduction :

Dans ce chapitre, nous présenterons les motivations pour la détection des incohérences entre les diagrammes de cas d'utilisation et les diagrammes d'activités d'UML2.0 ainsi que les objectifs de ce travail. Un aperçu de la structure du mémoire ainsi qu'une section des travaux connexes sera également présentée.

1.1. Motivation :

UML est une notation basée sur la technologie des objets pour les phases d'analyse et de conception exprimant les besoins de l'utilisateur.

Nous construisons les modèles afin de mieux comprendre la structure et le comportement de nos systèmes logiciels.

La modélisation comprend un ensemble de modèles décrivant des points de vue différents exprimés dans divers formalismes, Les éléments de modélisation et de leur représentation graphique, c'est ainsi que le problème de la cohérence entre différentes vues de la modélisation devient un problème prépondérant. Par exemple, les descriptions des diagrammes d'activités sont-elles cohérentes avec celles des diagrammes des cas d'utilisations dans une description à base d'UML ? Actuellement, en l'absence d'outils de vérification évolués, les incohérences ne peuvent être détectées que tardivement ce qui implique un retour aux phases initiales de développement.

1.2. Objectif de mémoire :

L'objectif de ce mémoire est de décrire une vérification et une validation automatique de la cohérence entre les diagrammes de cas d'utilisations et

diagrammes d'activités. Les règles de cohérences sont écrites en utilisant le langage de spécification formel OCL.

Les propriétés comportementales du modèle son vérifiés et validés lors de la communication entre les diagrammes de cas d'utilisation et les diagrammes d'activités.

1.3. Structure du mémoire :

Ce mémoire est organisé comme suit : le chapitre 2 composé de deux parties

La partie A : un bref aperçu sur UML en particulier les diagrammes d'activités et diagrammes des cas d'utilisations et la partie B : la vérification de la cohérence dans UML (UML consistency checking).Le chapitre 3 décrit Le langage OCL comme moyen de vérification pour le langage UML (modèles et méta-modèles)

Le chapitre 4 d'écrit comment utiliser OCL pour vérifier et valider la cohérence entre les diagrammes des cas d'utilisations et les diagrammes d'activités. Puis dans le chapitre 5, nous illustrerons notre apport par une étude de cas avec un outil. Enfin, notre conclusion et nos perspectives sont élaborées au chapitre 5.

1.4. Travaux Connexes :

La vérification et la validation des systèmes et des modelés est un domaine de recherche très vaste et très complexe, selon la nature des travaux nous avons réussi de les grouper en 02 grandes catégories : des travaux qui traitent la transformation des modèles en modèles utilisant des langages de transformations de modèles spécifiques, est en seconde lieu les travaux qui traitent la transformation des modèles en code à l'aide d'un langage exécutable.

Les sous-sections qui suivent décrivent les principaux domaines de recherche connexes et une évaluation par rapport à d'autres travaux.

1.4.1. Transformations modèle vers modèle :

Dans **[**1**]** montrent l'utilisation de l'OCL pour vérifier la cohérence du modèle UML. Ils utilisent un outil d'environnent d'OCL (OCLE) qui permet la spécification des contraints OCL à la fois au niveau modèle et Meta-modèle. Cet

outil utilise des règles spécifiées sur les modèles pour vérifier les incohérences. Le travail est démontré sur deux exemples, où l'OCLE est utilisé pour spécifier des règles liées à UML.

Les auteurs dans **[**2,3**]** utilisent un outil d'environnent de spécification USE basé sur UML pour valider les modèles et les contraintes OCL. Les auteurs définissent formellement le diagramme de classes UML et fournissent une définition précise du langage OCL de telle sorte qu'une interprétation ambiguë à la fois pour le diagramme de classes UML et OCL est évitée.

Sur la base de spécifications formelles, l'outil a été développé pour soutenir l'analyse, la simulation, la transformation et la validation des Modèles UML avec des contraintes OCL et a été utilisé pour valider les règles de bonne forme (well-formedness) dans le diagramme de classe UML.

Dans **[**3**]** les auteurs proposent **l'**application de l'outil USE pour valider les spécifications d'OCL à la base d'un cas industriel du système avancé de contrôle automatique des trains (BART : Bay Area Rapid Transit). Les spécifications qui expriment la vitesse du train, l'accélération et les positions sont mises à jour et validés à l'aide de contraintes OCL et avec plusieurs scénarios de test.

Ils générant un diagramme d'objets dans un fichier de commandes comprenant des instructions de création d'objets et de création de liens mais ce dernier montre une violation avec quelques incohérences spécifiées dans OCL.

Dans **[**5–6**]** proposent une génération automatisée utilisant la notion d'instance d'un système et basée sur l'approche de validation des modèles UML et OCL. Une instance représente les états du système qui se composent d'objets ayant des attributs, des valeurs et des liens. Ils utilisent l'outil USE avec un langage ASSL (A Snapshot Sequence Language) pour la construction des snapshots cohérentes.

Dans **[**23**]** les auteurs présentent une technique de découpage qui partitionne le modèle en tranches et vérifient chaque tranche indépendamment. Cependant, l'approche n'est applicable que sur les diagrammes de classes qui sont annotés

avec des contraintes OCL libre et d'autres propriétés spécifiques doivent être vérifiées.

Dans **[**9**]** les auteurs présentent une méthodologie qui traduit les modèles comportementaux et structurels UML enrichis de contraintes OCL en symboles formels. Les auteurs discutent également la façon dont la formulation symbolique aide à effectuer diverses tâches de vérification telles que la cohérence, l'accessibilité,… etc. Ils utilisent le framework SMT pour résoudre la formulation symbolique efficacement. L'approche proposée est semi automatisable.

Il existe également de nombreuses applications de la vérification des modèles, telles que la vérification de la conception des systèmes embarqués utilisant SVA [8], transformation des contraintes OCL en CSP [10].

La plupart des approches liées pour modéliser la validation et la vérification convertissent OCL en une sorte de formalisme et un certain nombre d'entre eux ne prennent en charge qu'un sous-ensemble limité d'OCL.

1.4.1 Transformations modèle vers code :

Il existe également des travaux importants qui traitent la transformation des modèles en code à l'aide d'un langage exécutable, plusieurs travaux brièvement présentés dans le tableau suivant :

Papier	Domaine	Stratégie utilisée	Couverture OCL	OCL transformation conversion	Cas d'étude
[11]	Génération de données de test	Analyse de partition	Invariants, pré-Post conditions	VDM – DNF	Planificateur de processus
[12]	Génération de cas de test	Fonction arbre Technique de minimisation	Pré-post conditions	CSP	Système de billetterie
[13]	Test de spécification	Traverse de graphe	Expression OCL	HOL	Liste liée
[14]	Test de programme java	Assertion, random	Pré-post conditions	ASPECTJ	Programme de fidélité
[15]	Unité de test	Basé sur la mutation	Expression OCL	XMI	Collections

Tableau1 : Résumé des approches de génération de code d'OCL [34]

Chapitre 2

Partie 1

Un aperçu sur UML en particulier les diagrammes des cas d'utilisations et les diagrammes d'activités

1. Introduction

UML (Unified Modeling Language) [16] est un langage de modélisation unifie reconnu actuellement par toute la communauté des chercheurs dans le domaine d'objet.

La description de la programmation par objets a fait ressortir l'étendue du travail conceptuel nécessaire: définition des classes, de leurs relations, des attributs et méthodes, des interfaces etc.

Pour programmer une application ou vérifier un système, il ne convient pas de se lancer tête baissée dans l'écriture du code, il faut d'abord organiser ses idées, les documenter, puis organiser la réalisation en définissant les modules et étapes de la réalisation. 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.

UML, dans sa version 2.0, comporte plus de treize diagrammes [16] qui peuvent être combinés pour définir toutes les vues d'un système. Ils se répartissent en trois grands groupes

Diagrammes structurels : ou diagrammes statiques (UML structure)

- Diagramme de classes (class diagram)
- Diagramme d'objets (objectdiagram)
- Diagramme de composants (component diagram)
- Diagramme de déploiements (Deploymentdiagram)

- Diagramme de paquetages (Package diagram)
- Diagramme de structures composites (Composite structure diagram)

Diagrammes comportementaux : ou diagrammes dynamiques (UML Behavior)

- Diagramme des cas d'utilisations (Use case diagram)
- Diagramme d'activités (Activitydiagram)
- Diagramme d'état-transitions (State machine diagram)

Diagrammes d'interactions (Interaction diagram)

- Diagramme de séquences (Sequencediagram)
- Diagramme de communications (Communication diagram)
- Diagramme globale d'interactions (Interaction overviewdiagram)
- Diagramme du temps (Timing diagram)

Ces diagrammes, d'une utilité variable selon le cas, ne sont pas nécessairement tous produits durant la modélisation du système.

Dans notre projet nous avons principalement concentré sur des deux types de diagrammes qui sont : diagrammes des cas d'utilisations et les diagrammes d'activités.

On choisit le diagramme de cas d'utilisation parce qu'apporte une vision utilisateur et absolument pas une vision informatique. Il leur faut donc un moyen simple d'exprimer leurs besoins. Il ne nécessite aucune connaissance informatique et Il ne liste que des fonctions générales essentielles et principales sans rentrer dans les détails. C'est précisément le rôle des diagrammes de cas d'utilisations qui permettent de recueillir, d'analyser et d'organiser les besoins, et de recenser les grandes fonctionnalités d'un système.

Il s'agit donc de la première étape UML de représenter, d'analyser et de capturer le comportement d'un système, d'un sous-système, d'une classe ou d'un composant tel qu'un utilisateur extérieur le voit. Il Permet de définir de manière précise les frontières du système à modéliser et montre les interactions entre le système et son environnement extérieur et aussi les dépendances existant entre les cas d'utilisations.

Il ne faut pas négliger cette première étape pour produire un logiciel conforme aux attentes des utilisateurs.

Le deuxième type c'est Les diagrammes d'activités qui 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.

Dans la phase de conception, les diagrammes d'activités sont particulièrement adaptés à la description des cas d'utilisations. Plus précisément, ils viennent illustrer et consolider la description textuelle des cas d'utilisations **(*chapitre5* section [1.11]]).**

De plus, leur représentation sous forme d'organigrammes les rend Facilement intelligibles et beaucoup plus compréhensives.

Nous pouvons attacher un diagramme d'activité à n'importe quel élément de modélisation afin de visualiser, spécifier, construire ou documenter le comportement de cet élément

Ils peuvent être utilisés pour détailler des situations dans lesquelles un traitement parallèle peut avoir lieu lors de l'exécution de certaines activités.

Notre approche de vérification de modèles UML a deux objectifs : 1) permettre la vérification de la cohérence d'un méta-modèle UML (ensemble de diagrammes), et 2) Réduire la difficulté de la vérification d'une propriété complexe en utilisant le langage de contrainte OCL

Ce mémoire est organisé comme suit ; dans la deuxième section, nous présentons

Un bref aperçu sur UML en particulier les diagrammes d'activités et les diagrammes des cas d'utilisations, dans la troisième section nous présentons le langage OCL puis dans la quatrième section nous montrons notre approche en précisant comment utiliser OCL pour vérifier la cohérence entre les diagrammes des cas d'utilisations et les diagrammes d'activités. Dans la section 5, nous testons notre approche avec une étude de cas. Finalement, nous donnons les conclusions et les perspectives de ce travail.

2. Diagramme des cas d'utilisations

Le diagramme des cas d'utilisations (Use Case Diagram) constitue la première étape de l'analyse UML en :

- Modélisant les besoins des utilisateurs.
- Identifiant les grandes fonctionnalités et les limites du système.
- Représentant les interactions entre le système et ses utilisateurs.

2.1. Les éléments d'un diagramme des cas d'utilisations

2.1.1. Les acteurs:

Un acteur est l'idéalisation d'un rôle joué par une personne externe, un processus ou une chose qui interagit avec un système [18].

Les acteurs se représentent sous la forme d'un petit personnage (stick man) ou sous la forme d'une case rectangulaire (appelé classeur) avec le mot clé «Acteur». Chaque acteur porte un nom.

Fig1 – Exemple de représentation d'un acteur sous forme d'une Personne et sous forme d'un classeur [11]

Remarque importante : En UML, une annotation entre guillemets est appelé '**stéréotype'**. Cela permet de préciser et de mieux caractériser l'élément à qui il s'adresse.

2.1.2. Les cas d'utilisations

Un cas d'utilisation est une unité cohérente d'une fonctionnalité visible de l'extérieur. Il réalise un service de bout en bout, avec un déclenchement, un déroulement et un fin, pour l'acteur qui l'initie. Un cas d'utilisation se représente par une ellipse [**Fig2**] contenant le nom du cas (un verbe à l'infinitif), et optionnellement, au-dessus du nom, un stéréotype (**Fig5**) [18].

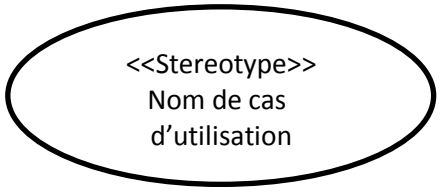


Fig2 – Exemple de représentation d'un cas d'utilisation [18]

Dans le cas où l'on désire présenter les attributs ou les opérations du cas d'utilisation, il est préférable de le représenter sous la forme d'un classeur stéréotypé « use case » **[Fig3].**

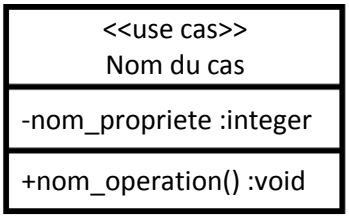


Fig3 – Exemple de représentation d'un cas d'utilisation
Sous la forme d'un classeur [18]

2.2. Représentation d'un diagramme de cas d'utilisation

Comme le montre la figure **[Fig4],** la frontière du système est représentée par un cadre. Le nom du système figure à l'intérieur du cadre, en haut. Les acteurs sont à l'extérieur et les cas d'utilisations à l'intérieur.

2.3. Relations dans les diagrammes des cas d'utilisations

2.3.1. Relation entre acteurs et cas d'utilisations

- **La relation d'association**
  - A chaque acteur est associé un ou plusieurs cas d'utilisations, la relation d'association peut aussi être appelée relation de communication.
  - Elle est représentée par un trait reliant l'acteur et le cas d'utilisation. Nous pouvons rajouter sur ce trait un stéréotype qui va préciser la relation de communication (« communicate ») [17].

Note : Nous prenons l'exemple d'un distributeur automatique de billets (DAB) car il est simple à utiliser et avec cela nous pouvons expliquer plus

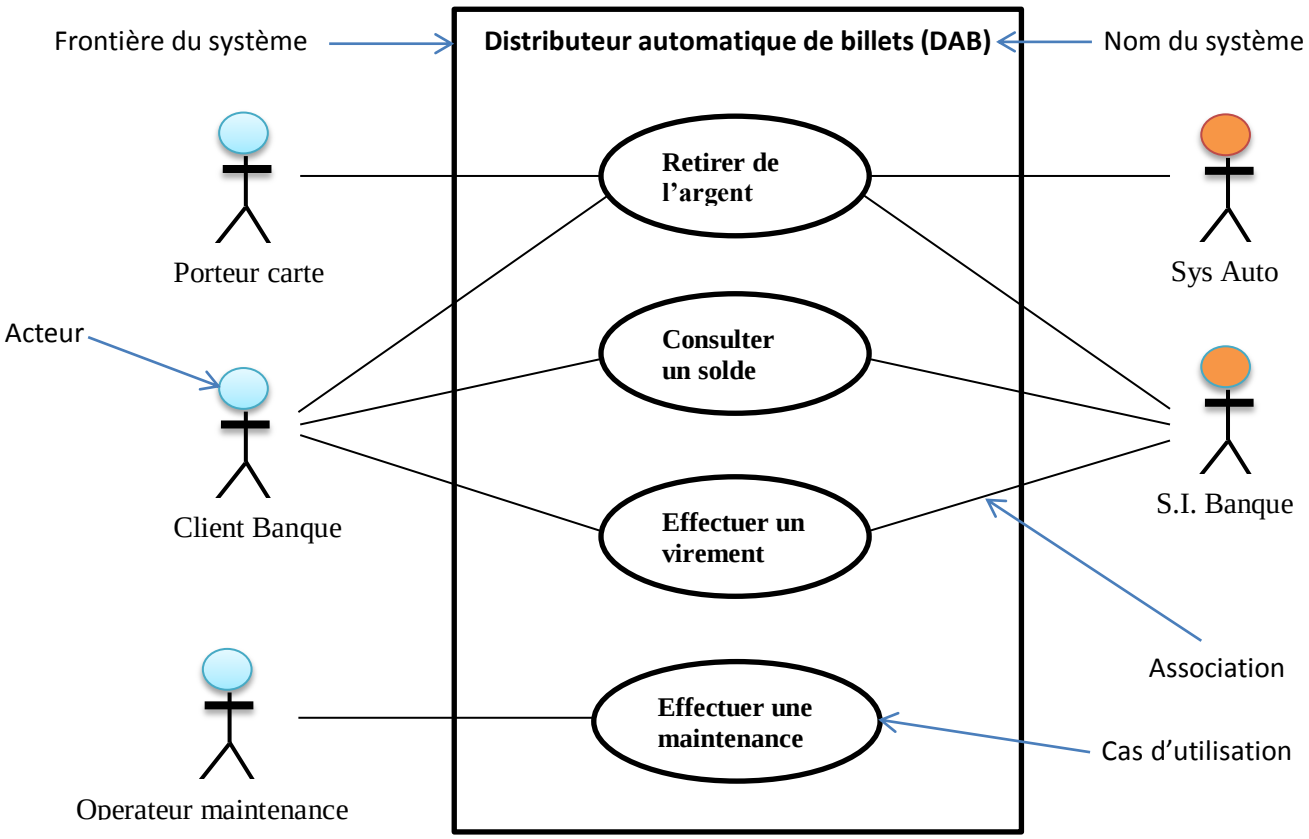


Fig4 – Exemple de diagramme de cas d'utilisation modélisant une borne d'accès à une banque.[17]

- **Multiplicité :**

  Lorsqu'un acteur peut interagir plusieurs fois avec un cas d'utilisation, il est possible d'ajouter une multiplicité sur l'association du côté du cas d'utilisation. Le symbole * signifie plusieurs. Exactement n s'écrit tout simplement n, n..m signifie entre n et m, etc. Préciser une multiplicité sur une relation n'implique pas nécessairement que les cas sont utilisés en même temps [17].

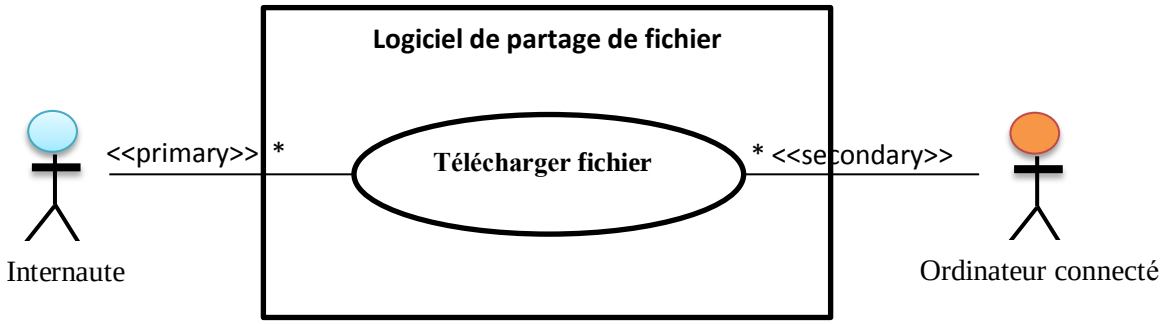


Fig5 – Diagramme de cas d'utilisation représentant un logiciel de partage de fichiers [17].

Acteurs principaux et secondaires

Un acteur est qualifié de principal pour un cas d'utilisation lorsque ce cas rend service à cet acteur. Les autres acteurs sont alors qualifiés de secondaires. Un cas d'utilisation a au plus un acteur principal. Un acteur principal obtient un résultat observable du système tandis qu'un acteur secondaire est sollicité

Pour des informations complémentaires. En général, l'acteur principal initie le cas d'utilisation par ses sollicitations. Le stéréotype « primary » vient orner l'association reliant un cas d'utilisation à son acteur principal, le stéréotype « secondary » est utilisé pour les acteurs secondaires [Fig5] [18].

2.3.2. Les relations entre les cas d'utilisations

- **Relation d'inclusion :**

  La relation d'inclusion sert à enrichir un cas d'utilisation par un autre cas d'utilisation (c'est une sous fonction).

  Dans un diagramme des cas d'utilisations, cette relation est représentée par une flèche pointillée reliant les 2 cas d'utilisations et munie du stéréotype «include».

  L'inclusion permet de [18] :

  - Partager une fonctionnalité commune entre plusieurs cas d'utilisations.
  - Décomposer un cas d'utilisation complexe en décrivant ses sous fonctions

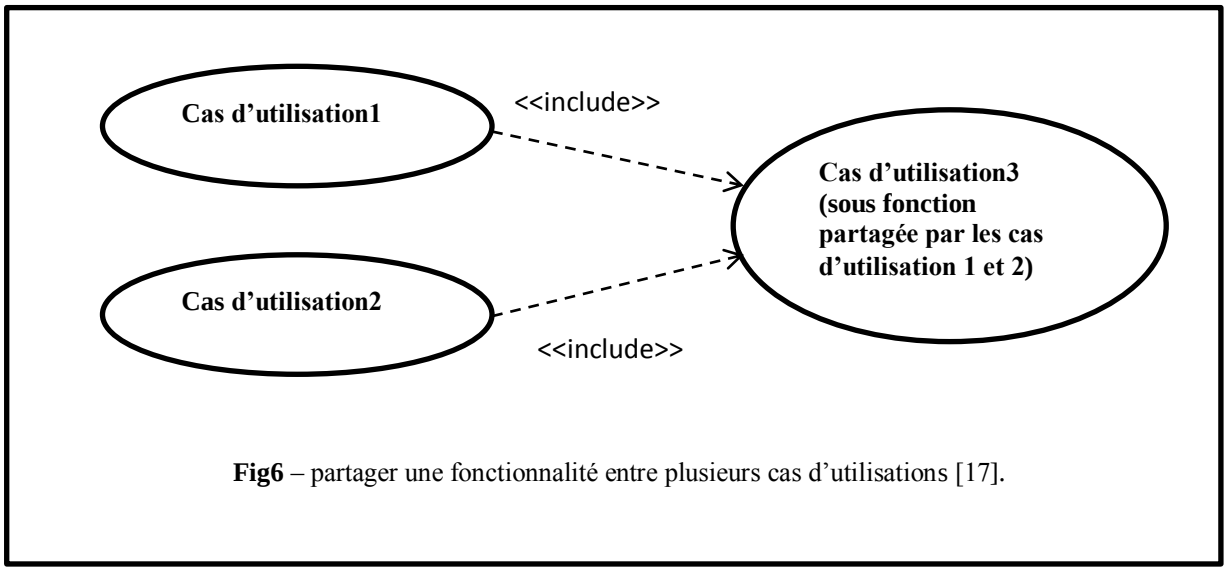


Fig6 – partager une fonctionnalité entre plusieurs cas d'utilisations [17].

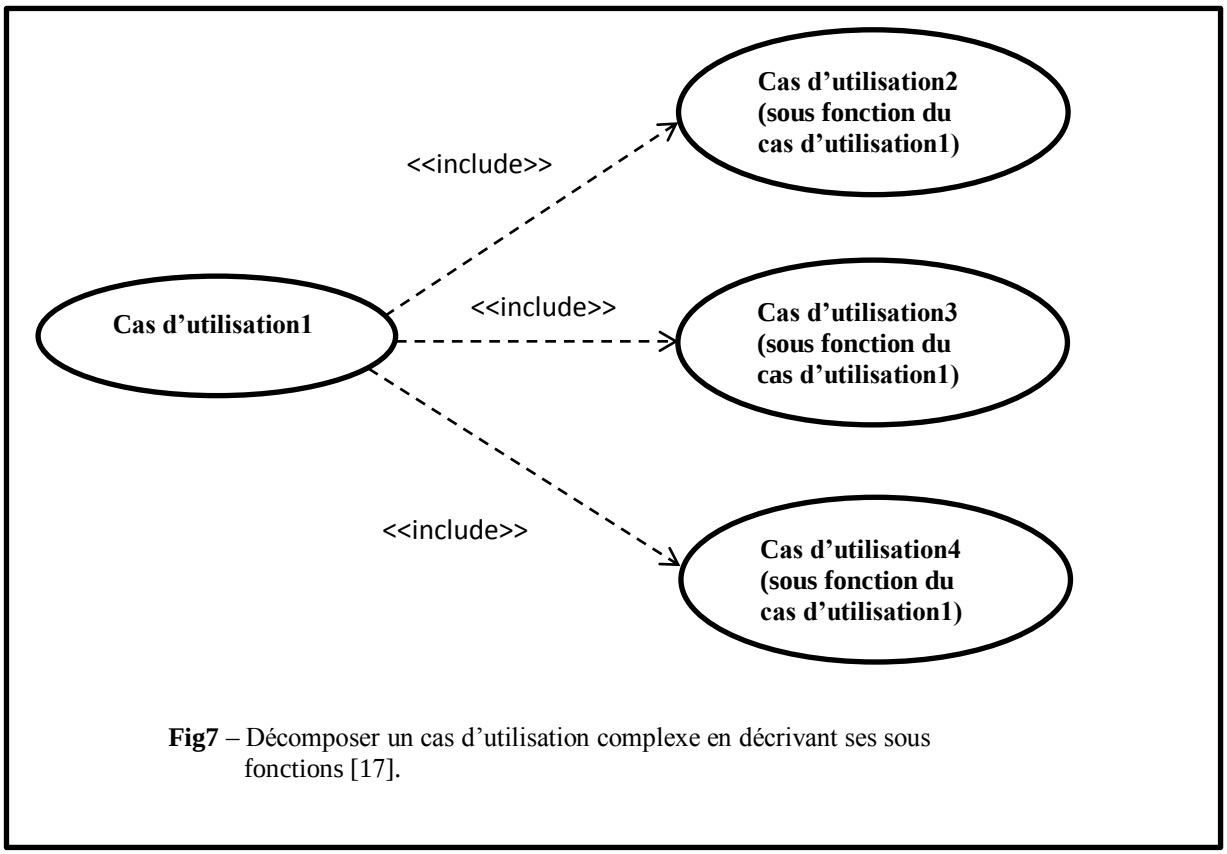


Fig7 – Décomposer un cas d'utilisation complexe en décrivant ses sous fonctions [17].

- **Relation d'extension :**

On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A : contrairement à l'inclusion, l'extension est optionnelle. Cette dépendance est symbolisée par le stéréotype « extend » [**Fig8]**

L'extension peut intervenir à un point précis du cas étendu. Ce point s'appelle le point d'extension.

Il porte un nom, qui figure dans un compartiment du cas étendu sous la rubrique point d'extension, et est éventuellement associé à une contrainte indiquant le moment où l'extension intervient. Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme d'une note [18].

La **[Fig8]** présente l'exemple d'une banque où la vérification du solde du compte n'intervient que si la demande de retrait dépasse 100Dinar.

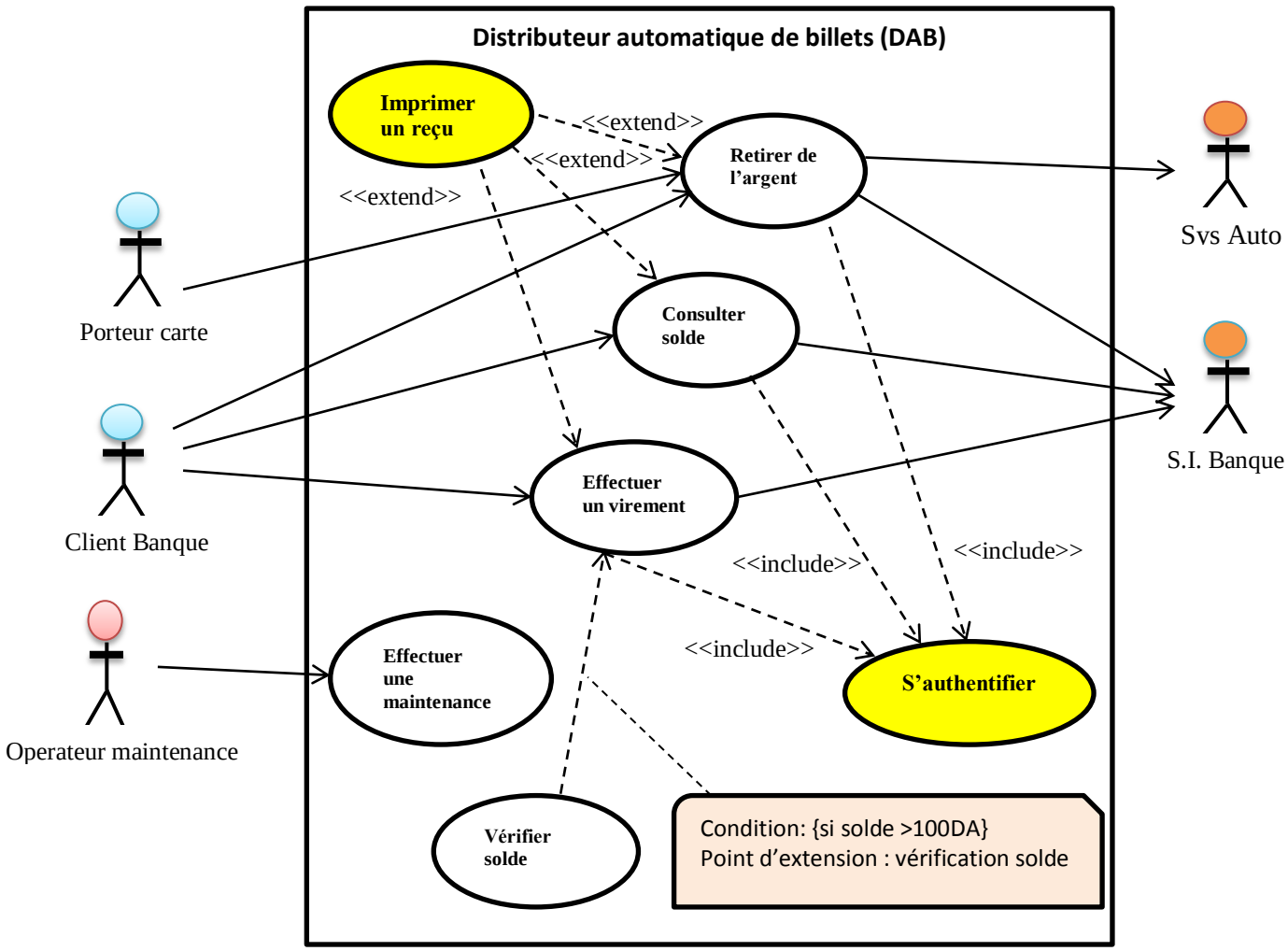


Fig8 – Exemple de diagramme de cas d'utilisation relation d'extension [17].

- **Relation de généralisation:**

Un cas A est une généralisation d'un cas B si B est un cas particulier de A. Dans la figure **[Fig9]**, la consultation d'un compte via Internet est un cas particulier de la consultation. Cette relation de généralisation/spécialisation est présentée dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages orientés objet [18].

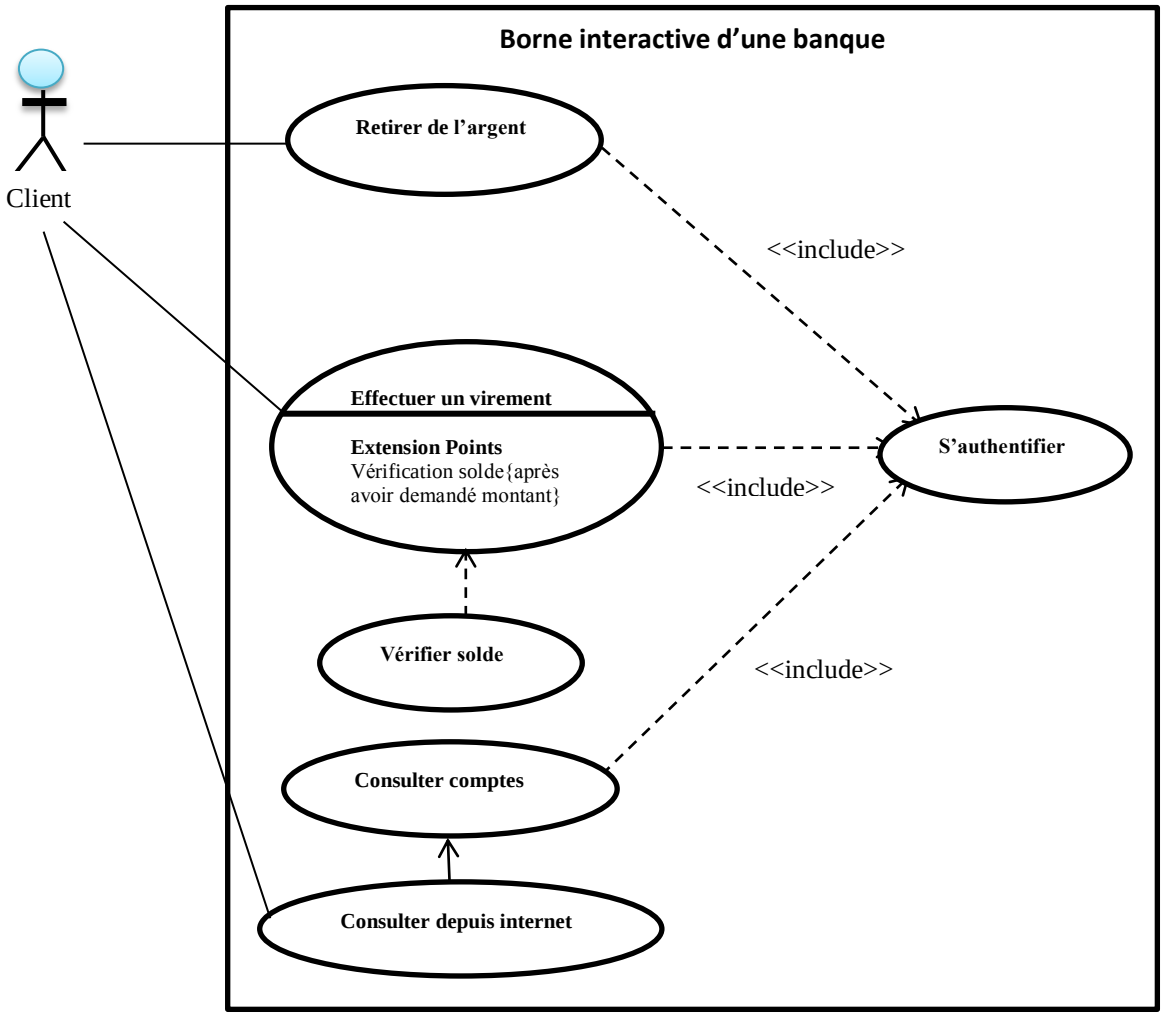


Fig9 – Exemple de diagramme de cas d'utilisation [17].

2.4. Description textuelle d'un cas d'utilisation

À chaque cas d'utilisation doit être associée une description textuelle des interactions entre l'acteur et le système et les actions que le système doit réaliser en vue de produire les résultats attendus par les acteurs.

UML ne propose pas de présentation type de cette description textuelle. La description textuelle d'un cas d'utilisation est articulée en six points :

• **Objectif** – Décrire succinctement le contexte et les résultats attendus du cas D'utilisation.

• **Acteurs concernés** – Le ou les acteurs concernés par le cas doivent être identifiés en précisant globalement leur rôle.

• **Pré conditions** – Si certaines conditions particulières sont requises avant l'exécution du cas, elles sont à exprimer à ce niveau.

• **Post conditions** – Par symétrie, si certaines conditions particulières doivent être réunies après l'exécution du cas, elles sont à exprimer à ce niveau.

• **Scénario nominal** – Il s'agit là du scénario principal qui doit se dérouler sans incident et qui permet d'aboutir au résultat souhaité.

• **Scénarios alternatifs** – Les autres scénarios, secondaires ou correspondant à la résolution d'anomalies, sont à décrire à ce niveau. Le lien avec le scénario principal se fait à l'aide d'une numérotation hiérarchisée (1.1a, 1.1b…) rappelant le numéro de l'action concernée [36].

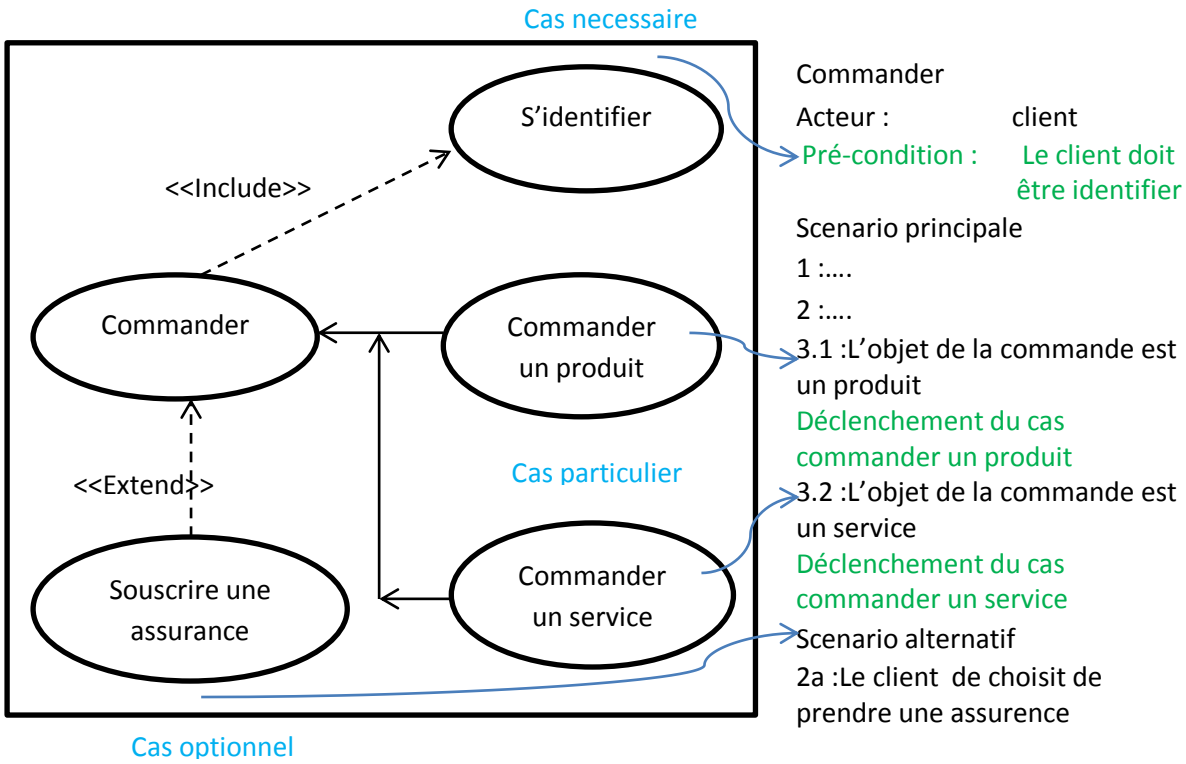


Fig10 – Description textuelle d'un cas d'utilisation [19].

3. Diagramme des activités

3.1. Introduction

Dans la phase de conception, les diagrammes d'activités sont particulièrement adaptés à la description des cas d'utilisation car ils viennent illustrer et consolider leur description textuelle. De plus, leur représentation sous la forme d'organigrammes les rend facilement compréhensible et permettent ainsi de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.

3.2. Activité (activity)

Une activité définit un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions.

Elle est représentée comme un rectangle à coins arrondis enfermant toutes les actions, les flux de contrôle et d'autres éléments qui composent l'activité [32].

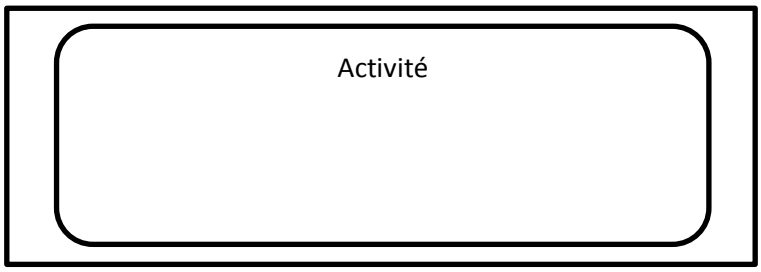


Fig11 – Forme d'une activité

4. Les éléments d'un diagramme d'activité

4.1. Action (action)

Une action représente un pas (une étape) seul (simple) dans une activité [37].

Une action représente l'exécution d'un traitement atomique, c'est à dire non interruptible. Cette exécution se traduit par un changement d'état du système ou le retour d'une valeur. Les actions correspondent à l'appel d'une opération, l'envoi d'un signal, la création ou la destruction d'un objet ou encore l'évaluation d'une expression. Il est dénotées par des rectangles ronds-coincés.

4.2. Nœud d'activité (activity node)

Un nœud d'activité est un type d'élément abstrait permettant de représenter les étapes le long du flot d'une activité. On observe trois types de nœuds d'activité [18]:

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

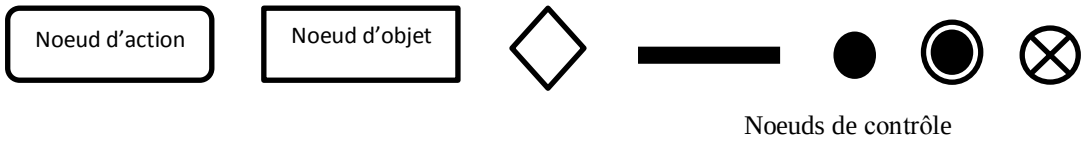


Fig12 – Forme du nœud d'activité : nœud d'action, nœud d'objet, nœud de décision (fusion), nœud de bifurcation (union), nœud initiale, nœud finale et nœud de flot [18]

4.3. Transition

Une transition est le passage du flux de contrôle d'une activité ou une action à une autre activité (ou action). Une activité ou une action a au moins une transition de sortie correspondant à la fin de son exécution.

Les transitions entre activités ou actions peuvent être déclenchées selon certaines conditions. Elles sont déclenchées dès que l'activité source est terminée et provoquent automatiquement et immédiatement le début de la prochaine activité à déclencher (l'activité cible) [17].

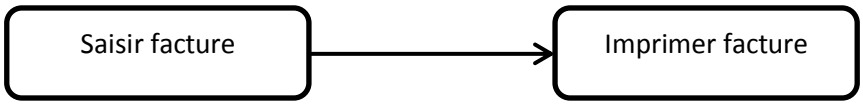


Fig13 – Représentation graphique d'une transition [17]

4.4. Nœud exécutable ou Nœud d'action (exécutable node)

Un nœud exécutable est un nœud d'activité qui donne lieu à une exécution d'actions. On trouve également le nœud d'action qui est un nœud d'activité exécutable qui constitue l'unité fondamentale de fonctionnalité exécutable dans une activité. Lorsqu'une action est exécutée, une transformation ou un calcul quelconque sont mis

en place dans le système modélisé. En général, les actions sont liées à des opérations qui sont directement appelées.

4.5. Nœud de contrôle (control node)

Un nœud de contrôle est un nœud d'activité abstrait utilisé pour coordonner les flots entre les nœuds d'une activité [18].

Il existe plusieurs types de nœuds de contrôle :

- nœud initial (initial node en anglais) ;
- nœud de fin d'activité (final node en anglais)
- nœud de fin de flot (flow final en anglais) ;
- nœud de décision (decisionnode en anglais) ;
- nœud de fusion (mergenode en anglais) ;
- nœud de bifurcation (forknode en anglais) ;
- nœud d'union (joinnode en anglais).

Les figures **Fig [12,14]** illustrent l'utilisation de ces nœuds de contrôle.

4.5.1. Nœud initiale

C'est un nœud de contrôle à partir duquel le flot débute lorsque l'activité enveloppante est invoquée. Une activité peut avoir plusieurs nœuds initiaux et un nœud à un arc sortant et pas d'arc entrant. Il est représenté par un petit cercle plein [18]

4.5.2. Nœud finale

Un nœud final est un nœud de contrôle possédant un ou plusieurs arcs entrants et aucun arc sortant. Il y a deux types de nœuds finaux: nœud de fin d'activité et nœud de fin de flot [18].

Lorsque l'un des arcs d'un nœud de fin d'activité est activé, l'exécution de l'activité enveloppante s'arrête et tous les nœuds ou flots actifs appartenant de cette activité est abandonné. Si l'activité a été appelée par un appel synchrone, un message **Reply** qui contient les valeurs sortantes est transféré en retour à l'appelant. Un nœud de fin d'activité est représenté par un cercle contenant un petit cercle plein

4.5.3. Nœud de décision

C'est un nœud de contrôle qui permet de faire un choix entre plusieurs flots sortants. Il possède un arc entrant et plusieurs arcs sortants, accompagnés de conditions de garde pour conditionner le choix [17]. Quand le nœud de décision est atteint et qu'aucun arc en aval n'est franchissable (ce qui veut dire qu'aucune condition n'est vraie), cela signifie que le modèle est mal formé. C'est pour cela que l'utilisation d'une garde (else) est recommandée après un nœud de décision, car elle garantit un modèle bien formé. En effet, la condition de garde est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses. Lorsque plusieurs arcs sont franchissables (c'est-à-dire que plusieurs conditions de garde sont vraies), l'un d'entre eux est retenu et ce choix est non déterministe. Un nœud de décision est représenté par un losange.

4.5.4. Nœud de fusion

Un nœud de fusion est un nœud de contrôle rassemblant plusieurs flots alternatifs entrants en un seul flot sortant. On ne l'utilise pas pour synchroniser des flots concurrents mais pour accepter un flot parmi plusieurs [18]. Un nœud de fusion est représenté par un losange comme le nœud de décision.

Graphiquement, il est possible de fusionner un nœud de fusion et un nœud de décision, c'est-à-dire de posséder plusieurs arcs entrants et sortants. Il est aussi possible de fusionner un nœud de décision ou de fusion avec un autre nœud. Mais pour mieux mettre en évidence un branchement conditionnel, il est préférable d'utiliser un nœud de décision

4.5.5. Nœud de bifurcation

Un nœud de bifurcation est un nœud de contrôle qui sépare un flot ou plusieurs flots concurrents. Il possède donc un arc entrant ou plusieurs arcs sortants [17]. Il est souvent accordé avec un nœud d'union section ([4.5.6]) pour équilibrer la concurrence

4.5.6. Nœud d'union

C'est un nœud de contrôle qui synchronise des flots multiples. Il possède plusieurs arcs entrants et un seul arc sortant. Lorsque tous les arcs entrants sont activés, l'arc sortant l'est aussi également [18]. Un nœud d'union est aussi représenté par un trait plein épais Enfin il est possible de fusionner un nœud de bifurcation et un nœud d'union, possédant donc plusieurs arcs entrants et sortants.

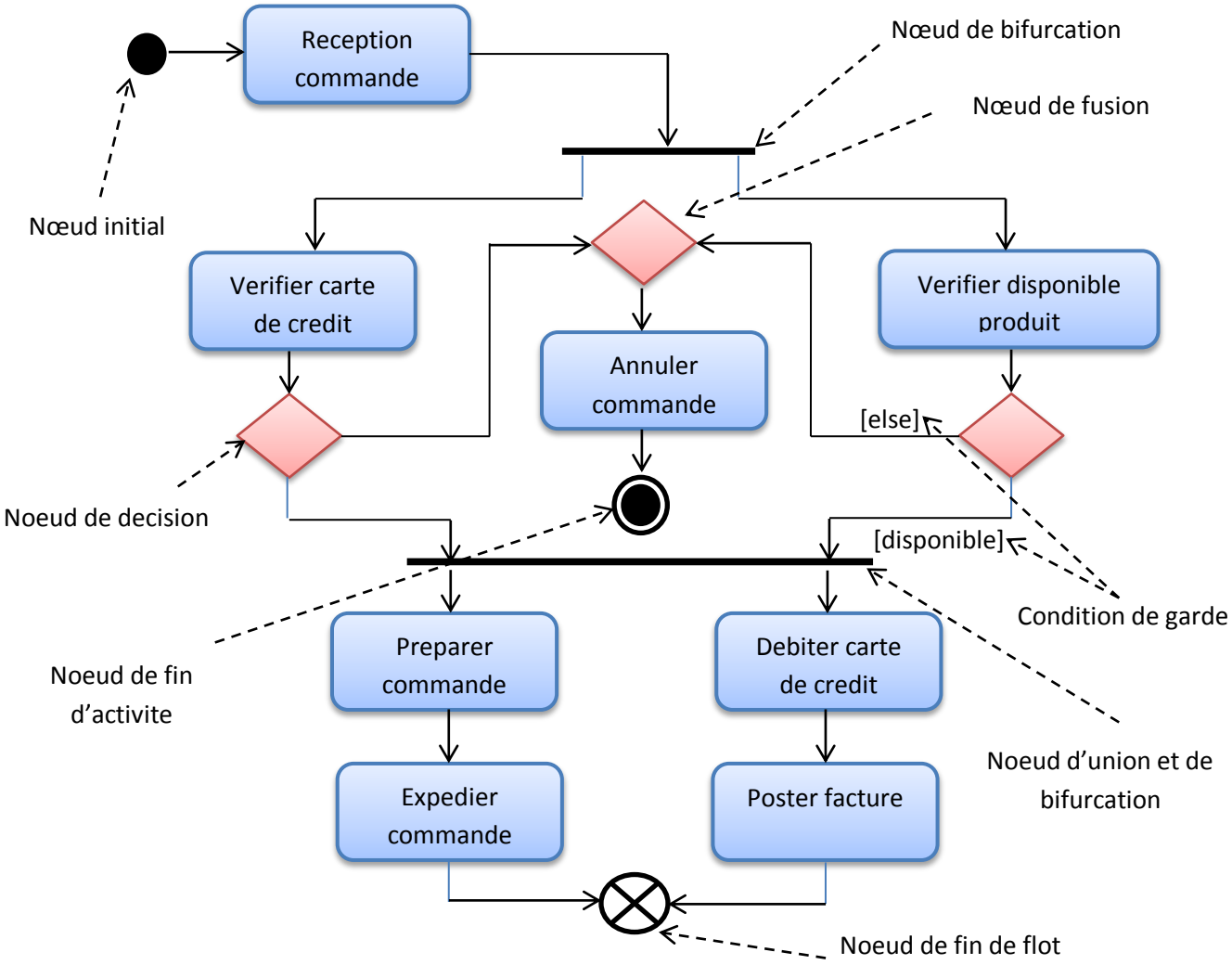


Fig14 – Exemple de diagramme d'activité illustrant l'utilisation de nœuds de contrôle. Ce diagramme décrit la prise en compte d'une commande [17].

5. Nœud d'objet (object node)

5.1. Introduction

Précédemment, nous avons expliqué la modélisation des flots de contrôle. Dans cette partie, nous allons aborder les flots de données, essentiels pour les traitements. Un nœud d'objet permet de définir un flot de données dans un diagramme d'activités.

5.2. Pin d'entrée ou de sortie

Pour exprimer les valeurs passées en argument à une activité ainsi que les valeurs de retour, il faut utiliser des nœuds d'objet appelés pins d'entrée ou de sortie. L'activité ne peut commencer uniquement lorsqu'une valeur est affectée à chacun de ses pins de sortie [18]. Un pin est représenté par un petit carré attaché à la bordure d'une activité.

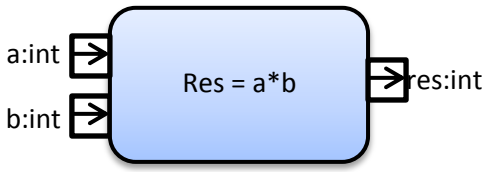


Fig15 – Représentation des pins d'entrée et de sortie sur une activité [18].

5.3. Pin de valeur (value pin)

Un pin de valeur est un pin d'entrée qui fournit une valeur à une action sans que cette valeur ne provienne d'un arc de flot d'objet. Il est toujours associé à une valeur spécifique. Graphiquement, il est représenté de la même manière qu'un pin d'entrée avec la valeur associée écrite à proximité [18].

5.4. Flot d'objet

Il permet de passer des données d'une activité à une autre. Un arc reliant un pin de sortie à un pin d'entrée est un flot d'objet. Dans cette situation, le type de pin récepteur doit être identique ou parent du type du pin émetteur [18].

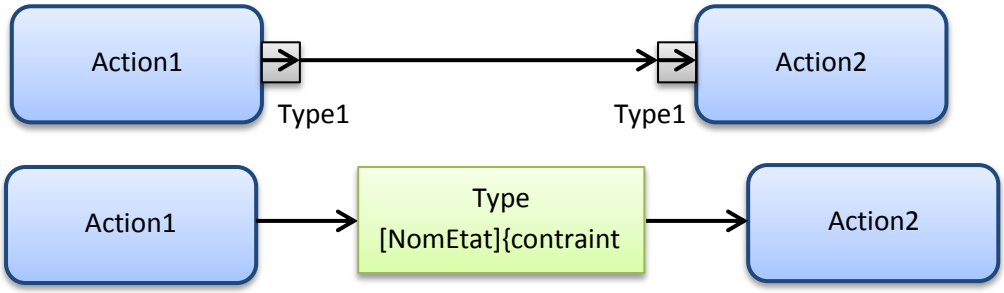


Fig16 – Deux notations possibles pour modéliser un flot de données [18]

5.5. Nœud tampon centrale (central buffer node)

C'est un nœud d'objet qui accepte les entrées de plusieurs nœuds d'objet ou produit des sorties vers plusieurs nœuds d'objet. Les flots en provenance d'un nœud tampon central ne sont donc pas directement connectés à des actions. Ce nœud modélise un tampon traditionnel qui peut contenir des valeurs venant de diverses sources et livrer des valeurs vers différentes destinations [18].

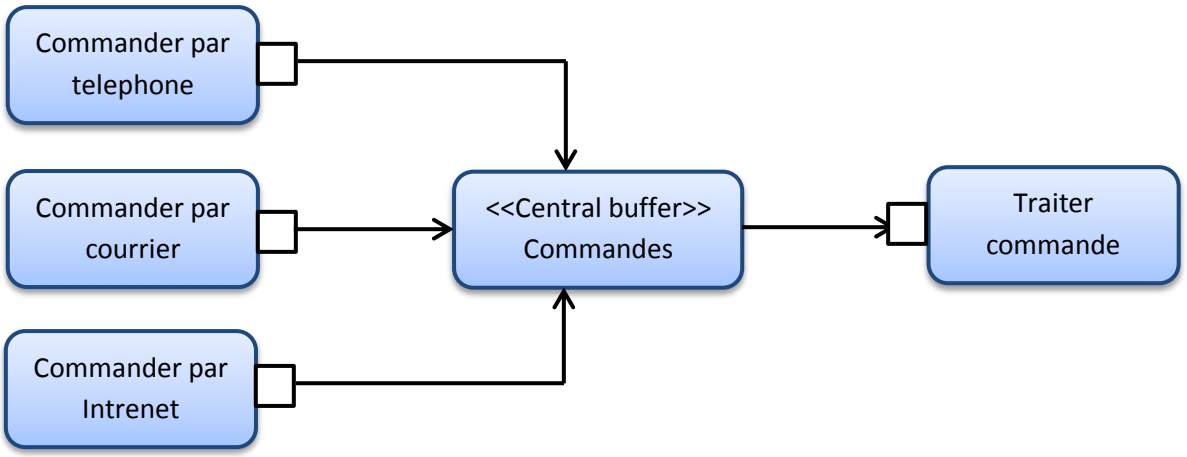


Fig17 – Exemple d'utilisation d'un nœud tampon central pour centraliser toutes les commandes prises par différents procédés, avant qu'elles soient traitées [18].

6. Partitions

Les partitions sont aussi des éléments essentiels dans un diagramme d'activité. En effet, elles sont appelées couloirs ou lignes d'eau. Elles ont pour but d'organiser les nœuds d'activités disposés dans un diagramme d'activité par le biais de regroupements. Ce sont des unités d'organisation du modèle. Ces partitions sont utiles lorsqu'on doit désigner la classe responsable qui rassemble un ensemble de tâches par exemple. La classe en question est donc responsable du comportement des nœuds à l'intérieur de la partition précédemment évoquée. Graphiquement, les partitions sont représentées par des lignes continues. Elles peuvent prendre la forme d'un tableau. De plus, les nœuds d'activités doivent appartenir à une seule et unique partition et les transitions peuvent passer à travers les frontières des partitions.

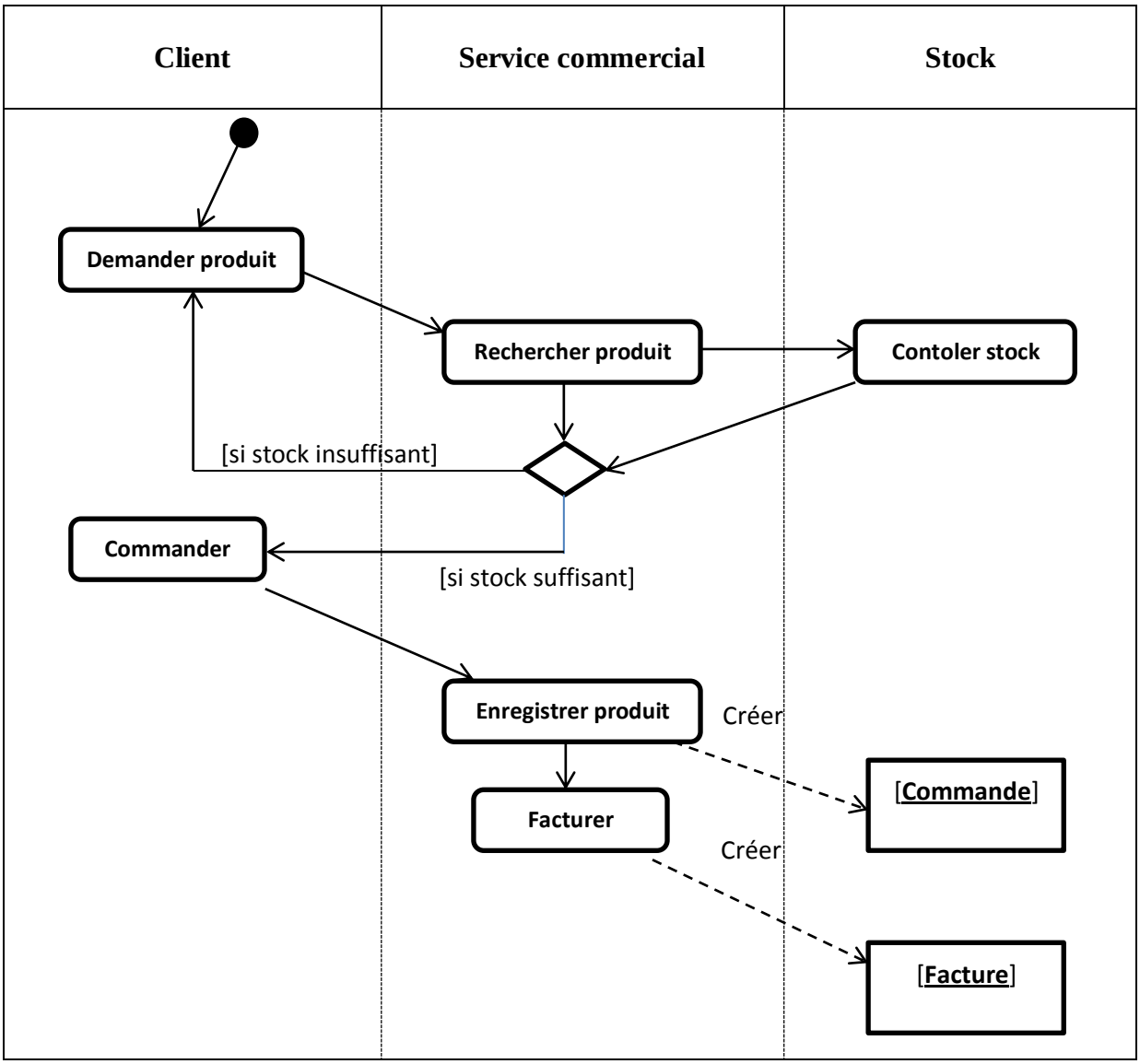


Fig18 - Exemple de diagramme d'activité avec couloir d'activité [17]

7. Exceptions

Une exception est générée quand une situation anormale entrave le déroulement nominal d'une tâche [18].

Elle peut être générée automatiquement pour signaler une erreur d'exécution (débordement d'indice de tableau, division par zéro, . . .), ou être soulevée explicitement par une action (RaiseException) pour signaler une situation problématique qui n'est pas prise en charge par la séquence de traitement normale.

Graphiquement, on peut représenter le fait qu'une activité peut soulever une exception comme un pin de sortie orné d'un petit triangle et en précisant le type de l'exception à proximité du pin de sortie

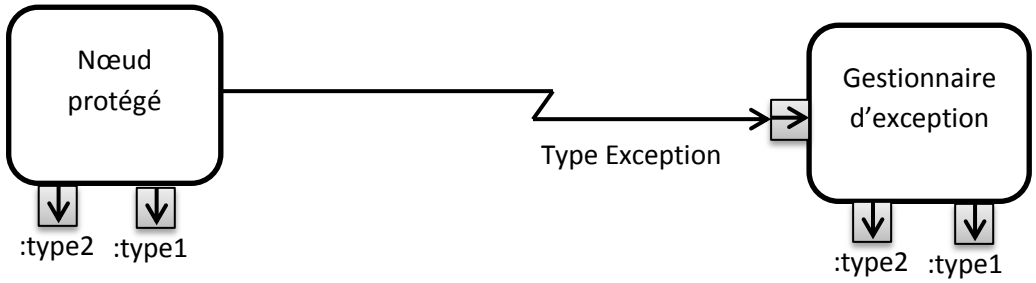


Fig19 – Notation graphique de la connexion entre une activité protégée et son gestionnaire d'exception associé [18].

8. Conclusion

Dans ce chapitre, nous avons fait deux diagrammes comportementaux pour recueil des besoins et d'écriture des activités, ces phases sont longues et fastidieuses Mais quand elles sont bien menées, elles permettent de lever toutes les ambiguïtés et de recueillir les besoins dans leurs moindres détails.

L'objectif de ces phases de la modélisation est donc de clairement identifier les frontières du système et donne la conception de l'architecture interne du système « par activités » qu'il doit offrir à l'utilisateur.

Partie 2

La vérification de la cohérence dans UML

(UML consistencychecking)

1. Introduction :

UML (Unified Modeling Language) est actuellement le standard pour la modélisation des systèmes orientés objets, Il définit différents points de vue d'une modélisation en la s'expriment à travers de treize types de diagrammes : structurel (statique) et comportemental (dynamique). Par ces points de vue il est nécessaire d'assurer la cohérence de l'élément de modélisation par rapport à la norme, par rapport à son contexte et par rapport à une démarche d'analyse et de conception.

Avoir des vues différentes permet d'assurer une transition semi-automatique vers l'étape de mise en œuvre qui s'apparente à un processus de transformation de modèle. Mais pouvant introduire des redondances, des imprécisions et des incomplétudes.

Le diagramme de classes de la figure **[Fig20]** montre que toute personne possède au plus deux parents.

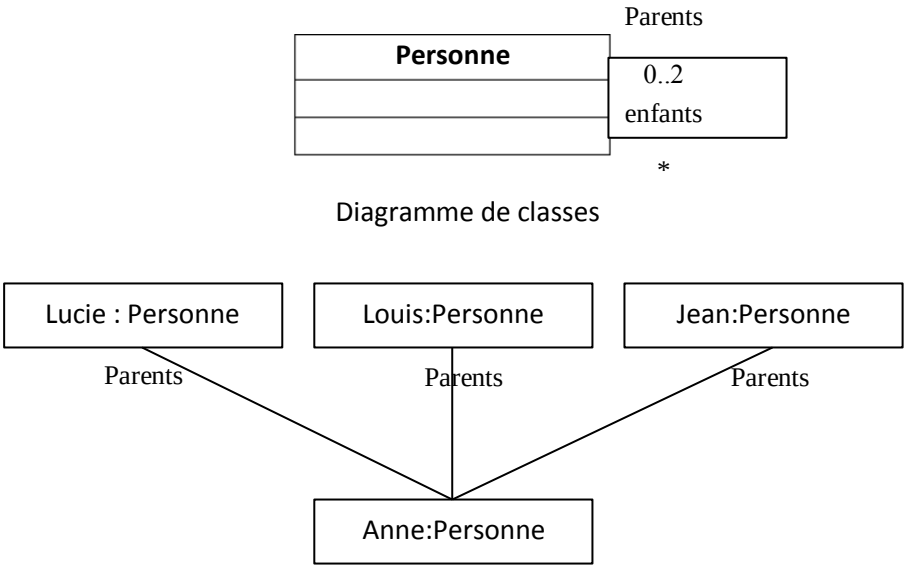


Fig20 - Exemple de modélisation [18].

Pour l'instanciation proposée, le diagramme d'objets met en évidence une incohérence par rapport au diagramme de classes.

Dans le contexte de la génération de code, cette incohérence se répercute sur le Plan de tests. Au pire, elle sera détectée lors des tests du code généré tel que le montre les deux fragments de code Java suivants [35] :

```
public class Personne
{
//Mise en oeuvre de l'association réflexive
Private vector parents ;
public voidadParent(Personne p) throws Exception
{
//Contrainte issue de la multiplicité
if (parents.size() == 2)
throw new Exception("Violation de contrainte");
…
}
}
```

La modification de code rendre ce test valide mais introduire une incohérence entre la mise en œuvre et la conception originelle.

2. Définition d'une vérification

Une vérification est définie par un nom, un but (commentaire informel), un espace de travail. Elle peut être réalisée à l'aide de règles (algorithmes, prédicats, code) ou de preuves dans un modèle formel. Une bonne pratique est de définir, si possible, les règles par des expressions OCL (*Object Constraint Language*) pour conserver un cadre homogène.

Dans une vérification implantée par un système à base de règles, l'utilisateur choisit les règles (listes, choix multiple) et lance la vérification. Le moteur prend ensuite les règles et les vérifie indépendamment sur le modèle en entrée. Cela implique une connaissance approfondie de la part de l'utilisateur des règles à vérifier et des

enchaînements raisonnables. On souhaite fournir une aide à la fois au concepteur de vérification et aux utilisateurs, via la notion de processus de vérification [38].

3. Vérification de la cohérence

On rencontre deux problèmes principaux de cohérence au cours de la construction du modèle:

- Inter-diagrammes.
- Intra-diagrammes (diagrammes d'un modèle).

4. Types d'incohérences

On distingue également la cohérence statique ou dynamique [20] :

- La cohérence statique (syntaxique) d'UML représente les propriétés que doit respecter un modèle UML sans l'exécuter. Elle est d´écrite formellement par le Meta modèle et des Contraintes OCL, ainsi que de manière textuelle. Elle peut être vérifiée entre autres par une inspection statique du modèle. Par exemple: <<**Une contrainte attachée à un stéréotype ne doit pas rentrer en conflit avec d'aucun stéréotype hérite, ou avec aucune Contrainte de la classe de base**>>.
- La cohérence (sémantique) dynamique représente les propriétés à respecter qui seront vérifiées à l'exécution. UML n´étant pas exécutable, ces contraintes dynamiques doivent être embarquées dans un formalisme exécutable (comme le code source). Par exemple **: <<il n'est pas possible de vérifier statiquement qu'une pré-condition d'une opération est satisfaite avant que l'opération n'ait été appelée dans un diagramme de cas d'utilisation** >>.

Nous nous intéressons aussi bien aux cohérences dynamiques que statiques.

Notre étude cherche donc à identifier toutes les propriétés sur les modèles qui sont dépendantes des fonctionnalités du système modélisé et exprimés dans la norme OCL.

5. Notion d'UML

5.1. Méta-modèle UML

Le méta-modèle UML [21,22] décrit la sémantique statique (partielle) du langage UML. La cohérence syntaxique est assurée par ce méta-modèle, dite encore la syntaxe

abstraite de UML. Ce méta-modèle donne une structure arborescente en utilisant les concepts de :

- classes, pour définir la structure des nœuds,
- attributs, pour définir les propriétés attachées à chaque nœud et,
- associations, pour représenter les connexions entre ces nœuds.

Le méta-modèle présente des relations de dépendances explicites entre les éléments structurels par exemple « chaque arc de transition doit avoir un élément ».

5.2. Langage de Contraints Objets (OCL)

OCL (langage de contraints objets, *object constraint language en anglais*) est un langage fonctionnel Intégré a UML basé sur la logique du premier ordre permettant d'exprimer des spécifications sur des modèles et des méta-modèles UML. UML sous la forme de diagrammes de classes, OCL peut être utilisé comme langage de spécification du méta-modèle, ce qui nécessite une bonne connaissance de la syntaxe et de la sémantique des éléments de modélisation. Des règles OCL de bonne formation (wellformedness rules) peuvent être adjointes à ce méta-modèle pour restreindre l'ensemble des instances valides.

Notre étude exploite ce langage pour vérifier des méta-modèles sans introduire un nouveau formalisme [35].

6. Notions d'incohérences

Aujourd'hui, de nombreuses classes d'incohérence existent dans UML [20], c'est-à-dire plusieurs façons de créer des incohérences dans les spécifications réalisées avec UML. Dans cette section, on essaiera de présenter : les types d'incohérences, la localisation de l'incohérence, l'analyse et les conséquences de l'incohérence dans les diagrammes UML (Diagrammes des cas d'utilisations et Diagrammes d'activités).

6.1. Incohérences comportementales dans les diagrammes des cas d'utilisations

Les incohérences comportementales peuvent être détectées en analysant la structure du modèle des spécifications élaborées par le concepteur, nous citons principalement

6.1.1. Acteur

Un acteur (Actor en anglais) représente un ensemble cohérent de rôles que peuvent jouer des utilisateurs d'un système lorsqu'ils interagissent avec les cas d'utilisations. Un acteur ne peut avoir des associations qu'avec des cas d'utilisations

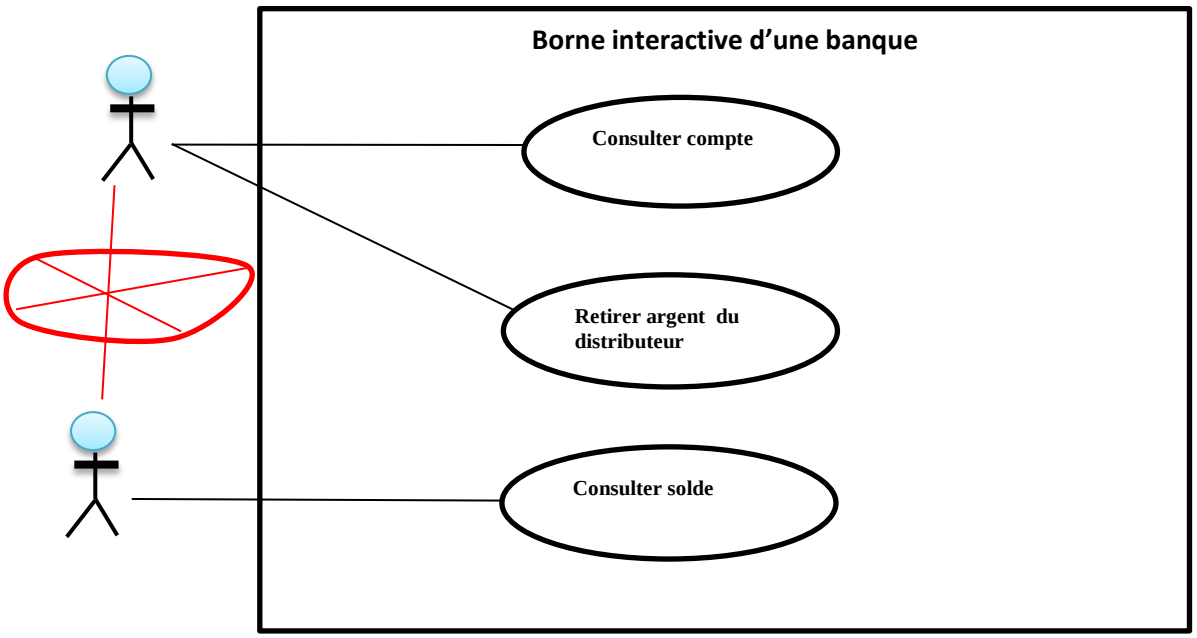


Fig21 – Exemple de diagramme de cas d'utilisation

6.1.2. Cas d'utilisation

Un cas d'utilisation (UseCase en anglais) est un classificateur qui représente une unité cohérente de fonctionnalités fournies par un système, un sous-système ou une classe. Un cas d'utilisation ne doit pas avoir d'associations avec d'autres cas d'utilisation qui spécifient le même système.

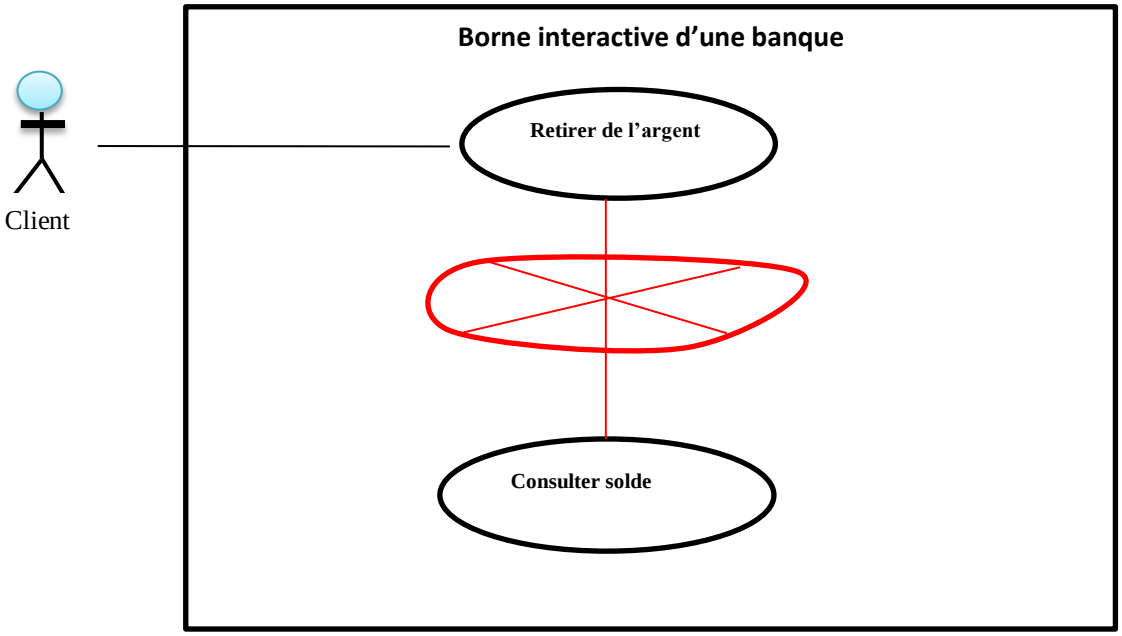


Fig22 – Exemple de diagramme de cas d'utilisation

6.1.3. Extend

Une relation d'extension (extend relationship en anglais) met en relation un cas d'utilisation < étendant > et un cas d'utilisation étendu et spécifie comment et quand le comportement du cas d'utilisation < étendant > peut être insère dans le comportement du cas d'utilisation étendu.

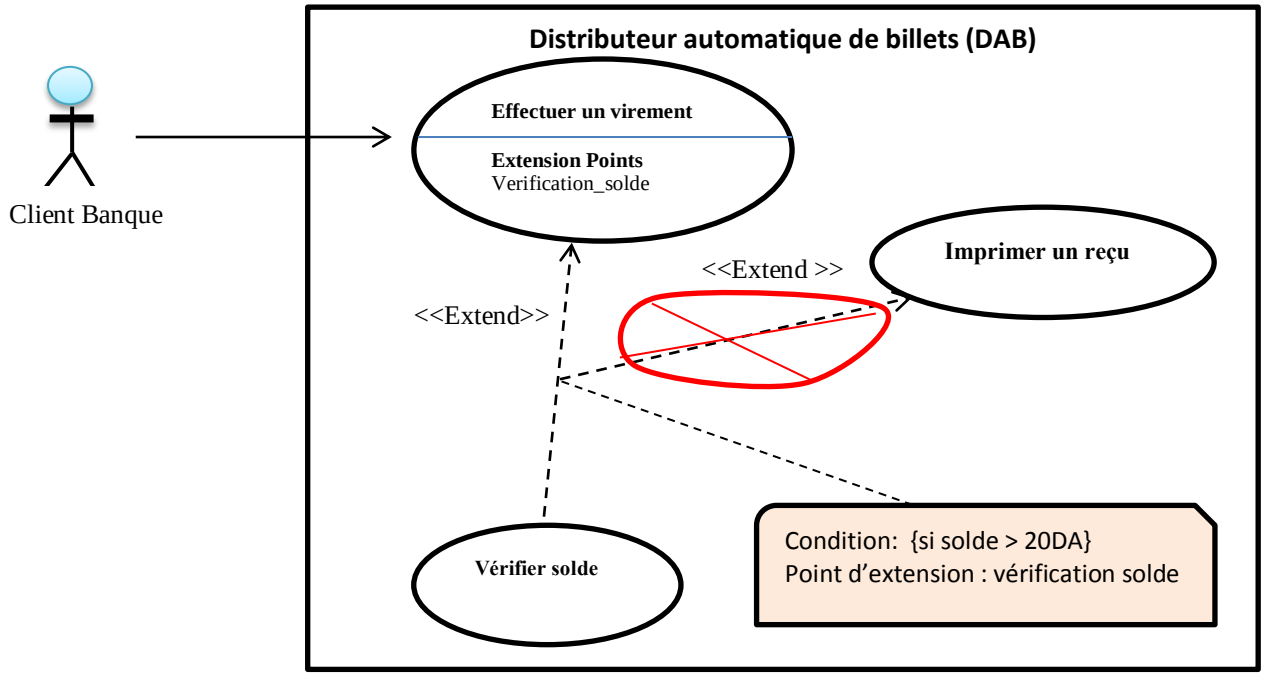


Fig23 – Exemple de diagramme de cas d'utilisation relation d'extension [17].

6.1.4. Include

Une relation d'inclusion (include relationship en anglais) entre deux cas d'utilisation implique que le comportement du cas d'utilisation qui est inclus est inséré dans le comportement du cas d'utilisation < incluant >. Une relation d'inclusion a comme source un (et un seul) cas d'utilisation

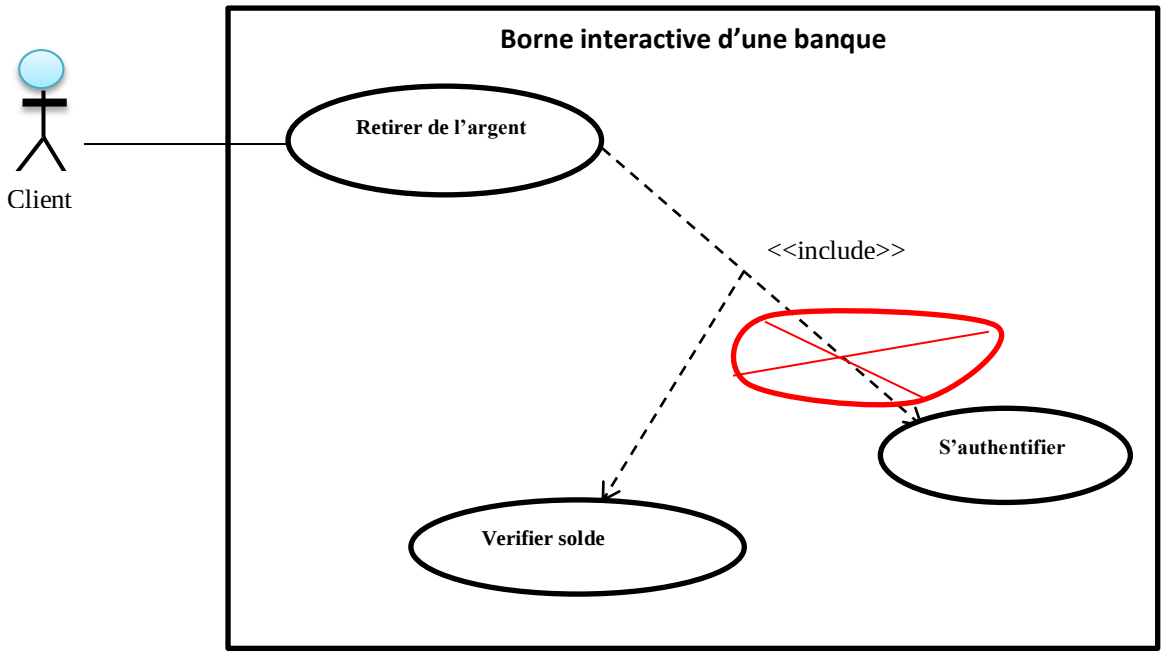


Fig24 – Exemple de diagramme de cas d'utilisation [17]

6.1.5. Diagramme de cas d'utilisation

Les cas d'utilisations sont un moyen de spécifier les usages attendus d'un système. Typiquement ils sont utilisés pour cerner les exigences fonctionnelles d'un système, c'est-`a-dire les fonctionnalités que le système doit remplir. Les concepts clés d'un tel diagramme sont les cas d'utilisation, les acteurs et les sujets (subject en anglais). Le sujet est le système considère sur lequel les cas d'utilisations s'appliquent. Les utilisateurs et autres systèmes qui interagissent avec le sujet sont représentés comme des acteurs.

Les diagrammes des cas d'utilisations sont une spécialisation des diagrammes de

Classes dans lesquels les seuls classificateurs qui apparaissent sont soit des acteurs soit des cas d'utilisations.

Les nœuds graphiques qui peuvent apparaitre dans un diagramme des cas d'utilisations sont

- des acteurs ;
- des points d'extension ;
- des cas d'utilisation.

6.2. Incohérences comportementales dans les diagrammes d'activité :

Les incohérences comportementales peuvent être détectées en analysant la structure du modèle de spécification élaboré par le concepteur, nous citons principalement

6.2.1. Nœud initial : un nœud initial ne doit pas un arc entrant

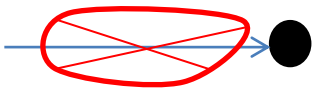


Fig25 – Exemple nœud initiale incohérent

6.2.2. Nœud finale : un nœud final ne doit pas un arc sortant

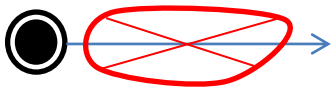


Fig26 – Exemple nœud finale incohérent

6.2.3. Nœud de décision : un nœud de décision n'a qu'un seul arc entrant

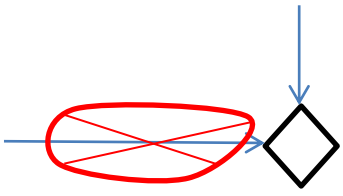


Fig27 – Exemple nœud de décision incohérent

6.2.4. Action : Une activité n'a pas une action d'entrée vide.

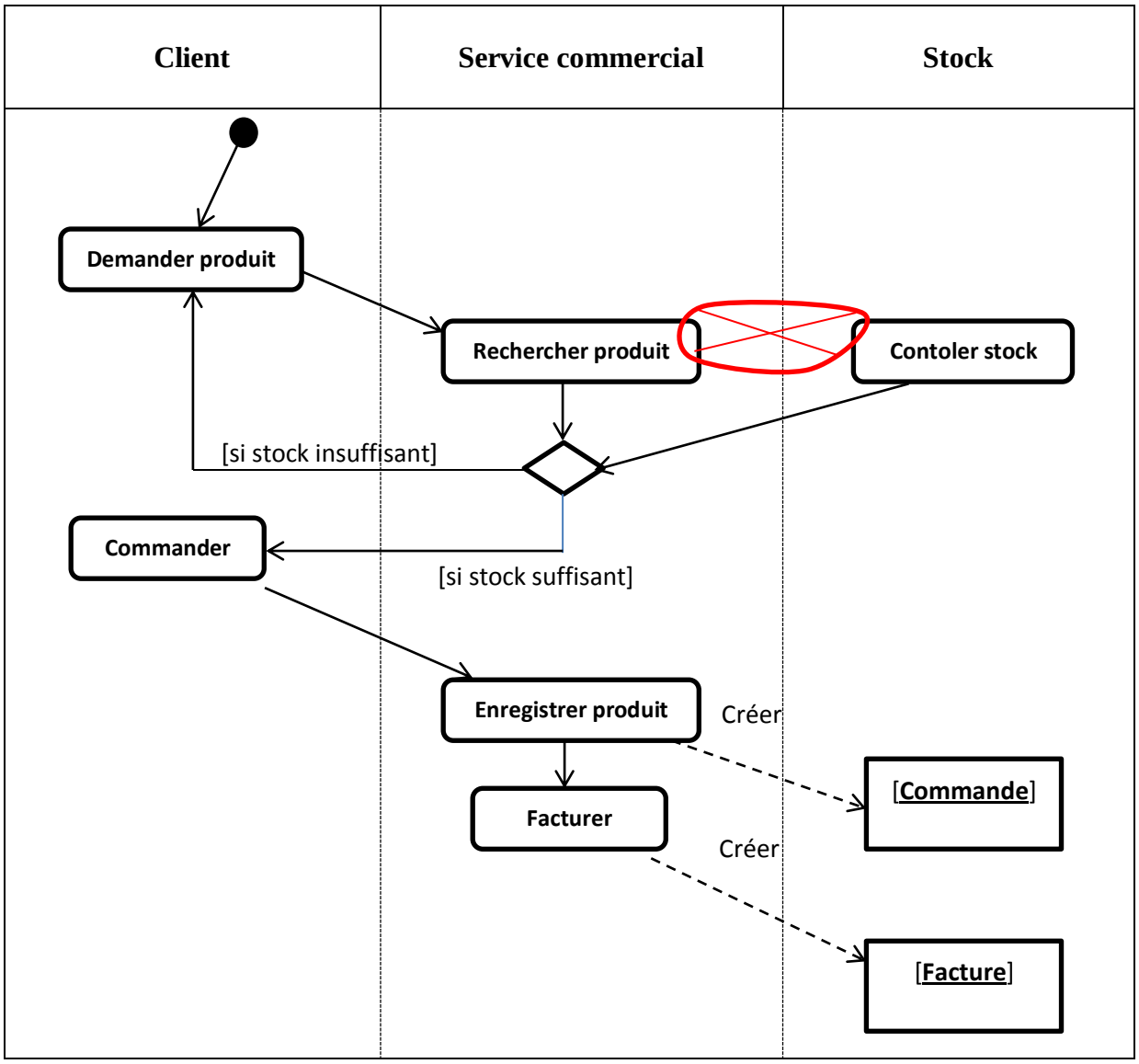


Fig28 – Exemple action incohérent [17].

- L'action d'entrée d'un état d'appel est une action d'appel unique

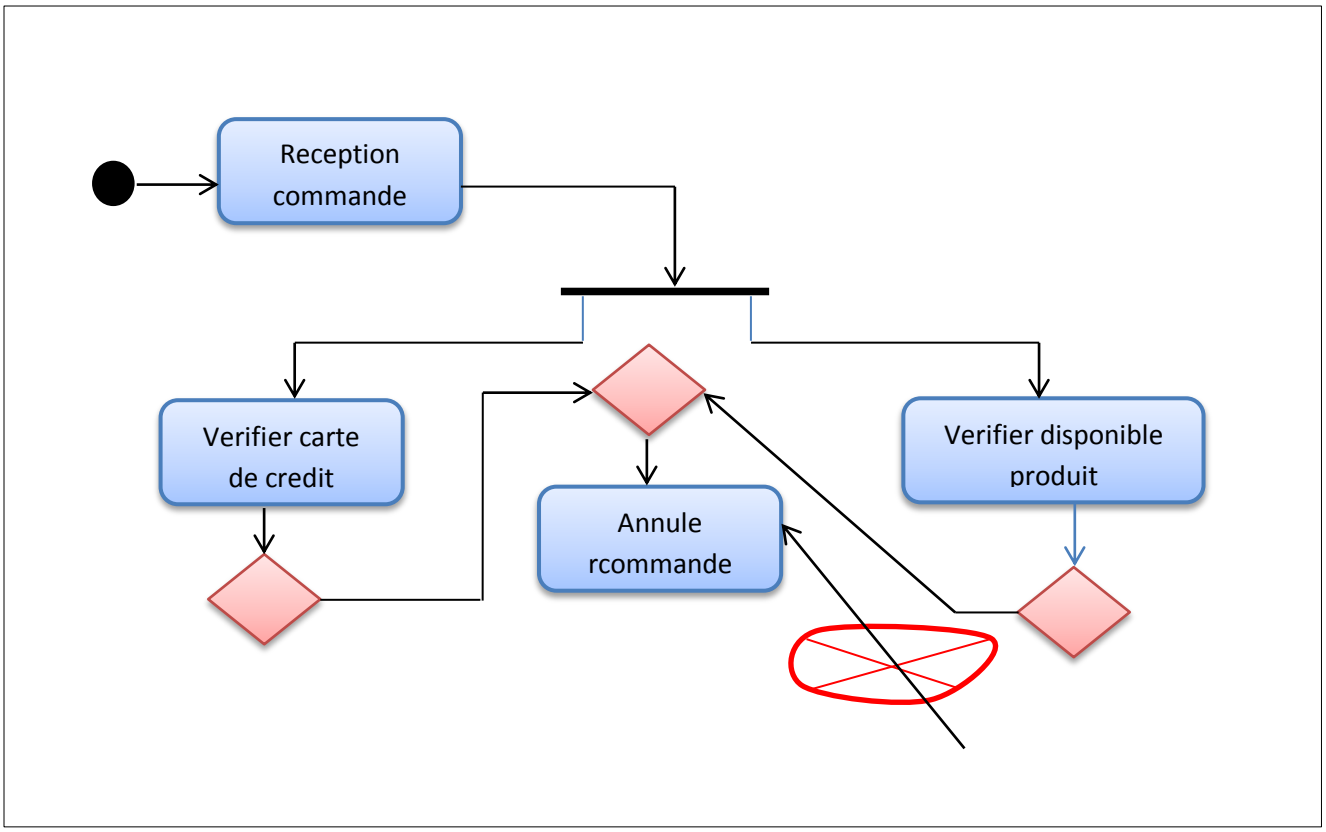


Fig29 – Exemple action incohérent

6.3. Incohérences dans les diagrammes comportementaux :

Les incohérences sémantiques détectées entre les diagrammes des cas d'utilisations et les diagrammes d'activités.

- Chaque cas d'utilisation est décrit par au moins un diagramme d'activité
  Ce diagramme doit être composé à au moins un nœud initial, un nœud final et une activité nœud

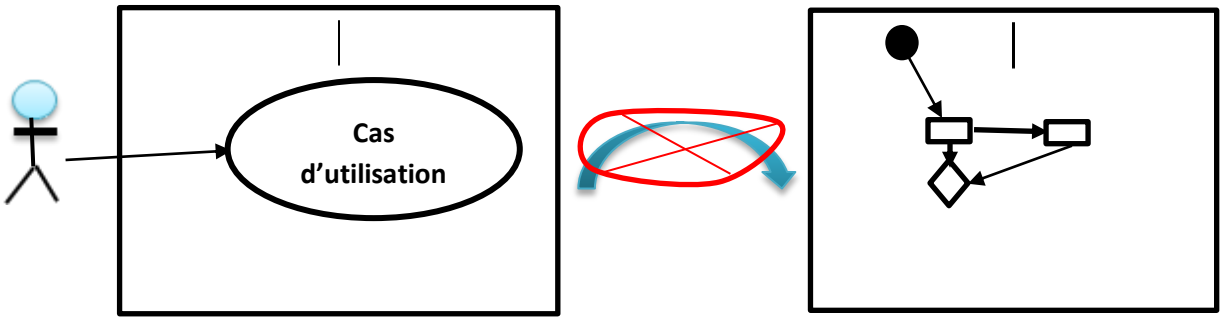


Fig.29.1- Diagramme de cas d'utilisation et son correspondant diagramme d'activité

- Etat d'incohérence si la première action effectuée par un acteur ne correspondant pas la première activité dans le diagramme d'activité

7. Analyse de la cohérence :

Lorsque la vérification de la cohérence a constaté qu'une spécification n'est pas cohérente, elle contiendra un certain nombre d'incohérences. Les incohérences sont liées à un certain nombre d'éléments de modèle, c'est-à-dire à sa portée, qui sont la source du comportement non correspondant. On s'attend à ce que ces éléments de modèle représentent le même comportement, mais d'une manière ou d'une autre.

Le développeur peut choisir de prendre des mesures et doit savoir quelles conséquences les différentes actions auront sur ces incohérences. Cette analyse devra expliquer ces actions et prévoir les conséquences et donner au développeur les informations nécessaires pour choisir de prendre des mesures sur l'incohérence ou non.

Dans [30], ils estiment que pour déterminer les mesures à prendre pour faire face à une incohérence sont déterminée en analysant les conséquences que chacun modifie l'action native sur la spécificité originale.

8. Risques de la cohérence :

Les incohérences ça peut reposent sur la sémantique de vérification des constructions du langage UML, c'est-`a-dire sur la bonne façon de constituer un modèle UML.

Une incohérence peut être vue comme la conséquence de constructions redondantes ou complémentaires incompatibles entre elles

Les fautes présentes dans les logiciels peuvent certainement être induites par des erreurs survenant lors du processus de conception, elles peuvent être également attachées à la technologie logicielle employée.

Les risques liés aux technologies logicielles qui nous intéressent sont les risques dont les dommages sont les fautes dans les programmes et dans les méta-modèles.

9. Approche de vérification formelle d'un modèle UML

L'originalité de notre approche est le fait d'exploiter les relations **explicites** et **implicites** présentes entre les diagrammes d'un modèle UML [23]. Les relations explicites sont dégagées à partir de la syntaxe d'UML, comme définie dans son méta-modèle. Les relations **implicites** représentent les chevauchements sémantiques

présents entre les diagrammes d'un modèle. Par exemple, il existe une relation implicite entre le diagramme de cas d'utilisations et le diagramme de classes. Il est à noter que, pour notre démarche, nous avons identifié ces relations implicites à partir du processus de développement.

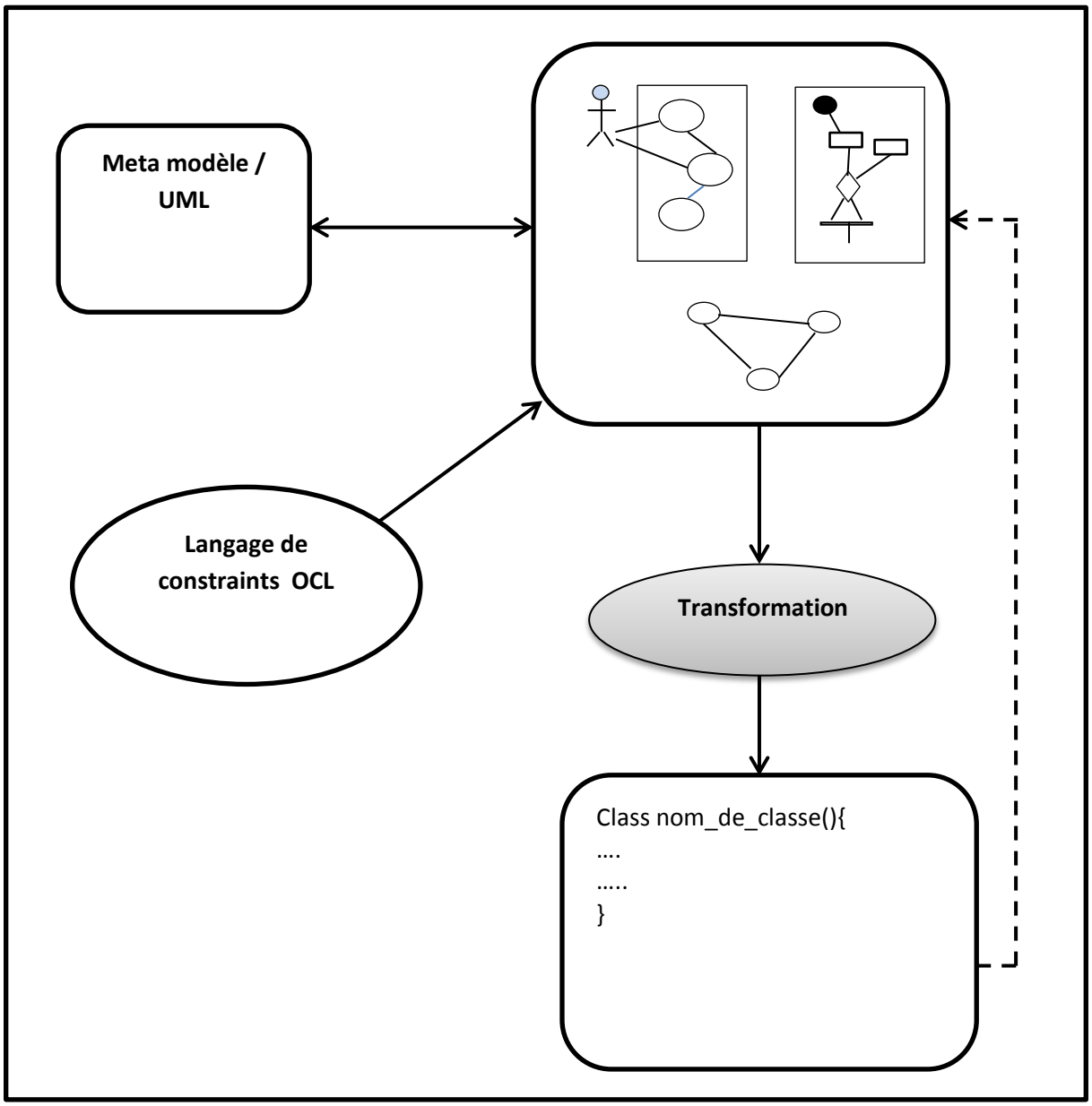


Fig30 - Une vue simplifiée du processus de vérification formelle des modèles UML [24].

10. Techniques de vérification de la cohérence :

La vérification de modèle UML se fait en général selon l'une des deux approches suivantes : les méthodes formelles et les systèmes à base de règles.

Dans l'approche par méthodes formelles, l'objectif est de définir un modèle formel sur lequel on peut ensuite exprimer et vérifier des propriétés.

Dans l'approche des systèmes à base de règles, une règle décrit un moyen de vérifier une (partie d'une) propriété. La description peut être formelle (OCL, algorithme, code Java...) ou informelle.

Chapitre 3

Le langage OCL comme moyen de vérification pour le langage UML

1. Intérêt d'un Langage de Contraintes Objet (OCL)

1.1. Introduction :

C'est avec OCL (Object Constraint Language) qu'UML formalise l'expression des contraintes. Il s'agit donc d'un langage formel d'expression de contraintes bien adapté aux diagrammes d'UML.

OCL peut s'appliquer sur la plupart des diagrammes d'UML et permet de spécifier des contraintes sur l'état d'un objet ou d'un ensemble d'objets comme des invariants sur des classes ; des pré-conditions et des post-conditions à l'exécution d'opérations :

- Les invariants sur les classes,
- les pré-conditions doivent être vérifiées avant l'exécution,
- les post-conditions doivent être vérifiées après l'exécution ;
- des ensembles d'objets destinataires pour un envoi de message ;
- des attributs dérivés, etc.[40]

1.2. Définition OCL :

- OCL : Object Constraint language.
- Langage formel pour exprimer les contraintes.
- Les expressions OCL sont évaluées, n'ont aucun effet de bord.
- Celui-ci s'applique non seulement au Meta-modèle mais aussi au modèle.

1.3. Pourquoi OCL ?

- Un diagramme UML peut manquer de précision pour exprimer tous les aspects d'une spécification [40]
    - Besoin de décrire des contraintes additionnelles au modèle.
    - Le langage naturel est souvent utilise (problème: apparition d'ambiguïtés).

- OCL a pour but de combler ce manque.
    - Langage formel : aucune ambiguïté
    - Pure langage d'expression : aucun effet de bord.
    - Ce n'est pas un langage de programmation : ne modifie pas le modèle.

1.4. Comment utiliser OCL?

L'Object Constraint Language (OCL) est un langage normalisé par l'OMG (Object Management Group) qui permet d'ajouter des contraintes aux objets d'un modèle **UML** en évitant les ambiguïtés du langage naturel.

Quand une expression OCL est évaluée, elle retourne une valeur sans rien changer dans le modèle. L'état du système n'est jamais modifié à cause de l'évaluation d'une expression OCL même si l'expression est utilisée pour spécifier le changement d'état dans une post condition.

OCL est utilisé :

- Comme un langage de requêtes
- Pour spécifier des invariants
- Pour décrire des pré et post conditions
- Pour décrire des conditions
- Pour spécifier des messages et des actions
- Pour spécifier des contraintes sur des opérations
- Pour spécifier les règles de déduction sur les attributs (attributs dérivés ou déduits) [40]

2. OCL : Comment l'écrire ?

2.1. De manière graphique

Fig.31 - diagramme de classe d'entreprise [17].

2.2. De manière textuelle :

Context
Expression OCL

Ex : **context** Company **inv** :

Self.numberOfEmployees> 50

3. Typologie des contraintes OCL

Mise en situation

Plaçons-nous dans le contexte d'une application d'entreprise selon le diagramme de classe suivant :

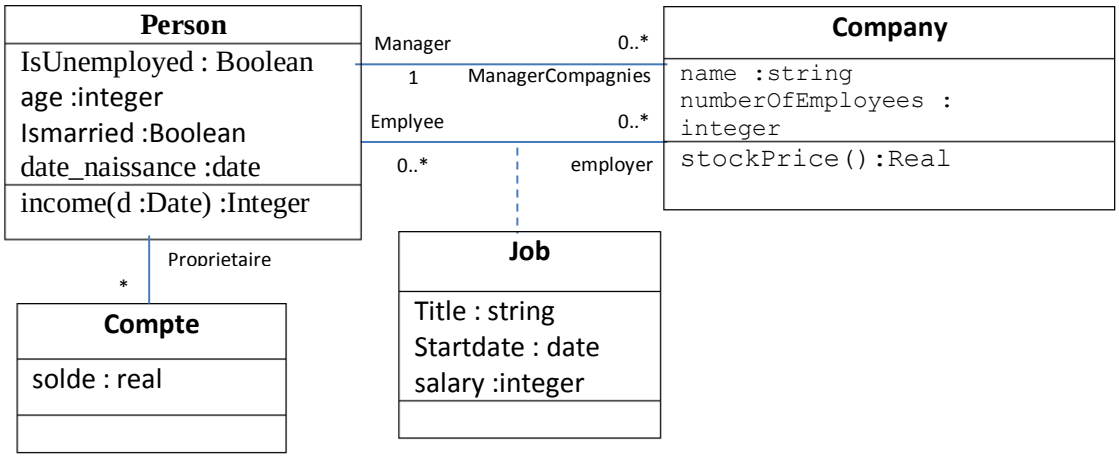


Fig32- diagramme de classe d'entreprise [22].

3.1. Contexte (context) :

Une contrainte est toujours associée à un élément de modèle. C'est cet élément qui constitue le contexte de la contrainte. Il existe deux manières pour spécifier le contexte d'une contrainte OCL.

Syntaxe :

Context entiteUML type De contexte

[nomDeLarRegle]:

Ex :

Context company **inv :**

3.2 Invariants (inv)

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

Syntaxe :

inv : <expression_logique>

<expression_logique> : est une expression logique qui doit toujours être vraie.

Ex :

context Compte inv :

Solde > 0 ----------Le solde d'un compte doit toujours être positif.

Remarque : Dans le cas d'in contexte invariant :

Syntaxe :

Context entiteUML **inv:**

Expression OCL

Ex :

Context company **inv :**

Self.numberOfEmployees> 50

3.3 Pré-conditions et post-conditions (pré, post) :

3.3.1. Dans le cas d'un contexte pré et post-conditions s'applique aux opérations :

Syntaxe :

Context typeName :: operationName(param1 :type1,….): ReturnType
Pre : param1>….
Post : result1=…

Ex:
Context person::income(date:Date) :Integer
Post : result = 5000

3.4. Résultat d'une méthode (body)

Ce type de contrainte permet de définir directement le résultat d'une opération.

Syntaxe:

Context entiteUML **inv:**

body : <requête>

Ex:
Context person::income(date:Date) :Integer
Body : solde

3.5. Définition d'attributs et de méthodes (def et let…in) :

Let : Permet de définir un attribut ou une opération dans une contrainte.

Syntaxe:

Context entiteUML **inv:**

let<déclaration>=<requête>**in**<expression>

Ex :

Context person **inv**

let income : Integer = self.job.salary→sum()

let hasTitle(t:string):Boolean=self.job→exists(title=t) **in**

if isUnemployed **then**

self**.income**< 100

else

self.**income**>= 100 and

self.**hasTitle**(‘manager’)

endif

def est un type de contrainte qui permet de déclarer et de définir la valeur d’attributs comme la séquence let…in.

def permet également de déclarer et de définir la valeur retournée par une opération interne à la contrainte.

Syntaxe :

Context entiteUML **inv:**

def :<déclaration>=<requête>

Ex :

context Person

def : argent : int = compte.solde ->sum()

context Person

inv : age>=18 implies argent>0

3.6 Initialisation (init) et évolution des attributs (derive)

Le type de contrainte **init** permet de préciser la valeur initiale d’un attribut ou d’une terminaison d’association.

Les diagrammes d’UML définissent parfois des attributs ou des associations **dérivées**. La valeur de tels éléments est toujours déterminée en fonctions d’autres éléments du diagramme. Le type de contrainte dérive permet de préciser comment la valeur de ce type d’élément évolue.

Syntaxe :

Context entiteUML **inv:**

init : <requête>

Context entiteUML **inv:**

derive : <requête>

Ex :

context Person::married : Boolean

init : false --attribut marie est initialisé par false.

context Person::employer : Set(Société)

init : Set{} --la personne ne possède pas des employeurs.

context Person::age : Integer

derive : date_de_naissance - Date::current() --retourne nombre d'année

4. Accès aux caractéristiques et aux objets dans les contraintes OCL

4.1. Accès aux attributs et aux opérations (self) :

Le mot clé self (même sémantique que le This du C++, Java ou C#) fait référence à une instance du contexte

1- Accès à un attribut :

Syntaxe :

Context entiteUML **inv:**

Self.attribut<expression>

2- Accès à une opération :

Syntaxe :

Context entiteUML **inv:**

Self.methode

Ex :contextperson

Self. income(aDate) > 10

3- Accès à une fin d'association :

Syntaxe :

Context entiteUML **inv:**

Self.nomDuRoleOppose

Ex :

Context company

Inv :self.manager.isUnemployed = false

Inv :self.employee->notEmpty()

Context Person **inv**:

Self.employer->isEmpty() --Ensemble de personnes qui ont le rôle employeur

4.2. Navigation vers la classe associations

Pour naviguer vers une classe association, il faut utiliser la notation pointée classique en précisant le nom de la classe association en minuscule.

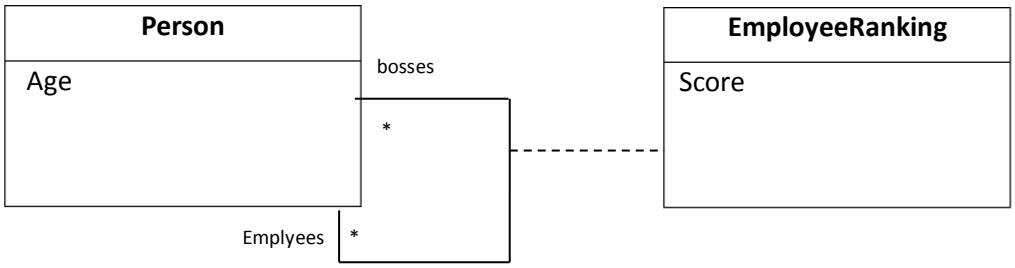


Fig33- diagramme de classe d'entreprise [22].

Context person **inv**

Self.employeeRanking[bosses]→sum()>0

→Ensemble des **employeeRanking** qui appartiennent à la collection **bosses**

4.3. Navigation à partir des classes associations

On peut naviguer à partir d'une classe association vers les objets qui participent à cette association. Ceci est réalisé en utilisant la notation par le point et les noms de rôles sur les extrémités des associations

Context Job

inv: self.employer.numberOfEmployees>= 1

inv: self.employee.age> 21

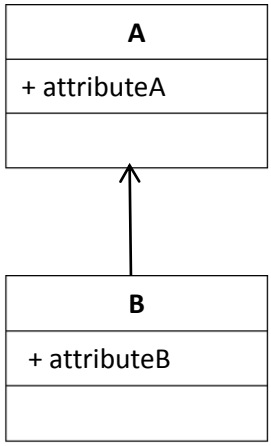
4.4. Accéder à une caractéristique redéfinie (oclAsType())

Accès aux propriétés des super types

Context B **inv** :

Self.**oclAsType(A).**attributeA

Self.attributeA



- Nous avons accès à la propriété définie dans la classe A.
- Ensuite, nous accédons à la propriété définie dans la classe B

Fig34- diagramme de classe

4.5. Opérations prédéfinies sur tous les objets

Il existe plusieurs caractéristiques qui s'appliquent à l'ensemble des objets et qui sont prédéfinis en OCL: [41]

- **oclIsTypeOf (t : OclType) : Boolean**
  retourne vrai si l'objet est du type t
- **oclIsKindOf (t : OclType) : Boolean**
  retourne vrai si l'objet est du type t ou d'un de ses sous-types
- **oclInState (s : OclState) : Boolean**
  retourne vrai si l'objet est dans l'état s
- **oclIsNew () : Boolean**
  l'objet est créé pendant l'opération
- **oclAsType (t : OclType) : instance of OclType**
  retournevrai si le type de **self** et **t** sont les mêmes.

Ex :

context Person

inv: self.oclIsTypeOf(Person) -- est vrai

inv: self.oclIsTypeOf(Company) -- est faux

5. Opérations sur les collections

5.1 Définition : «.», «->», «::» et self

«.» : permet d'accéder à une caractéristique (attributs, terminaisons d'associations, opérations) d'un objet

«->» : permet d'accéder à une caractéristique d'une collection

«::» permet de désigner un élément (comme une opération) dans un élément englobant(comme un classeur ou un paquetage)

self : pseudo-attribut référençant l'objet courant

5.2 Opérations de base sur les collections

Nous décrivons ici quelques opérations de base sur les collections que propose le langage OCL. [41]

size(): Integer – retourne le nombre d'éléments (la cardinalité) de *self*.

includes(objet:T): Boolean – vrai si *self* contient l'objet *objet*.

excludes(objet:T): Boolean – vrai si *self* ne contient pas l'objet *objet*.

count(objet:T): Integer – retourne le nombre d'occurrences de *objet* dans *self*.

includesAll(c:Collection(T)): Boolean – vrai si *self* contient tous les éléments de lacollection *c*.

excludesAll(c:Collection(T)): Boolean – vrai si *self* ne contient aucun élément de la collection *c*.

isEmpty() – vrai si *self* est vide.

notEmpty() – vrai si *self* n'est pas vide.

sum():T retourne la somme des éléments de *self*.

forAll([<élément> [: <Type>] |] <expression_logique>) : permet d'écrire une expression logique vraie si l'expression <expression_logique>est vraie pour tous les éléments de self.

exists([<élément> [: <Type>] |] <expression_logique>) : permet d'écrire une expression logique vraie si l'expression <expression_logique> est vraie pour au moins un élément de self.

select([<élément> [: <Type>] |] <expression_logique>) : permet de générer une sous-collection de self ne contenant que des éléments qui satisfont l'expression logique <expression_logique>.

reject([<élément> [: <Type>] |] <expression_logique>) : permet de générer une sous collection contenant tous les éléments de self excepté ceux qui satisfont l'expression logique<expression_logique>.

5.3. Opérations de base sur les ensembles

- **Set :**
  C'est un ensemble au sens mathématique, les doublons ne sont pas admis.
  {1, 5, 4, 8}[40]
- **Bag :**
  Cet ensemble accepte plusieurs même éléments les doublons sont admis.
  {1, 1, 5, 4, 4, 8}[40]
- **Séquence :**
  Cet ensemble peut contenir des doublons et tous ses éléments sont ordonnes.
  {1, 1, 4, 4, 5, 8}[40]
- **L'operateur @pre :**
  - Celui-ci utilise pour spécifier les pre et post-conditions sur les opérations et méthodes dans UML.
  - IL indique la valeur de la propriété au début de l'opération.
  - IL est post fixe. [40]

  Ex : **Context** Person::birthdayHappens()

  Post : age = age@pre + 1

5.4. Sélection dans un sous ensemble

Collection →select(…)

Ex : **Context** Company **inv** :

Self.employee→**select**(age>50)→notEmpty()

- **Reject d'un élément de collection**

  Collection→reject(…)

  Ex: **Context** Company **inv**:

  Self.employee→**reject**(isMarried)→isEmpty()

- **Existence**:

  Collection→exist(…)

  Ex: **context** company **inv**:

  Self.employee→**exists**(forename='rafik')

- **Iteration**

  Collection→iterate(elem:Type;acc:type=<expression>|expression-with-elem-and-acc)

  Ex:

  Collection→iterate(p:Person ;acc=Bag{}|acc→including(p.forename<>'brahim'))

5.5. Opération sur les collections de type Set et Bag

Set→union(set2 :Set(T))

Bag→union(bag2:Bag(T))

Set→intersection(set2:Set(T)) ou

Set→intersection(bag2:Bag(T))

(set ou bag)→select(expressionOCL)

(set ou bag)→reject(expressionOCL)

(set ou bag)→count(object)

5.6. Valeurs élémentaires et types

Type de base	**Operations sur les types**
Booléen	And, or, xor, not, implies, if-then-else
Entier	+,-,*,/,abs(),max(),…
Réel	+,-,*,/,floor(),max(),…
Chaîne de caractères : String	toUpper(), concat(), size()…..
Type	**Valeurs**
Boolean	true, false
Integer	1, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...

Tableau2 - valeurs élémentaires et types [43].

6. Conclusion:

1. **Avantages d'OCL :**
   - Langage formel à la syntaxe simple
   - Bien adapté à une utilisation dans un contexte objet (UML)
   - Permet de spécifier clairement des contraintes sur un ensemble de diagrammes UML
   - Permet de réaliser des spécifications complètes et non ambigües
   - Normalisé par l'OMG
2. **Inconvenient** :
   - Ecriture pouvant tout de même s'avérer complexe dans certains cas
   - Peu d'outils permettant de manipuler des contraintes OCL

Chapitre 4

OCL pour vérifier la cohérence entre les diagrammes des cas d'utilisations et les diagrammes d'activités.

1. Introduction

OCL est considéré comme l'un des nombreux langages de spécifications existants et largement utilisées pour vérifier la cohérence entre les diagrammes comportementaux, cela en raison des fonctionnalités dont il dispose: proche de l'utilisateur et facile à utiliser.

Ainsi, dans notre travail, nous avons résumé et proposé un ensemble de règles écrites en OCL pour assurer la cohérence et éliminer toutes les ambiguïtés:

2. Règles de cohérences D'OCL Entre diagrammes (cas d'utilisation-diagramme d'activité)

Afin d'assurer la cohérence entre un diagramme de cas d'utilisation et un diagramme d'activité, nous présentons un ensemble de règles et leur formalisation à l'aide du langage de spécification OCL.

Dans nos recherches, nous nous sommes appuyés sur un ensemble de documents de recherche des scientifiques afin d'extraire un ensemble de règles de cohérences comportementales relient les deux diagrammes. Parmi les travaux importants que nous avons utilisés pour élaborer les règles de cohérences :

-Vérification de la cohérence entre les cas d'utilisation et diagrammes d'activité [25].

- Chaque cas d'utilisation apparaît a au moins dans un diagramme d'activité **[**26**]**.

- Chaque cas d'utilisation dans le diagramme de cas d'utilisation doit avoir un diagramme d'activité correspondant et pour chaque acteur dans un diagramme de cas d'utilisation, il doit exister une classe dans le diagramme d'activité correspondant [27].

- Définir une action / activité dans le diagramme d'activité en tant qu'événement d'un cas d'utilisation dans un diagramme de cas d'utilisation [28,29].

Les pages suivantes montrent comment les utiliser en utilisant la spécification OCL.

3. Un aperçu

La figure **[Fig35]** donne un aperçu de notre travail pour la vérification et la validation de la cohérence inter diagrammes UML.

Premièrement, nous avons identifié un ensemble de règles correspondantes entre le diagramme de cas d'utilisation et le diagramme d'activité basé sur notre méta-modèle. Ensuite, nous avons élaboré ces règles par l'expression OCL

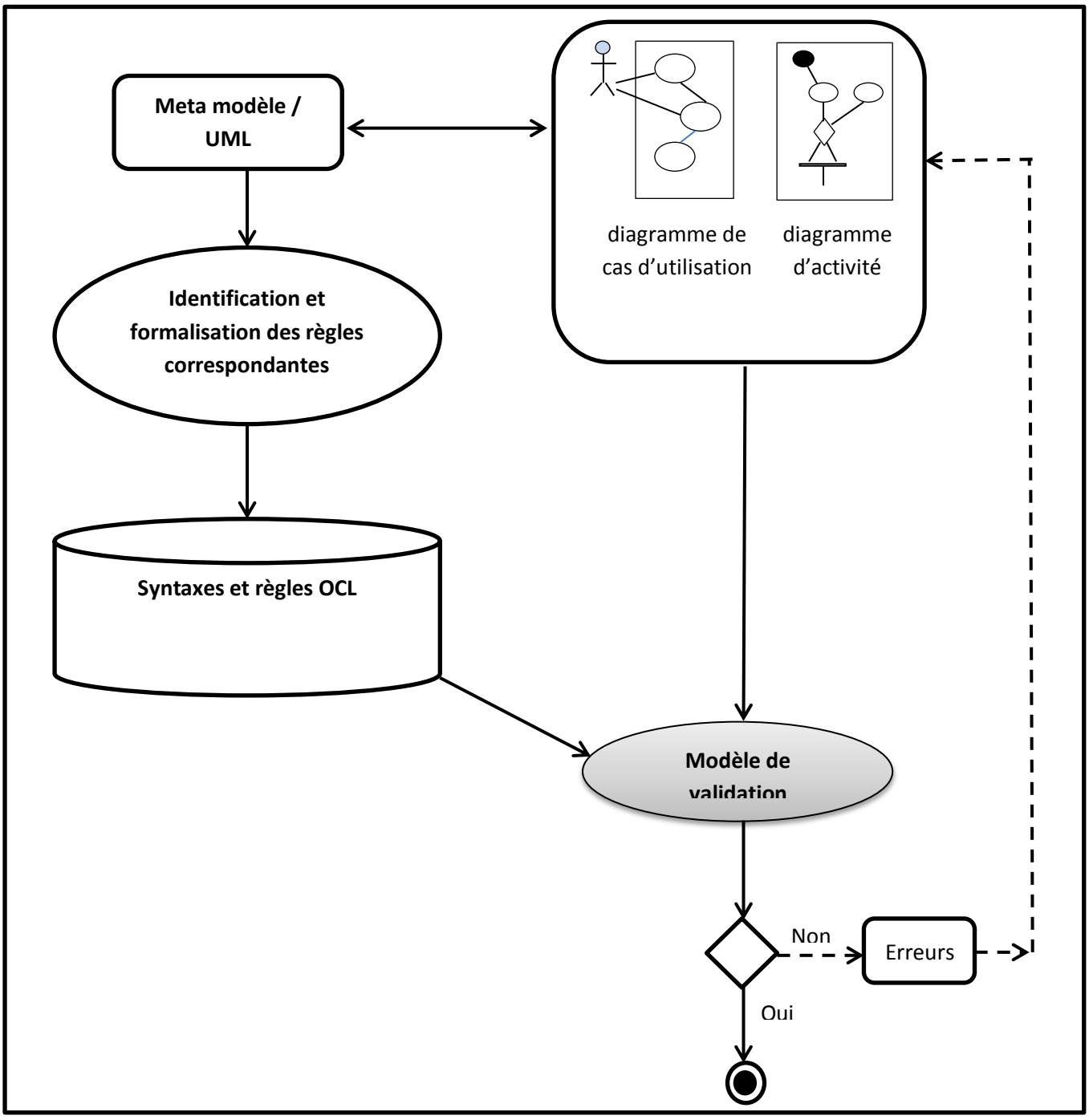


Fig35 - Un aperçu de l'approche de vérification et validation d' UML.

4. Présentation du méta-modèle de diagramme de cas d'utilisation

Un Meta-modèle d'un diagramme de cas d'utilisation décrit comment une entité peut utiliser le système .À cette fin, Il contient un cas d'utilisation et une ligne entre eux qui représente l'implication des acteurs dans une ou plusieurs des interactions couvertes par le cas d'utilisation [31], Il prend aussi une relation entre des cas d'utilisations qui représentée par (association, extension, inclusion).

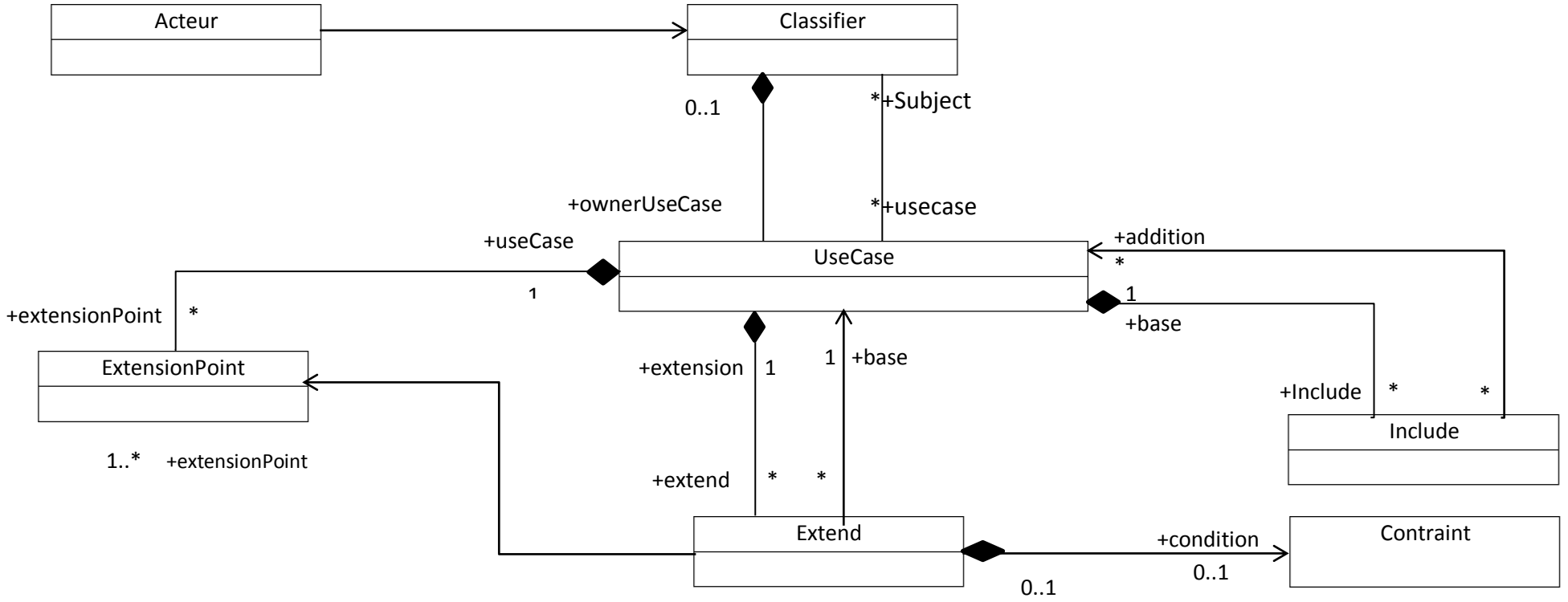


Fig36: Meta-modèle de cas d'utilisation (UseCaseDiagram) [25].

5. Présentation du méta-modèle de diagramme d'activité

Un Meta-modèle d'un diagramme d'activité contient : activité, action, nœuds, pin entrée, pin sortie, pin valeur, flot objet, bord activité, partition, transition. Dans notre travail, nous nous sommes concentrés sur l'utilisation du méta modèle illustré dans la figure **[Fig37]**.

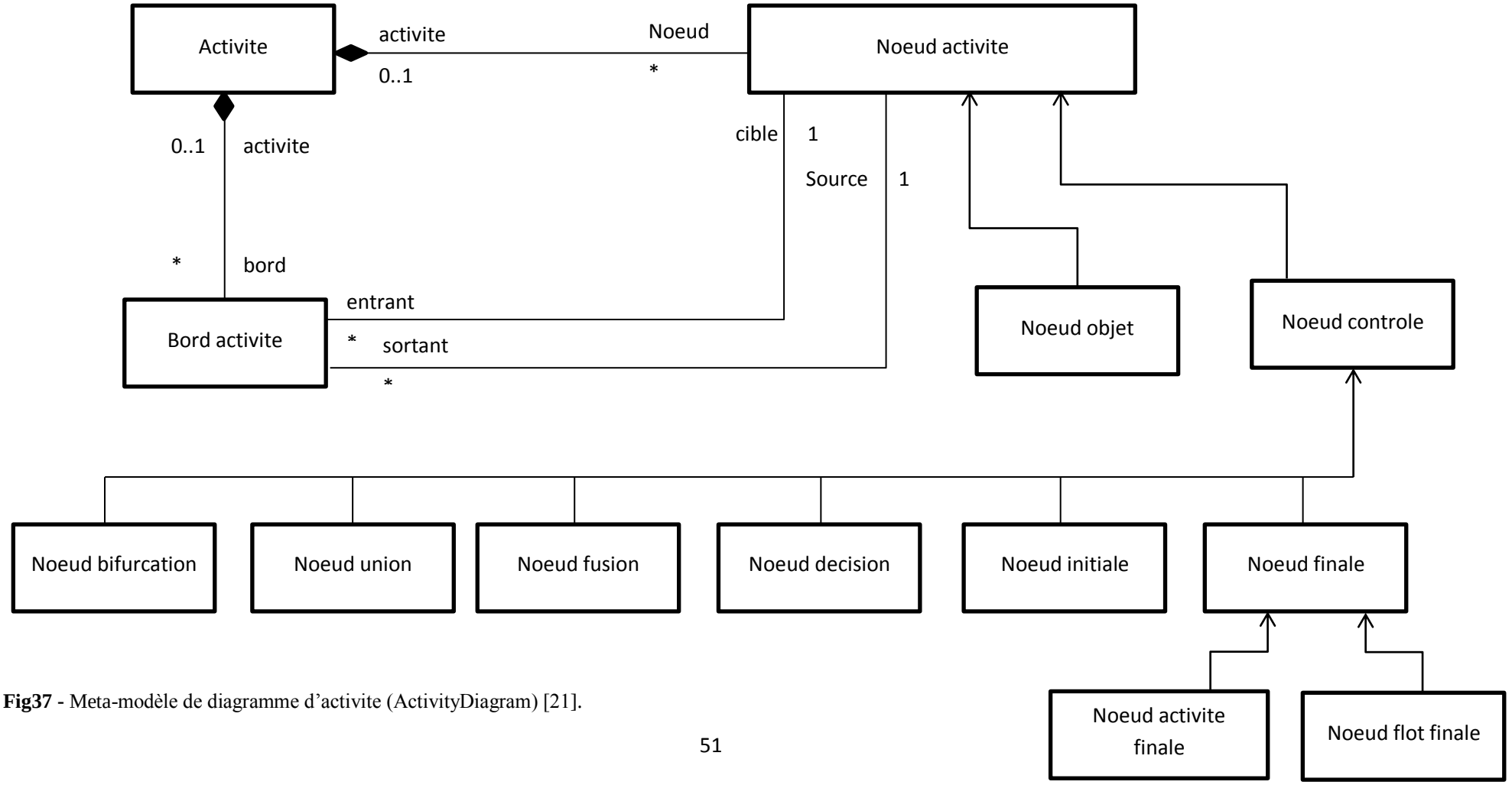


Fig37 - Meta-modèle de diagramme d'activite (ActivityDiagram) [21].

6. Règles de cohérences d'OCL de cas d'utilisation

- **Acteur**

- **Règle 1** : Un acteur ne peut avoir des associations que avec des cas d'utilisations, des sous-systèmes, des composants et des classes et ces Associations sont binaires [31].

```
context acteur inv acteur_R1:
self.associations->forAll(a| a.connection->size = 2 anda.allConnections-> exists(r|
r.participant.oclIsKindOf(acteur)) and
a.allConnections->exists(r| r.participant.oclIsKindOf(UseCase) or
r.participant.oclIsKindOf(Subsystem) orr.participant.oclIsKindOf(Class)))
```

- **Règle 2** : Un acteur doit avoir un nom [31].

```
Context  acteur   inv acteur_R2:
not (self.name = '')
```

- **Règle 3** : Un acteur ne doit pas être contenu par un autre classificateur [31].

```
context acteur inv acteur_R3:
self.ownedElement->isEmpty
```

- **Extension Point**

- **Règle 4** : Un point d'extension doit avoir un nom [31]

```
Context ExtensionPoint inv ExtesionPoint_R4:
not (self.name = '')
```

- **Règle 5** : Les Extension Points référencés doivent être inclus dans l'ensemble d'extension Point dans la UseCase cible [31].

```
Context Extend  inv ExtesionPoint _R5:
self.base.allExtensionPoints ->includesAll (self.extensionPoint)
```

- **Cas d'utilisation**

- **Règle 6** : Un cas d'utilisation doit avoir un nom [31].

```
Context UseCase inv  UseCase_R6:
not (self.name = '')
```

- **Règle 7** : Les cas d'utilisations ne doivent participer qu''a des associations binaires [31].

```
Context UseCase inv  UseCase_R7:
self.associations->forAll(a | a.connection->size = 2)
```

- **Règle 8** : Un cas d'utilisation ne doit pas avoir d'associations avec d'autres cas d'utilisations qui spécifient le même système [31].

```
Context UseCase inv  UseCase_R8:
self.associations->forAll(a| a.allConnections->forAll(s, o|
(s.participant.oclAsType(UseCase).specificationPathN->isEmpty and
o.participant.oclAsType(UseCase).specificationPathN->isEmpty) or (not
s.participant.oclAsType(UseCase).specificationPathN->
includesAll(o.participant.oclAsType(UseCase).specificationPathN) and not
o.participant.oclAsType(UseCase).specificationPathN->
includesAll(s.participant.oclAsType(UseCase).specificationPathN))))
```

- **Règle 9**: Un cas d'utilisation ne peut contenir aucun classificateur [31].

```
Context UseCase inv  UseCase_R9:
self.ownedElement->reject(e| e.oclIsKindOf(Action) or
e.oclIsKindOf(Collaboration))->isEmpty
```

- **Règle 10** : Les noms des extension points doivent être uniques dans un cas d'utilisation[31].

```
Context UseCaseinv  UseCase_R10:
self.allExtensionPoints->forAll(x, y| x.name = y.name implies x = y)
```

- **Extend**

- **Règle 11** : Une relation d'extension a comme source un (et un seul) cas d'utilisation [31].

```
Context Extend inv Extend_R11:
self.base.ExtensionPoints ->forAll (e|e.usecase->size = 1)
```

- **Règle 12** : Une relation d'extension a comme cible un (et un seul) cas d'utilisation [31].

```
Context Extend inv Extend_R12:
self.extension.ExtensionPoints ->forAll (e|e.usecase->size = 1)
```

- **Règle 13** : Le point d'extension qui est référencé par la relation d'extension doit appartenir au cas d'utilisation étendu [31].

```
context Extend inv Extend_R13:
self.base.allExtensionPoints ->includesAll (self.extensionPoint)
```

- **Include**

- **Règle 14** : Une relation d'inclusion a comme source un (et un seul) cas d'utilisation [31].

```
Context UseCase inv UseCase_R14:
self.base.include ->forAll (e|e.usecase->size = 1)
```

- **Règle 15** : Une relation d'inclusion a comme cible un (et un seul) cas d'utilisation [31].

```
Context UseCase inv UseCase_R15:
self.addition.include ->forAll (e|e.usecase->size = 1)
```

7. Règles de cohérence d'OCL de diagramme d'activité

- **Activité**

Contexte : Une activité (activity en anglais) définit un comportement paramétré par un séquencement organisé d'unités subordonnées dont les éléments simples sont les actions. Le flot d'exécutions est modélise par des nœuds reliés par des arcs.

- **Nœud d'activité**

- **Règle 16** : Les nœuds d'activité ne peuvent être possédés que par des activités ou des groupes d'activités [32][NR]..

```
Context Noeudactivite inv Noeudactivite _R16:
self.entry->size > 0
```

- **Action**

- **Règle 17** : Un nœud d'action doit avoir au moins un arc entrant [32][NR].

```
Context ActionState inv ActionState_R17:
self.entry->size > 0
```

- **Nœud Initiale**

- **Règle18** : Un nœud initial ne doit pas avoir d'arc entrant [32][NR].

```
Context Noeudinitiale inv Noeudinitiale _R18:
self.entry->size =  0
```

- **Règle19** : Un nœud initial a au moins un arc sortant [32][NR].

```
Context Noeudinitiale inv Noeudinitiale _R19:
self.sortant->size > 0
```

- **Nœud Finale et Nœud Flot finale et Nœud Flot initiale**

  - **Règle20** : Un nœud final ne doit pas avoir d'arcs sortants [32][NR].

```
Context Noeudfinale inv Noeudfinale _R20:
self.sortant->size = 0
```

  - **Règle21** : Un nœud final a au moins une transition entrante [32][NR].

```
Context Noeudfinale inv Noeudfinale _R21:
self.entry->size > 0
```

- **Nœud fusion**

  - **Règle22**: Un nœud d'interclassement possède un arc sortant [32][NR].

```
Context Noeudfusion  inv Noeudfusion _R22:
self.sortant->size =1
```

  - **Règle23**: Un nœud d'interclassement possède au moins deux arcs entrants [32][NR].

```
Context Noeudfusion inv Noeudfusion _R23:
self.entry->size >=2
```

- **Nœud Décision :**

  - **Règle24**: Tout nœud de décision doit avoir exactement un arc entrant [32][NR].

```
Context Noeuddecision inv Noeuddecision _R24:
self.entry->size =1
```

- **Nœud bifurcation :**

  - **Règle25**: Un nœud de bifurcation doit posséder exactement un arc entrant [32][NR].

```
Context Noeudbufircation inv Noeudbufircation_25:
self.entry->size =1
```

  - **Règle26**: Un nœud de bifurcation doit posséder au moins deux arcs sortants [32][NR].

```
Context Noeudbufircation inv Noeudbufircation _R26:
self.sortant->size >= 2
```

- **Nœud d'union**

  - **Règle27**: Un nœud d'union doit avoir exactement un arc sortant [32][NR].

```
Context Noeudunion inv Noeudunion_R27:
self.sortant->size =1
```

  - **Règle28**: Un nœud d'union doit avoir au moins un arc entrant [32][NR].

```
Context Noeudunion inv Noeudunion_R28:
self.entry->size >0
```

Note : [NR] Nouvelle Règle.

8. Règles de cohérences d'OCL entre diagrammes (cas d'utilisation – activité) :

Règle1

Chaque cas d'utilisation est décrit par au moins un diagramme d'activité. Ce diagramme doit être composé à au moins par un nœud initial, un nœud final et un nœud d'activité [33]

Règle de cohérence	R1
Règle à un niveau indépendant du méta-modèle	Chaque cas d'utilisation est décrit par au moins un Diagramme d'activité. Ce diagramme doit être composé à au moins un nœud initial, un nœud final et une activité nœud.
expression OCL	Context UseCase inv UCD_AD1: UseCaseDiagram->forAll(ucd | Exists(ad:ActiviteDiagram | Exists(act :ad.activite|forAll(a :ucd.usecase| Exists(u:a.SNominal|Exists(s:u.action| s.name=act.name ∧s.numAction=act.numAction))))))

Tableau3 -R1 : Règle de cohérence inter-diagrammes

Règle2

Un scénario alternatif **AS** (ou scénario d'erreur **ES**) d'un cas d'utilisation est représenté par un diagramme d'activité composé d'au moins un nœud de décision et d'un ensemble d'activités reliées par un flux de contrôle ou flux de données[33]

Règle de cohérence	R2
Règle à un niveau indépendant du méta-modèle	Un scénario alternatif **AS** (ou scénario d'erreur **ES**) d'un cas d'utilisation est représenté par un diagramme d'activité composé d'au moins un nœud de décision et d'un ensemble d'activités reliées par un flux de contrôle ou flux de données.
expression OCL	Context UseCase inv UCD_AD2: forAll(ucd | Exists(ad | Exists(act :ad.activity|forAll(a :ucd.usecase|Exists(u:a.SAlternatifl|Exists(s:u.action|s.name=act.name ∧s.numAction=act.numAction))))))

Tableau4 -R2 : Règle de cohérence inter-diagrammes

Règle3

Chaque action d'un acteur est représentée par une activité et la pré-condition dans le cas d'utilisation est représentée par la condition de garde dans l'activité.

Note : L'ordre de l'action dans le cas d'utilisation doit être préservé par l'ordre de l'activité dans le diagramme d'activité [33]

Règle de cohérence	R3
Règle à un niveau indépendant du méta-modèle	Chaque action d'un acteur est représentée par une activité et la pré-condition dans le cas d'utilisation est représentée par la condition de garde dans l'activité.
expression OCL	Context UseCase inv UCD_AD3: forAll(ucd | forAll(a :ucd.usecase|Exists(u:a.SNominal|Exists(s:u.action|Exists(ad | forAll (act:ad.activity | s.conditionGarde=act.conditionGarde))))) forAll(ucd | forAll(a :ucd.usecase|Exists(u:a.SAlternatif|Exists (s:u.action|Exists(ad | forAll (act:ad.activity | s.conditionGarde=act.conditionGarde))))) forAll(ucd | forAll(a :ucd.usecase|Exists(u:a.SErreur|Exists(s:u .action|Exists(ad | forAll (act:ad.activity | s.conditionGarde=act.conditionGarde)))))

Tableau5 -R3 : Règle de cohérence inter-diagrammes

Règle4

La première action effectuée par un acteur est représentée par la première activité dans le diagramme d'activité correspondant.

Note : L'activité doit être liée avec le nœud initial et avoir la même signature avec l'action correspondante [33]

Règle de cohérence	R4
Règle à un niveau indépendant du méta-modèle	La première action effectuée par un acteur est représentée par la première activité dans le diagramme d'activité correspondant.
expression OCL	Context UseCase inv UCD_AD4: forAll(ucd | forAll(a :ucd.usecase|Exists(u:a.SNominal|Exists(s:u.action|Exists(ad | forAll (act :ad.activity | s.numAction =1 ∧ act.numAction =1 ∧ s.name =act.name))))))

Tableau6 -R4 : Règle de cohérence inter-diagrammes

Règle5

La dernière action effectuée par le système du scénario nominal est représenté par la dernière activité du diagramme d'activité qui est correspondant au nœud final par un flux de contrôle [33]

Note : l'action et l'activité ont la même signature.

Règle de cohérence	R5
Règle à un niveau indépendant du méta-modèle	La dernière action effectuée par le système du scénario nominal est représenté par la dernière activité du diagramme d'activité qui est correspondant au nœud final par un flux de contrôle.
expression OCL	Context UseCase inv UCD_AD5: forAll(ucd | forAll(a :ucd.usecase|Exists(u:a.SNominal|Exists(s:u.action|Exists(ad | forAll (act :ad.activity | s.numAction =u.nbaction∧ act.numAction =ad.nbactivity∧ s.name =act.name))))))

Tableau7 -R5 : Règle de cohérence inter-diagrammes

Règle6

Toutes les pré-conditions d'un cas d'utilisation sont représentés par la condition de déclenchement du premier événement du diagramme d'activité correspondant[33]

Règle de cohérence	R6
Règle à un niveau indépendant du méta-modèle	Toutes les pré-conditions d'un cas d'utilisation sont représentés par la condition de déclenchement du premier événement du diagramme d'activité correspondant.
expression OCL	Context UseCase inv UCD_AD6: forAll(ucd | forAll(a :ucd.usecase|Exists(ad | forAll (act :ad.activity | a.precondition =act.conditionGarde∧ act.numAction =1))))

Tableau8 -R6 : Règle de cohérence inter-diagrammes

Règle7

Toutes les post-conditions d'un cas d'utilisation sont représentées par la condition du flux entre la dernière activité et le nœud final du diagramme d'activité correspondant au cas d'utilisation [33]

Règle de cohérence	R7
Règle à un niveau indépendant du méta-modèle	Toutes les post-conditions d'un cas d'utilisation sont représentées par la condition du flux entre la dernière activité et le nœud final du diagramme d'activité correspondant au cas d'utilisation.
expression OCL	Context UseCase inv UCD_AD7: forAll(ucd | forAll(a :ucd.usecase|Exists(ad | forAll (act :ad.activity | a.postcondition =act.conditionGarde))))

Tableau9 - R7 : Règle de cohérence inter-diagrammes

9. Conclusion

Nous avons proposé dans ce chapitre une spécification des règles de cohérences pour la vérification et la validation des modèles UML en utilisant deux diagrammes comportementaux (diagramme de cas d'utilisation et diagramme d'activité). Les règles sont illustrées pour chacun des diagrammes en utilisant le langage de spécification formel OCL. Dans le chapitre suivant, nous illustrerons l'utilisation de ces règles à la base d'une étude de cas spécifié en utilisant UML.

Etude de cas

Nous essayerons à travers un exemple de montrer en détails l'application de notre approche de vérification et validation. Nous avons créé un Meta-modèle et modèle d'une compagnie en utilisant l'environnement OCLE-2.04 [39].

Notre application **EMS (EmployeeMangement System)** a un ensemble de classes, la classe **personne** contient les employeurs de la compagnie (célibataires et mariées), la classe **company** a un ensemble d'attributs et méthodes (**numberofemployees** : le nombre d'employés qui ne dépasse pas 5000), **stockprice**() : le stock en dinars, **company**() : une méthode qui donne le nombre d'employées par compagnie, la classe **job** :contient la date de début de l'employé et sa salaire mensuel , la compagnie a une relation directe avec la banque afin de gérer l'aspect financier de ses employés.
L'entreprise peut également embaucher de nouvelles personnes qui répondant aux conditions d'emploi représenter par la classe **NewPerson**.

Notre diagramme de classes en relation entre eux à travers des associations, des rôles et Multiplicités et ça Pour l'indication sur la participation de la classe à l'association et la contrainte sur le nombre d'objets associés

Nos digrammes et captures d'écran d'OCLE, sont représentés dans la section suivante :

1. Diagrammes d'EMS

1.1. Diagramme de classe EMS

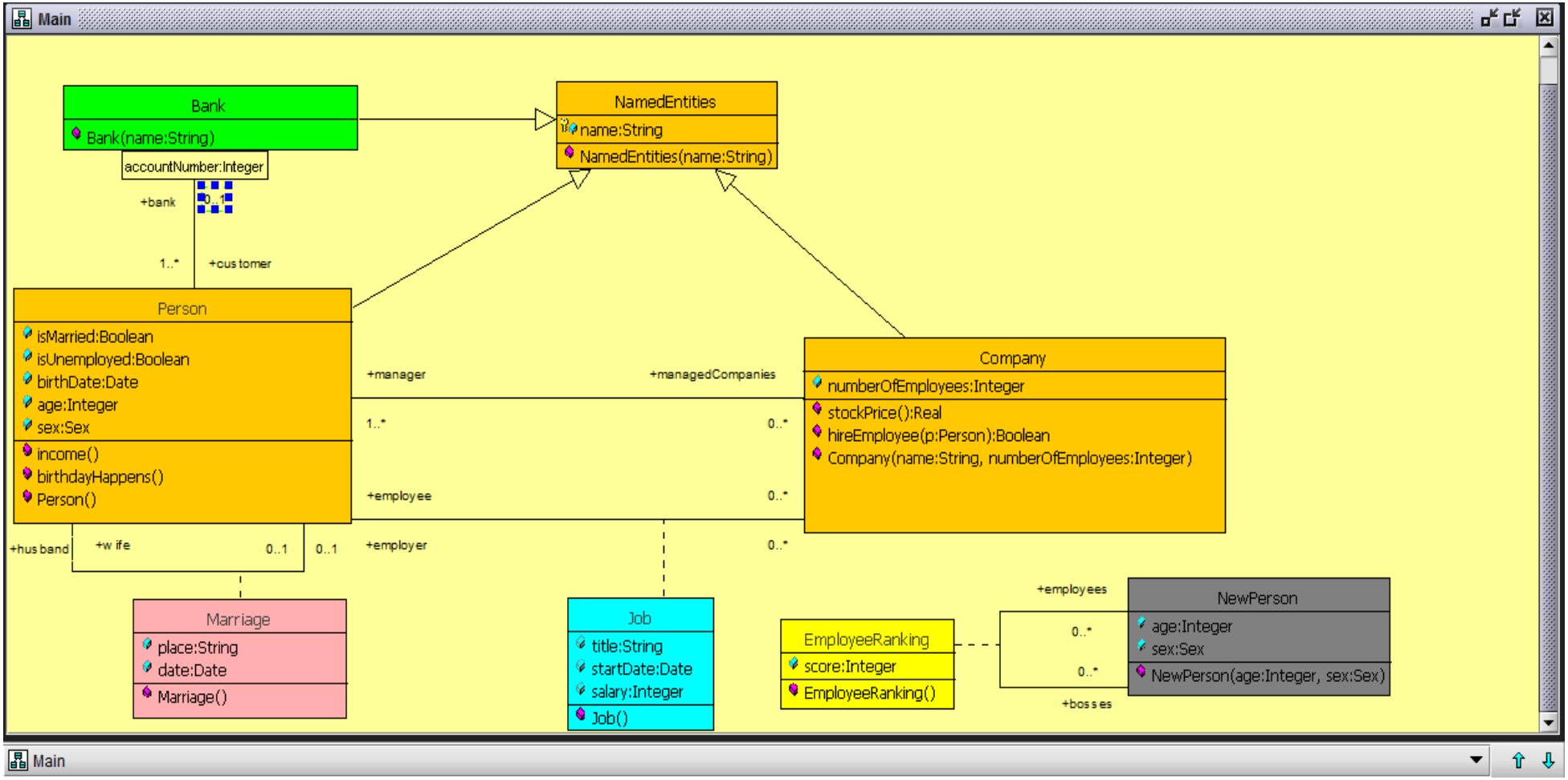


Fig38 - diagramme de classe EMS (Employée Mangement System)

1.2. Fichier OCL de Modèle EMS

E:\glsd\memoire\ocle-2.0.4\EMS\EMS.bcr

```
model EMS

-- Navigating association classes

context Person

    inv managersSalary:

        if self.managedCompanies->isEmpty
            then true
            else self.managedCompanies->forAll(c | (self.employer->includes(c))
                and (self.job->select(j | j.employer = c).salary->sortedBy(s | s)->last = c.job.salary->sortedBy(s| s)->last))
        endif



    inv isMarried:

        let marr: Set(Marriage) = Marriage.allInstances in

        self.is
            if sex = Sex::male
                then marr->exists(m:Marriage| m.husband = self)
                else marr->exists(m:Marriage| m.wife = self)
```

Insert 1:1 Write enable

Figure 30: Un aperçu de l'approche de validationd' UML.

Fig39 - fichier OCL de modèle EMS

1.3. Premier diagramme de cas d'utilisation <Paiement>

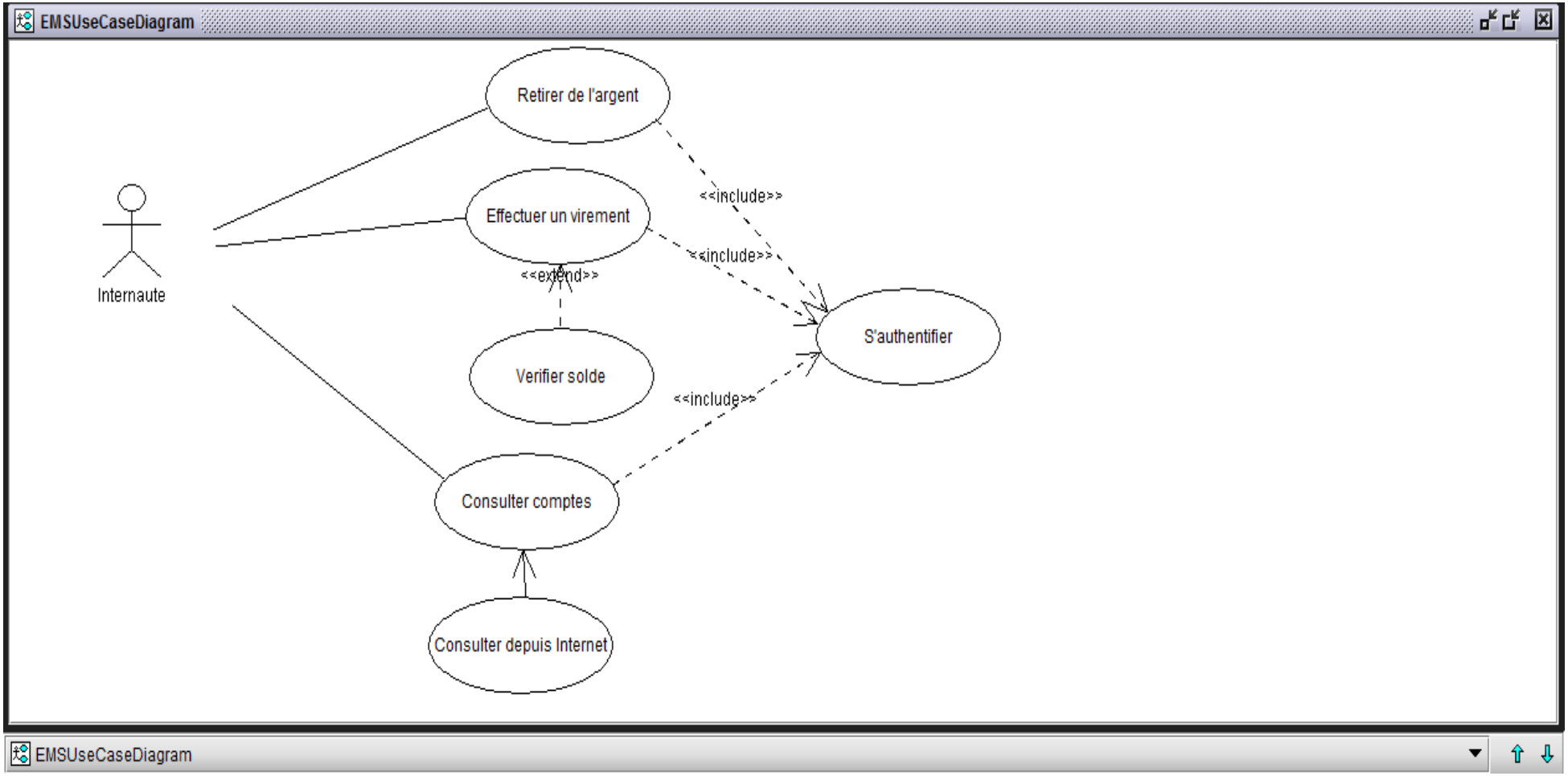


Fig40 - diagramme de cas d'utilisation<paiement>

1.4. Deuxième diagramme de cas d'utilisation < Recrutement >

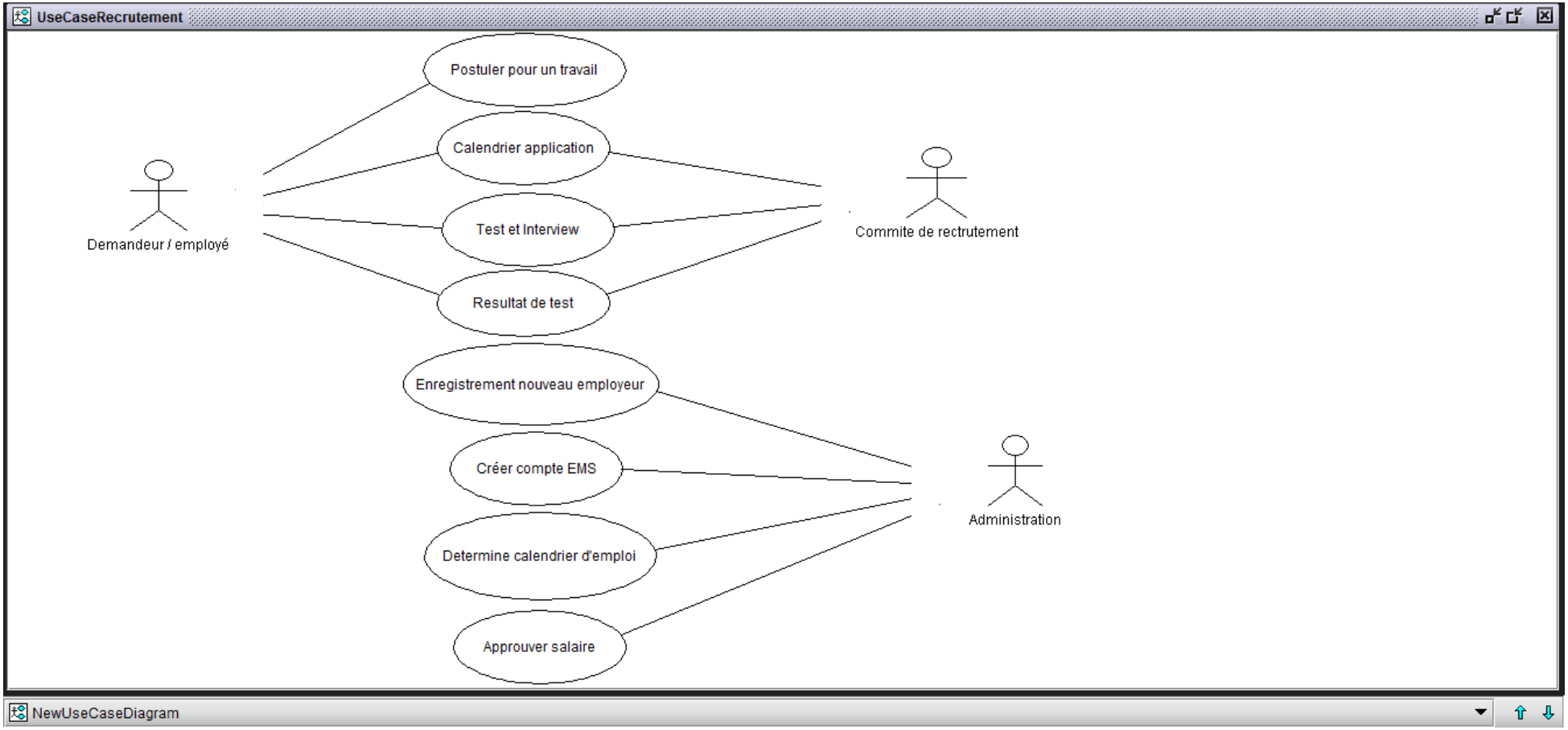


Fig41 - diagramme de cas d'utilisation <recrutement>

1.5. Fichier OCL de cas d'utilisation EMS

E:\glsd\memoire\ocle-2.0.4\EMS\inv_Use_Cases.ocl

```
package Behavioral_Elements::Use_Cases

context Actor

inv Actor_WFR_1:
--   [1] Les acteurs ne peuvent avoir des associations que pour les cas d'utilisation, les sous-systèmes et les classes et ces associations sont bin

    self.associations->forAll(a| a.connection->size = 2 and a.allConnections->exists(r| r.participant.oclIsKindOf(Actor)) and
    a.allConnections->exists(r| r.participant.oclIsKindOf(UseCase) or r.participant.oclIsKindOf(Subsystem) or
    r.participant.oclIsKindOf(Class)))

inv Actor_WFR_2:
--   [2] Les acteurs ne peuvent contenir aucun classificateur.

    self.ownedElement->isEmpty

context Extend

inv Extend_WFR_1:
--   [1] Les ExtensionPoints référencés doivent être inclus dans l'ensemble d'ExtensionPoint dans le UseCase cible.

    self.base.allExtensionPoints -> includesAll (self.extensionPoint)

context ExtensionPoint
```

Insert 3:14 Write enabled

E:\glsd\memoire\ocle-2.0.4\EMS\inv_Use_Cases.ocl

Fig42 - fichier OCl de cas d'utilisation

1.6. Premier diagramme d'activité <Paiement>

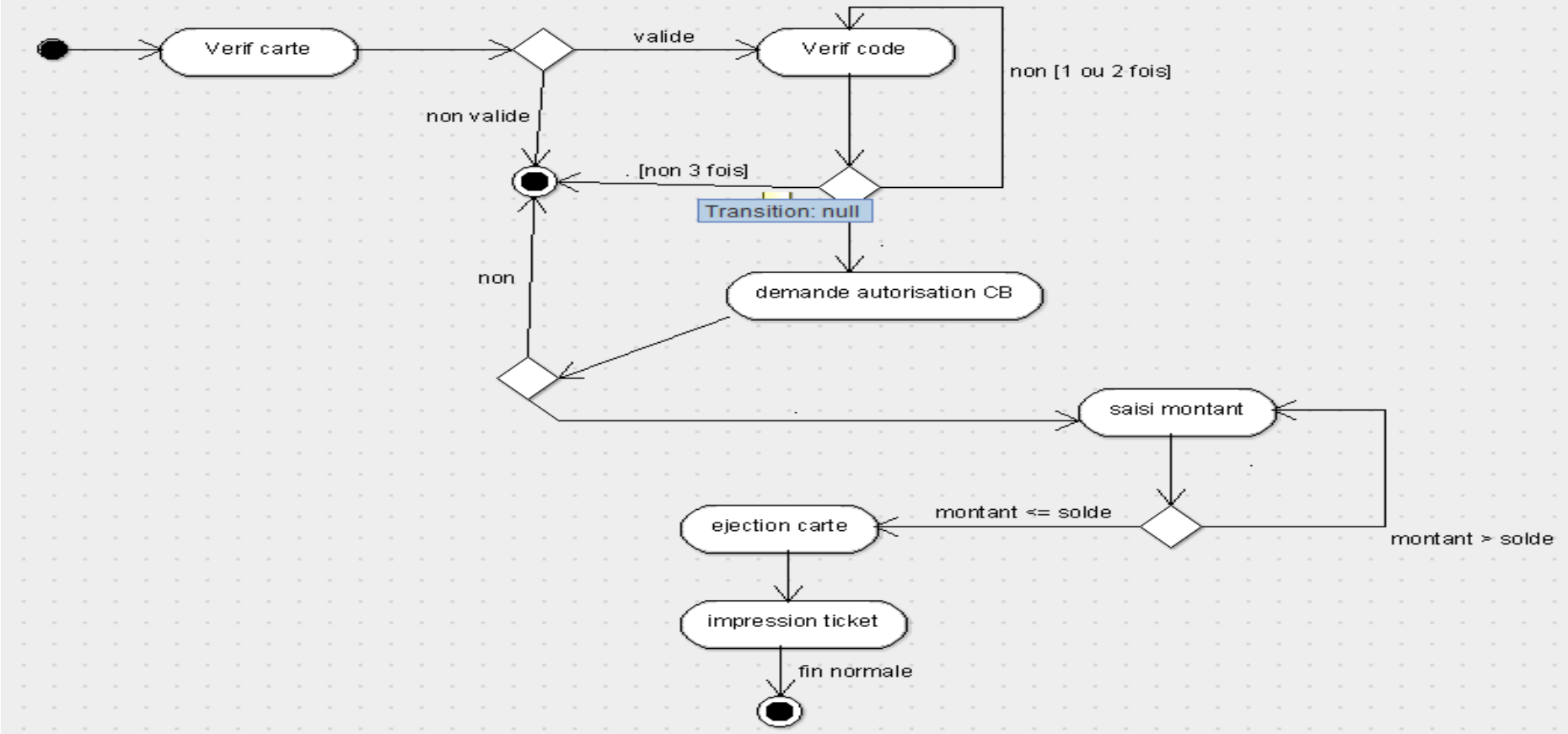


Fig43 - diagramme d'activité <Paiement>

1.7. Deuxième diagramme d'activité <Recrutement>

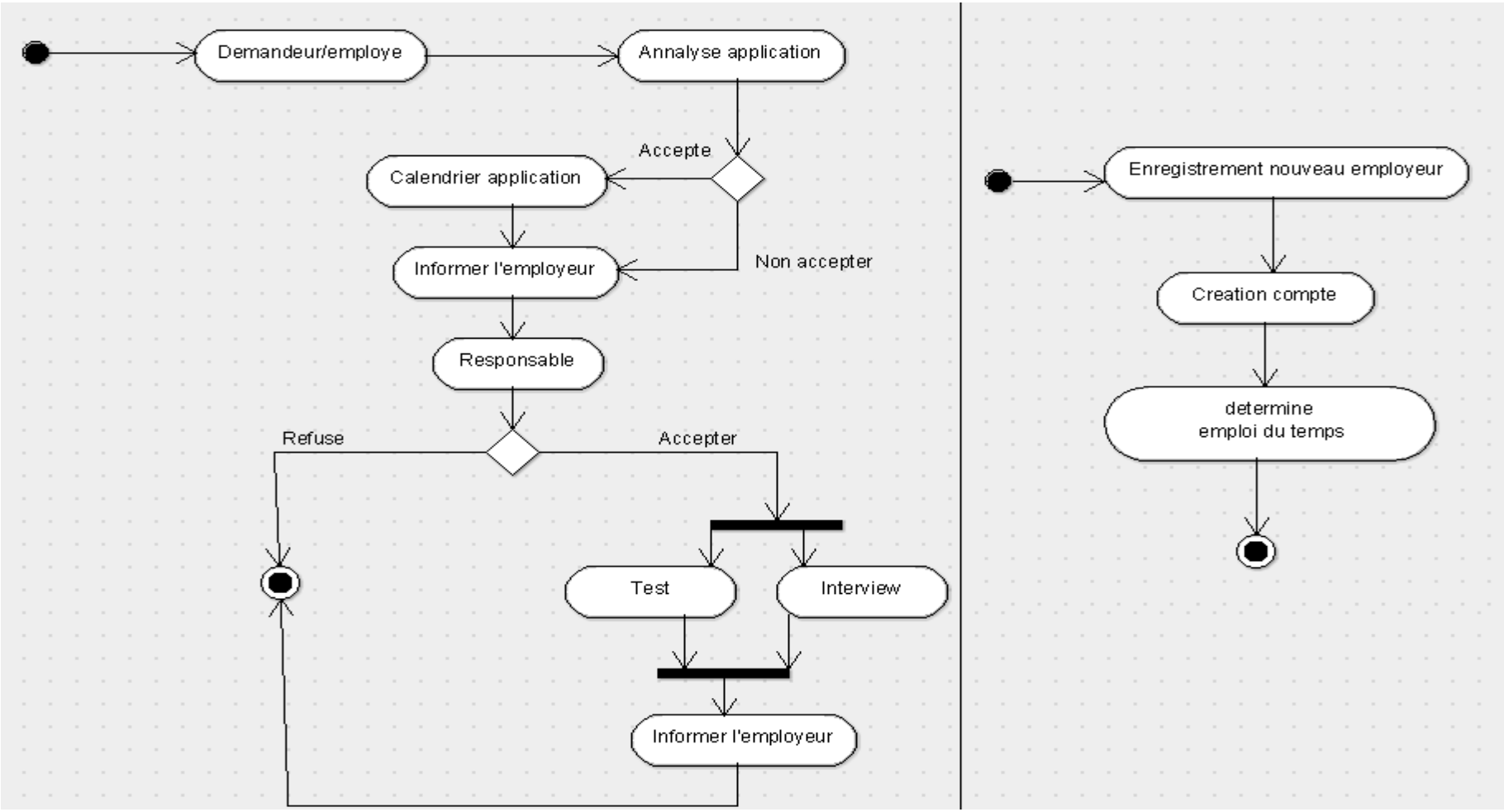


Fig44 - diagramme d'activité <Recrutement>

1.8. Fichier OCL diagramme d'activité, OCL inter-diagrammes d'EMS

E:\glsd\memoire\ocle-2.0.4\EMS\inv_Activity_Graphs.ocl

```
package Behavioral_Elements::Activity_Graphs

context ActivityGraph

inv ActivityGraph_WFR_1:

--    [1] Un ActivityGraph spécifie la dynamique de (i) a Package, or (ii) a Classifier (including UseCase), or
--        (iii) a BehavioralFeature.

    (self.context.oclIsTypeOf(Package) xor  self.context.oclIsKindOf(Classifier) xor self.context.oclIsKindOf(BehavioralFeature))


context ActionState

inv ActionState_WFR_1:
```

Insert 4:1 Write enabled

E:\glsd\memoire\ocle-2.0.4\EMS\inv_UseCases_ET_ActivityGraphs.ocl*

```
package Behavioral_Elements::Use_Cases

--R1  Chaque cas d'utilisation est represente par un diagramme d'activite

context Actor
inv UCD_AD1:

self.associations->forAll(a | a.allConnections->exists(r| r.participant.oclIsKindOf(UseCase) implies a.allConnections->exists(Act| Act.participar


--R2
```

Insert 11:5 Write enabled

Fig45 - fichier OCL de diagramme d'activité, OCL inter-diagrammes d'EMS

1.9. Fichier EMS.xml d'EMS

C:\Users\brahim\ocle_2.0\Temporary\EMS.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<XMI xmlns:UML="//org.omg/UML/1.3" xmi.version="1.1">
 <XMI.header>
  <XMI.documentation>
   <XMI.exporter>ocle 2.0</XMI.exporter>
   <XMI.exporterVersion>1.0</XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
 </XMI.header>
 <XMI.content>
  <UML:Model isAbstract="false" isLeaf="false" isRoot="false" isSpecification="false" name="EMS" visibility="public" xmi.id="S.1" xmlns:UML="http://lci/xm
   <UML:Namespace.ownedElement>
    <UML:Stereotype baseClass="Class" isAbstract="false" isLeaf="false" isRoot="false" isSpecification="false" name="type" namespace="S.1" visibility="pub
    <UML:Stereotype baseClass="Package" isAbstract="false" isLeaf="false" isRoot="false" isSpecification="false" name="facade" namespace="S.1" visibility=
    <UML:Stereotype baseClass="Usage" isAbstract="false" isLeaf="false" isRoot="false" isSpecification="false" name="send" namespace="S.1" visibility="pub
    <UML:Association isAbstract="false" isLeaf="false" isRoot="false" isSpecification="false" link="S.20 S.21 S.22" name="{Company-Person}{3C3D4AD9009E}"
     <UML:Association.connection>
      <UML:AssociationEnd aggregation="none" association="S.5" changeability="changeable" isNavigable="true" isSpecification="false" linkEnd="S.7 S.8 S.9"
       <UML:AssociationEnd.multiplicity>
        <UML:Multiplicity xmi.id="S.10">
         <UML:Multiplicity.range>
          <UML:MultiplicityRange lower="0" multiplicity="S.10" upper="-1" xmi.id="S.11"/>
         </UML:Multiplicity.range>
        </UML:Multiplicity>
```

Insert 1:1 Write enabled

C:\Users\brahim\ocle_2.0\Temporary\EMS.xml

*

Fig46 - fichier EMS.xml d' EMS

1.10. Fichier MAIN.xml d'EMS

C:\Users\brahim\ocle_2.0\Temporary\Diagrams\Main.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<XMI xmlns:GXML="//lci.cs.ubbcluj.ro/gxml" xmlns:UML="//org.omg/UML/1.3" xmi.version="1.1">
 <XMI.header>
  <XMI.documentation>
   <XMI.exporter>ocle 2.0</XMI.exporter>
   <XMI.exporterVersion>1.0</XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="GXAPI" xmi.version="0.1"/>
 </XMI.header>
 <XMI.content>
  <GXML:ClassDiagram background="java.awt.Color[r=255,g=255,b=153]" gridVisible="false" isLocked="false" name="Main" xmi.id="S.1" xmlns:GXML="http://lci/x
   <GXML:ClassDiagram.owner>
    <UML:Model href="../EMS.xml|S.1" xml:link="simple" xmlns:UML="http://lci/xml"/>
   </GXML:ClassDiagram.owner>
   <GXML:ClassDiagram.flat>
    <GXML:DefaultGraphCell allowsChildren="true" userObjectString="0..*" xmi.id="S.2">
     <GXML:DefaultGraphCell.attributes>
      <GXML:HashtableStrings keys="moveable value" values="true 0..*" xmi.id="S.3"/>
     </GXML:DefaultGraphCell.attributes>
    </GXML:DefaultGraphCell>
    <GXML:SpecialEdge allowsChildren="true" children="S.8" isRouted="false" source="S.6" target="S.5" xmi.id="S.4">
     <GXML:DefaultGraphCell.attributes>
      <GXML:HashtableStrings keys="endsize editable linewidth bendable moveable lineEnd beginSize" values="18 false 1.0 true true 2 18" xmi.id="S.7">
       <GXML:HashtableStrings.value>
```

Insert 1:1 Write enabled

C:\Users\brahim\ocle_2.0\Temporary\Diagrams\Main.xml

Fig47 - fichier MAIN.xml d' EMS

1.11. Description textuelle des cas d'utilisation

Cas d'utilisation : Enregistrement nouveau employeur

Acteur : Employeur

Pré-conditions : aucune

Post-conditions : Les données d'employeur sont accessibles par le service administratif.

Scénario nominal

Début

1. Sauvegarder les données de l'employeur.
2. création un compte.
3. Déterminer emploi du temps

Fin

Scénario Alternative

Début

Aucune

Fin

Fin Cas d'utilisation

La compagnie à ensemble de fonctions de base qui sont :

1. Publicité en ligne

2. Accepter les candidatures

3. Inscription de nouveaux employés

4. Création de comptes et présence pour les employés.

5. Les employés peuvent vérifier leurs détails / statut et marquer leur présence.

6. L'employé, le gestionnaire ou l'administrateur voient leurs performances.

7. Calcul du salaire net et livraison en fonction de l'assiduité.

Le système fournit une partie comité de recrutement pour traiter la candidature des candidats, une partie administrative pour enregistrer les employés, une partie gestionnaire pour gérer les données du personnel, une partie employé pour marquer la présence ou voir ses coordonnées / statut et une partie finance pour contrôler le paiement du salaire.

1.12. Objectif EMS

-Le rôle de l'EMS est de centraliser le référentiel des données du personnel, y compris le recrutement, les présences et les salaires. Un système de gestion des employés efficace permet de générer des informations précises et opportunes sur les employés pour atteindre les objectifs.

- conçu pour satisfaire les employés en fournissant des services tels que le salaire avec précision, à temps et rapidement. Les employés sont gérés facilement. Les systèmes de présence et de salaire des employés sont automatisés. Le système stockera toutes les données et transactions du personnel.

- Fournir une image complète de l'information.

-Accroître la vitesse des transactions d'informations.

-Améliorer la satisfaction des employés en leur fournissant des services en temps opportun, plus rapidement et plus précisément.

- mettre en place un système d'information des employés sur le statut de l'employé et l'assiduité des employés et le processus de salaire mensuel et la livraison.

1.13. Exemple diagramme de cas d'utilisation d'incohérence

Nous prenons un cas d'utilisation représente dans la figure Fig[48] Mais nous ajoutons une association entre les deux acteurs

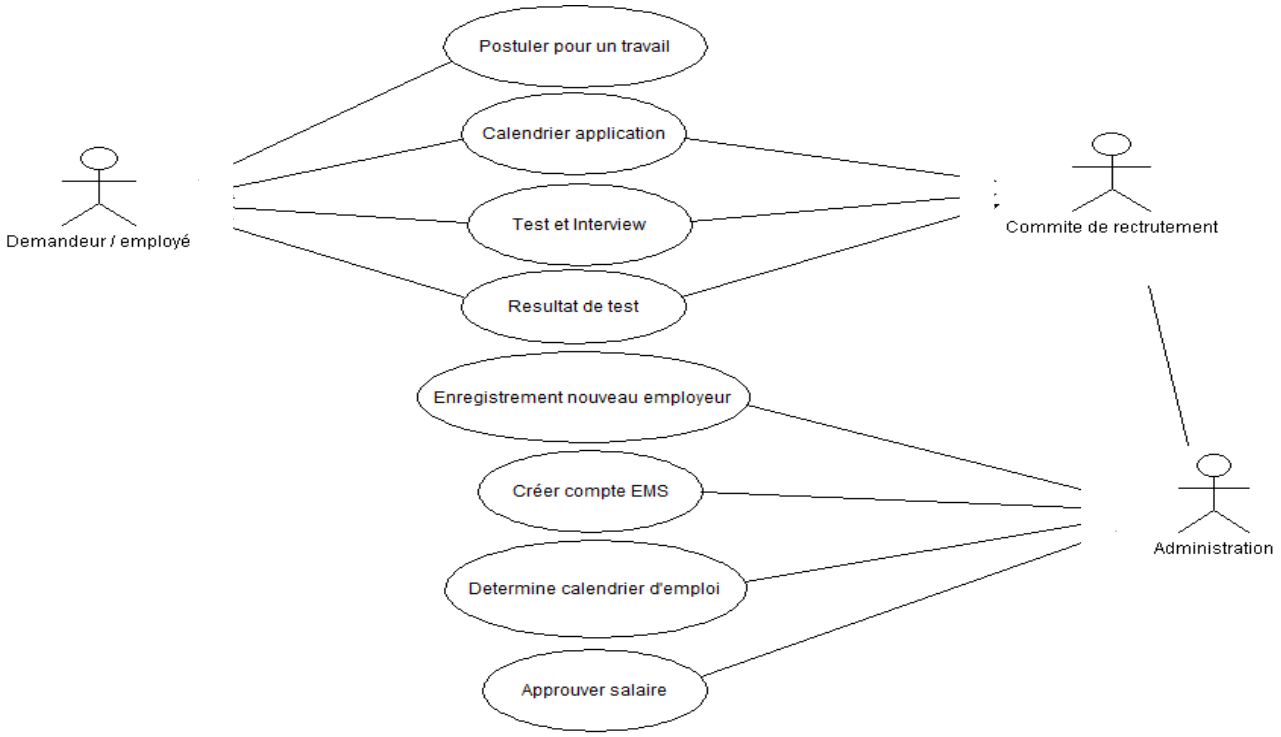


Fig48 - Exemple diagramme de cas d'utilisation incohérent

Nous allons obtenir une erreur d'incohérence dans la règle R1 représenté dans le fichier Meta-modèle **inv_use_case.ocl**

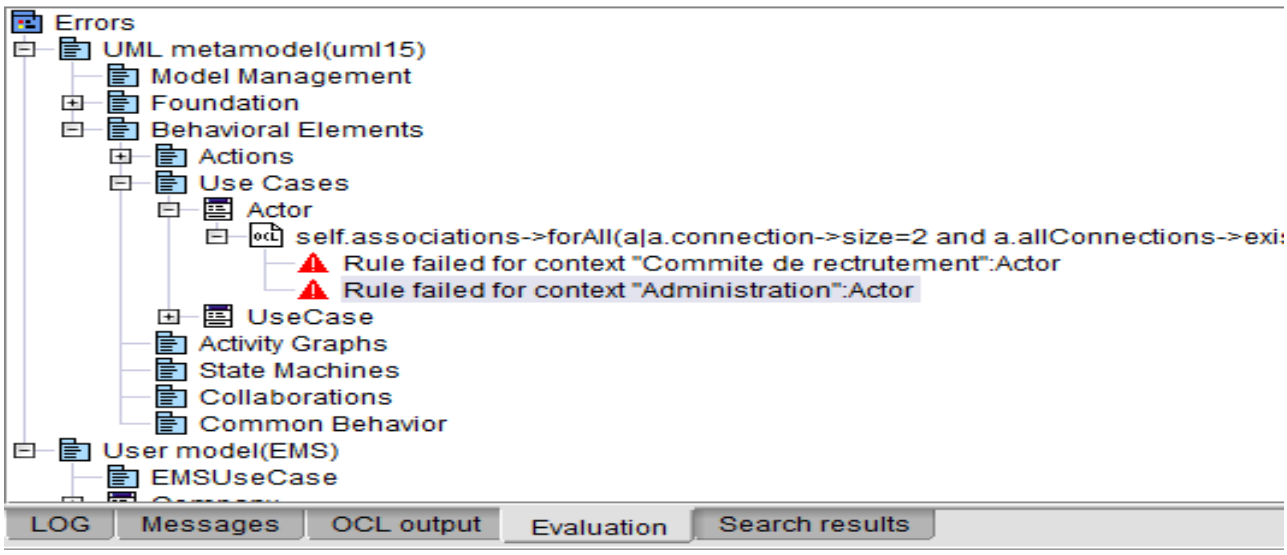


Fig49 - Evaluation de projet

1.14. Exemple de diagramme de cas d'utilisation cohérent

Nous utilisons un diagramme de cas d'utilisation qui contient un ensemble des tâches que l'employé peut effectuer : Marquer la présence, Mettre à jour le statut, Afficher ces détails de l'employé qui sont Montrés dans le diagramme suivant.

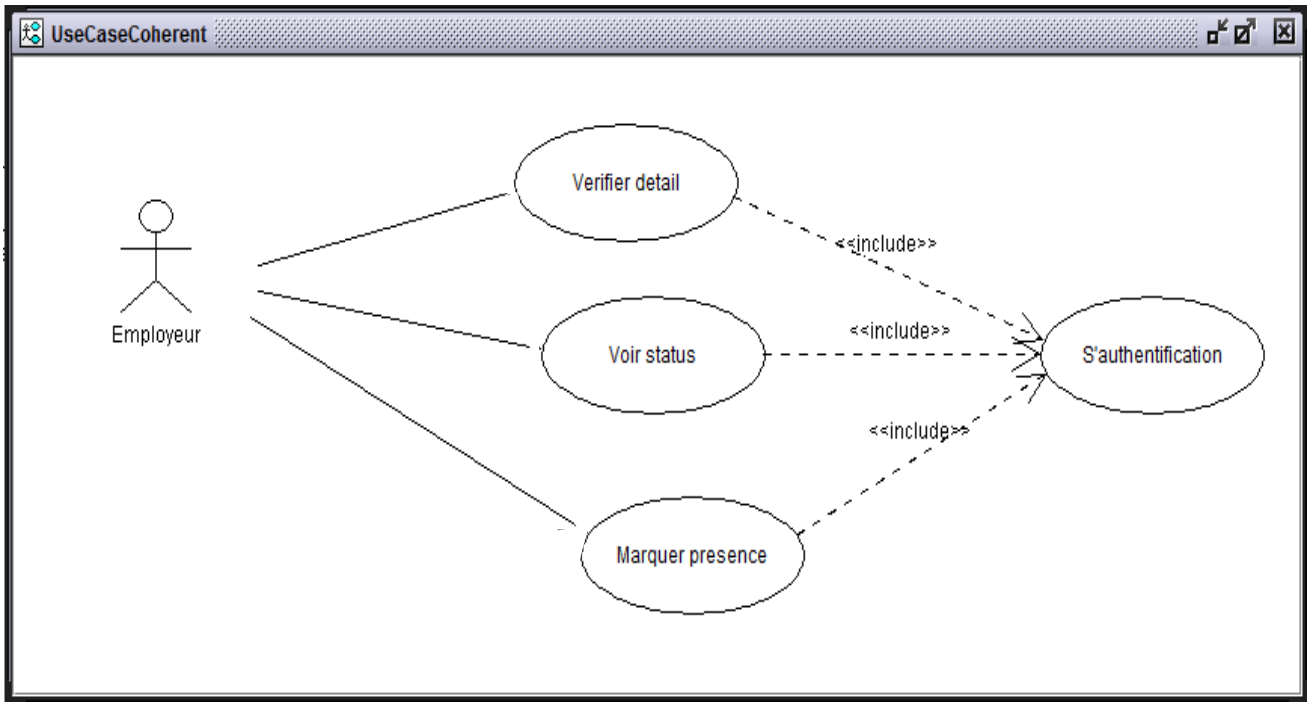


Fig50 - Exemple diagramme de cas d'utilisation cohérent

1.15. Exemple de diagramme d'activité cohérent

Nous utilisons aussi un diagramme d'activité qui recueille les taches suivantes : Marquer la présence, Mettre à jour le statut, Afficher ces détails de l'employé qui sont Montrés dans le diagramme suivant.

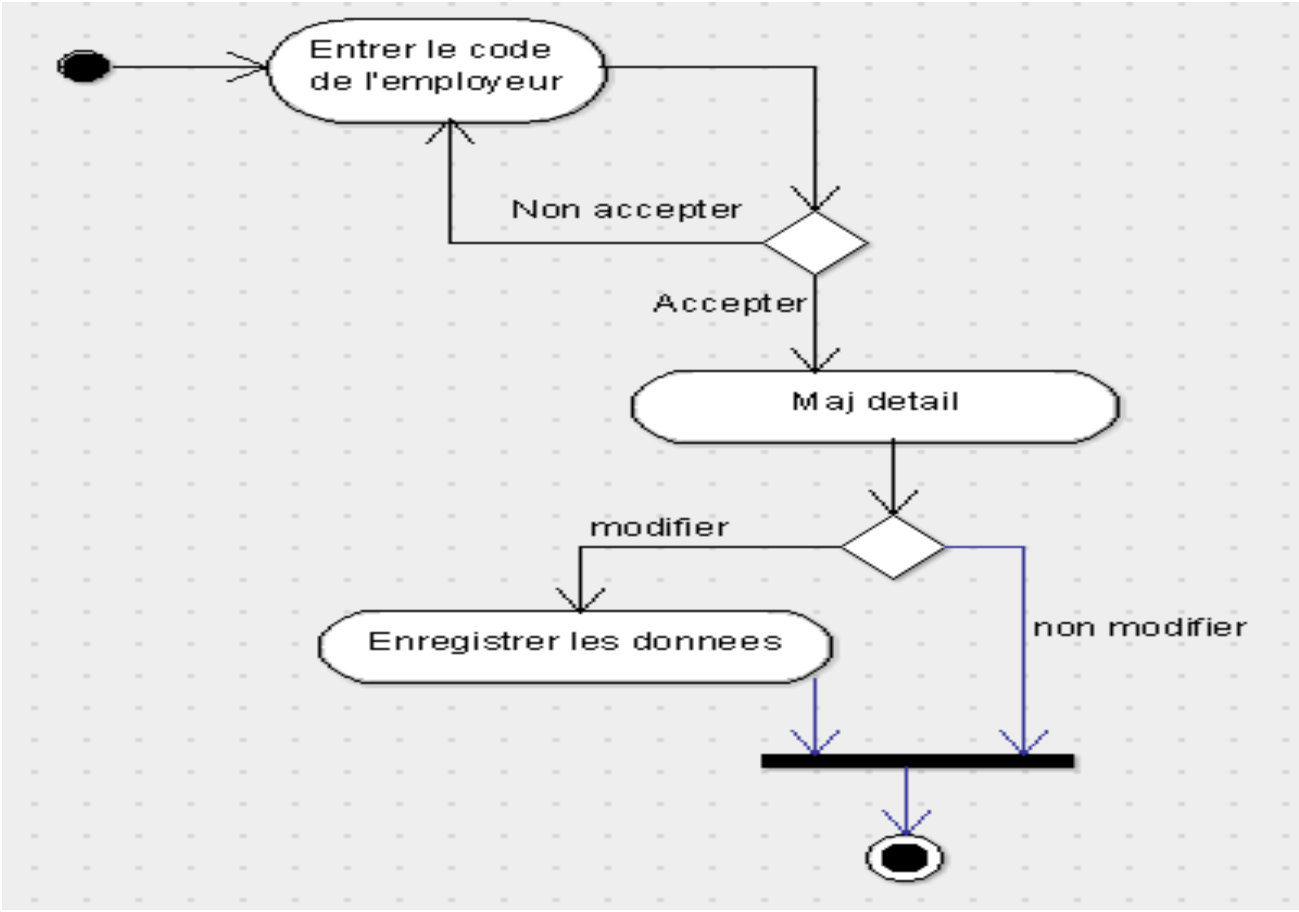


Fig51 - Exemple diagramme d'activité cohérent

Nous avons appliqués toutes les règles de cohérence qui existe dans le diagramme de cas d'utilisation et le diagramme d'activité qui donne un résultat positif.

Reste de vérifier les règles entre les deux digrammes.

- Règle R1 : Cette règle est satisfaite car le "Maj détail" de cas d'utilisation a un "Maj détail" dans le diagramme d'activité.

-Règle R2 : Cette règle est vérifiée car il y a un flux alternatif " Maj détail " dans le texte description du cas d'utilisation "Maj détail" et un seul nœud de décision correspondante dans le diagramme d'activité. En outre, le "Enregistrement les données "pour le cas d'utilisation a une activité «Enregistrement les données».

-Règle R3 : Cette règle est vérifiée car toute action prise par un acteur dans la description textuelle de cas l'utilisation cas "Majdetail" est représenté par une activité dans le diagramme d'activités correspondant. En fait, le «Voir status», «Marquer la présence » Activités.

-**Règle R4** : Cette règle est vérifiée car la première action" Entrer le code de l'employeur" effectuée par un acteur "Employeur" est représenté par la première activité «Entrer le code de l'employeur» liée avec le nœud initial dans le diagramme d'activité.

-**Règle R5** : Cette règle est vérifiée car la dernière action «Enregistrer les données» effectué par le système du scénario est représenté par la dernière activité «Enregistrement les données» de diagramme d'activité connecté au nœud final par un flux de contrôle.

1.16. CONCLUSION

Les résultats obtenus dans les expériences ont confirmé que l'utilisation d'OCL pour vérifier la cohérence du modèle en se basant sur les Méta-modèles UML représente une approche intéressante. Les modèles et les Méta-modèles utilisés dans ces exemples étaient des modèles réels construits à l'aide d'outils UML OCLE connus. En utilisant OCLE, nous avons réussi à détecter différents types d'incohérences qui ne peuvent être identifiées en utilisant d'autres approches.

Toutes les règles concernant la cohérence des modèles UML sont définies du méta-modèle. Par conséquent, ces règles sont indépendantes du modèle utilisateur, prise en charge de leur réutilisation pour tout modèle UML.

À notre avis, tous ces arguments sont des raisons logiques selon lesquels OCL est un langage formel et devrait devenir une norme de vérification et validation de facto.

CONCLUSION GENERALE

A travers ce mémoire, notre objectif était d'utiliser le langage formel OCL pour la vérification et la validation des diagrammes comportementaux d'UML 2.0.

Les notions théoriques des diagrammes UML et du langage OCL sont traités au niveau des premiers chapitres en particulier les diagrammes d'activités et des cas d'utilisations. Nous nous sommes penchés sur un ensemble de travaux connexes pour nous permettre la bonne analyse et la compréhension du problème. Cette analyse nous a permis la prise en charge de ce problème en développant des règles de cohérence en utilisant le langage OCL.

Dans le troisième chapitre nous avons montré que l'OCL est un langage formel permettant la vérification et la validation des modèles UML en se basant sur les méta-modèles du langage UML

Notre travail présenté au chapitre 4 consiste en l'élaboration des méta-modèles des diagrammes UML et le développement de règles de cohérences inter diagrammes pour la vérification et la validation des diagrammes UML

Notre perspective est d'utiliser cette stratégie pour la vérification et la validation des diagrammes UML en touchant toutes les vues UML et ainsi généraliser ce travail pour les diagrammes statiques et dynamiques.

BIBLIOGRAPHIE

[1] D. Chiorean , M. Paşca, A. Cârcu, C. Botiza, S. Moldovan, Ensuring UML models consistency using the OCL environment, Electron. Notes Theor. Comput. Sci. 102 (2004) 99–110.

[2] M. Richters, A Precise Approach to Validating UML Models and OCL Constraints,Citeseer, 2002.

[3] M. Richters, M. Gogolla, in: Validating UML models and OCL constraints, International Conference on the Unified Modeling Language, 2000, pp. 265–277.

[4] P. Ziemann, M. Gogolla, Validating OCL specifications with the use tool: an examplebased on the bart case study, Electron. Notes Theor. Comput. Sci. 80 (2003) 157–169.

[5] M. Gogolla, J. Bohling, M. Richters, Validating UML and OCL models in USE byautomatic snapshot generation, Softw. Syst. Model. 4 (2005) 386–398.

[6] M. Gogolla, F. B€uttner, M. Richters, USE: a UML-based specification environment for validating UML and OCL, Sci. Comput. Program. 69 (2007) 27–34.

[7] A. Shaikh, R. Clariso´ , U.K. Wiil, N. Memon, in: Verification-driven slicing of UML/OCL models, Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 185–194.

[8] M.W. Anwar, M. Rashid, F. Azam, M. Kashif, Model-based design verification for embedded systems through SVOCL: an OCL extension for SystemVerilog, Des. Autom. Embed. Syst. 21 (2017) 1–36.

[9] N. Przigoda, M. Soeken, R. Wille, R. Drechsler, Verifying the structure and behavior in UML/OCL models using satisfiability solvers, IET Cyber Phys. Syst. Theor. Appl. 1 (2016) 49–59.

[10] J. Cabot, R. Clariso´ , D. Riera, in: Verifying UML/OCL operation contracts, International Conference on Integrated Formal Methods, 2009, pp. 40–55.

[11] M. Benattou, J.-M.Bruel, N. Hameurlain, in: Generating test data from OCL specification, Proc. ECOOP Workshop Integration and Transformation of UML Models,2002.

[12] B.-L. Li, Z.-s. Li, L. Qing, Y.-H. Chen, in: Test case automate generation from UML sequence diagram and OCL expression, 2007 International Conference on Computational Intelligence and Security, 2007, pp. 1048–1052.

[13] A.D. Brucker, M.P. Krieger, D. Longuet, B. Wolff, in: A specification-based test case generation method for UML/OCL, International Conference on Model Driven Engineering Languages and Systems, 2010, pp. 334–348.

[14] Y. Cheon, C. Avila, in: Automating Java program testing using OCL and AspectJ, 2010 Seventh International Conference on Information Technology: New Generations (ITNG), 2010, pp. 1020–1025.

[15] L.C. Ascari, S.R. Vergilio, in: Mutation testing based on OCL specifications and aspect oriented programming, 2010 XXIX International Conference of the Chilean Computer Science Society (SCCC), 2010, pp. 43–50.

[16] OMG, <OMG Unified Modeling Lanuage TM-Superstructure Version 2.4.1 2011>

[17] Site internet <UML>http://www.uml.org.

[18] Laurent Audibert <*UML 2.0*> d'IUT de Villetaneuse département informatique VENUE Jean Baptiste Clément 9430 Villetaneuse.

[19] Delphine Longuet cas d'utilisation <UML> de Polytech Paris-Sud Formation initiale 3e année Spécialité Informatique site : http://www.lri.fr/~longuet/Enseignements/16-17/Et3-UML

[20] M. Elaasar and L. Briand.An Overview of UML Consistency Management. Technical report, DepartementOf Systems And Computer Engineering, Carleton University, August 2004. TR n◦ SCE-04-18.

[21] Meta Object Facility (MOF), OMG. version 2.0. Technical report, Object Management Group, 2003.

[22] UML 2.0 Superstructure Specification, OMG Document, ptc/03-08-02http://www.omg.org/ docs/ptc/03-08-02.pdf, 2003.

[23]. Ali M., Ben-Abdallah H., Gargouri F. "Validation des Besoins dans les Modèles UML2.0". Proceeding of INFORSID 2006: 959-974.

[24] Mouez ALI, Hanene BEN-ABDALLAH < Vérification formelle de la cohérence d'un modèle UML à base de relations de dépendance inter-diagrammes >Faïez GARGOURI Laboratoire MIRACL, FSEG - ISIM, Université de Sfax, BP 1030-3018, Sfax. TUNISIE.

[25] D.Sana Oueslati Ben Amor, D.Mouez Ali et D.FaiezGargouri. <Verification of the consistency between use case and activity diagrams> de MIRACL Laboratoty, ISIMS University of Sfax.Tunisia

[26]Shinkawa, Y. (2006). Inter-Model Consistency in UML Based on CPN Formalism. Paper presented at the 13th Asia Pacific Software Engineering Conference (APSEC '06) Kanpur.

[27] Sapna, P. G., &Mohanty, H. (2007, 17-20 Dec. 2007). Ensuring Consistency in Relational Repository of UML Models. Paper presented at the 10th International Conference on Information Technology (ICIT 2007).

[28] Chanda, J., Kanjilal, A., Sengupta, S., & Bhattacharya, S. (2009). Traceability of Requirements and Consistency Verification of UML UseCase, Activity and Class diagram: A Formal Approach. Paper presented at the International Conference on Methods and Models in Computer Science 2009 (ICM2CS), New Delhi, India.

[29] Chanda, J., Kanjilal, A., &Sengupta, S. (2010). UML-Compiler: A Framework for Syntactic and Semantic Verification of UML Diagrams (Vol. 5966, pp. 194-205): Springer-Verlag Berlin Heidelberg.

[30] Remco M. Dijkman, Stef M.M. Joosten<Deriving Use Case Diagrams from Business Process Models > University of Twente, Faculty of Computer Science

P.O. Box 217, 7500 AE Enschede et Ordina Finance Utopics, and Open University of the NetherlandsP.O. Box 2960, 6401 DL HeerlenThe Netherlands
[31] OMG – UnifiedModelingLanguage, v1.4 <UML semantics> de www.omg.orgSeptember 2001
[32]Hugues Malgouyres, Jean-Pierre Seuma-Vidal,Gilles Motet <*Règles de Cohérence UML2.0*> de Laboratoire ESIA Institut national des sciences appliquées – Toulouse
[33] Mouez ALI, Faiez GARGOURI <Verification of the Consistency between Use Case and Activity Diagrams – AStep Towards Validation of User Requirements.>See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/220710693.
[34] Atif A. Jilani, Muhammad Z. Iqbal, Muhammad U. Khan, Muhammad Usman<Advances in Applications of Object Constraint Language for Software Engineering >Quest Laboratory, FAST-National University of Computer & Emerging Sciences, Islamabad, Pakistan
[35] Jean-paulBodeveix, Thierry Millan, Christian Percebois<*Vérification de modèles UML fondée sur OCL*> de Conférence PaperJanuary 2003
[36] Cockburn, A. (2001). Rédiger des cas d'utilisations efficaces, Paris : Eyrolles. (voir aussi :http://alistair.cockburn.us/)
[37] Site SparxSystems<UML 2 Tutoriel -Diagrammes d'Activité>https://www.sparxsystems.fr/resources/uml2_tutorial/uml2_activitydiagram.html
© 2000 - 2020 SparxSystemsPty Ltd
[38] Pascal André — Gilles Ardourel — Gerson Sunyé<Un cadre pour la vérification demodèles UML>LINA - Université de Nantes2, rue de la Houssinière - BP 9220844322 Nantes Cedex 03
[39] Outils de verification<objectconstraintlanguage environnement –OCLE2.0.4> http://lci.cs.ubbcluj.ro/ocle/
[40] Claude Belleil Université de Nantes <Le langage UML 2.0 OCL (Object Constraint Language)
[41] UML 2.0 OCL Specification, OMG Final Adopted Specification ptc/03-10-14 site:http://www.omg.org/