République Algérienne Démocratique et Populaire
Ministère l'enseignement supérieur et la recherche scientifique
UNIVERSITE ABBES LAGHROUR – KHENCHELA

FACULTE DES SCIENCES ET DE LA TECHNOLOGIE

DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUES

mémoire de fin d'étude pour l'obtenir de diplôme

De master informatique

Spécialité :Génie logicielle système distribuée

Thème

Vers un profil UML pour la vérification des diagrammes d'uml 2.5

Présenté Par : Mahrez Karim

Saighi Aymen

Dirigé Par : Dr Messoudi Nabil

2021/2020

Remerciements

Nous tenons à remercier très chaleureusement Dr Messoudi Nabil, notre encadreur de thèse pour ses conseils, ses encouragements et sa confiance. Nous le remercie aussi pour sa patience, sa bienveillance durant toutes cette année de thèse et son soutien.

Nous remercions les honorables membres de jury d'avoir accepté d'être membre de jury de mémoire, d'évaluer notre travail..

Nous adressons également nos remerciements à nos amis et nos collègues et nos professeurs de l' université abbés laghrour qui nous ont soutenus pendant toutes ces années, en particulier le Dr Mehdaoui Rafiq, le Dr chouhel Wahiba et Benaroua Anouar.

Enfin, c'est l'occasion pour adresser nos remerciements à nos parents pour le soutien inconditionnel qui nous a été apporté au cours des années de nos études supérieures. Leur appui nous a été très précieux et nous leur en témoigne aujourd'hui notre plus grande reconnaissance. nous remercions nos frères pour leurs soutien moral.

Abstract

The Unified Modelling Language (UML), is the first step of developing an object-oriented design method and a standard notation for the modelling of real-world objects and it consists of fourteen various diagram types. The complex relationships between UML diagrams will lead to the inconsistencies among the unified modelling language (UML) diagrams therefore, it is so important to verify model in the early phase before implement it because early error detection will be easier to fix than later phase.

This memory focuses on in what the way to determine and identify inconsistencies between unified modelling language diagrams. Many of consistency rules will be applied to check the consistency models between mostly used categories of unified modelling language diagrams in the field of systems design and systems analysis. As follows state of these diagrams: (Use Case Diagram, Class Diagram , State machine Diagram , Communication Diagram, and Sequence Diagram).

To check for inconsistencies between Unified Modeling Language diagrams, we have researched and given consistency rules and applied them using the design pattern decorator on which we built our work.

Résumé

Le langage de modélisation unifié (UML) est la première étape du développement d'une méthode de conception orientée objet et d'une notation standard pour la modélisation d'objets du monde réel et se compose de quatorze types de diagrammes différents. Les relations complexes entre les diagrammes UML conduiront à des incohérences parmi les diagrammes d'UML. Par conséquent, il est si important de vérifier le modèle au début de la phase avant de l'implémenter, car la détection précoce des erreurs sera plus facile à corriger que la phase ultérieure .

Cet mémoire se concentre sur la manière de déterminer et d'identifier les incohérences entre les diagrammes de langage de modélisation unifié. De nombreuses règles de cohérence seront appliquées pour vérifier les modèles de cohérence entre les catégories les plus utilisées de diagrammes UML dans le domaine de la conception et de l'analyse des systèmes.

Pour vérifier les incohérences entre les diagrammes du langage de modélisation unifié, nous avons recherché et donné des règles de cohérence et les avons appliquées a l'aide du décorateur design pattern sur lequel nous avons construit notre travail.

الملخص

لغة النمذجة الموحدة ، هي الخطوة الأولى لتطوير طريقة تصميم موجهة للكائنات وتدوين قياسي لنمذجة كائنات العالم الحقيقي وتتكون من أربعة عشر نوعًا مختلفًا من الرسوم البيانية. ستؤدي العلاقات المعقدة بين المخططات إلى وجود تناقضات بين الرسوم التخطيطية للغة النمذجة الموحدة ، لذلك من المهم للغاية التحقق من النموذج في المرحلة المبكرة قبل تنفيذه لأن اكتشاف الخطأ المبكر سيكون أسهل في الإصلاح من المرحلة اللاحقة .

تركز هذه المذكرة على كيفية تبيان وتحديد التناقضات بين مخططات لغة النمذجة الموحدة. سيتم تطبيق العديد من قواعد التناسق للتحقق من نماذج التناسق بين الفئات المستخدمة في الغالب لمخططات لغة النمذجة الموحدة في مجال تصميم الأنظمة وتحليل الأنظمة. كما يلي حالة هذه المخططات: (استخدم مخطط الحالة ، ومخطط الفصل ، ومخطط آلة الحالة ، ومخطط الاتصال ، ومخطط التسلسل.

للتحقق من عدم الاتساق بين الرسوم البيانية للغة النمذجة الموحدة ، قمنا بالبحث وإعطاء قواعد التناسق وتطبيقها باستخدام مصمم نمط التصميم الذي بنينا عليه عملنا.

Table Des matières

Table Des Figures

Liste des Tableaux

Introduction Générale

Le lancement d'UML [1] a ouvert une nouvelle voie pour la conception d'applications orienté objet [1]. La norme UML d'Object Management Group (OMG) [1] contient un ensemble de diagrammes utiles pour exprimer les propriétés statiques et dynamiques d'une application orienté objet en phase de conception [2]. La traduction automatique des modèles UML en code minimise le nombre d'erreurs et génère un code orienté objet plus conforme et plus fiable que la traduction manuelle. Il est donc important d'avoir un UML cohérent modèles pour obtenir le code orienté objet conforme. Cela nécessite de se concentrer pleinement sur la cohérence des modèles UML, par exemple si un diagramme de séquence UML appelle une méthode sur un objet d'une classe, alors la signature de méthode pour cette méthode doit exister dans le diagramme de classes UML dans cette classe spécifique [1]. La cohérence des modèles UML est également importante car les modèles UML incohérents entraînent une génération de code OO inexacte. La validation de la cohérence entre les modèles UML est utile car il est plus difficile de modifier le code source que dans les modèles UML. Ainsi, chaque fois que les modèles UML sont modifiés, il est très important de s'assurer que les modèles UML sont toujours cohérents après les modifications. La validation de la cohérence entre les modèles UML aide également les éditeurs de logiciels sur le plan financier en minimisant le coût pendant le logiciel processus de développement .Dans cet mémoire , nous présentons les techniques de vérification de cohérence entre UML des modèles. La plupart les techniques se concentrent sur la cohérence inter et intra niveau pour une validation entre modèles UML. Presque toutes les techniques de contrôle de cohérence se basent sur des règles de cohérence pour valider les modèles UML.

CHAPITRE

1

Introduction

Sommaire

1.1 Introduction

Aujourd'hui les systèmes informatiques simples ou complexes jouent un rôle important dans plusieurs domaines d'applications (par exemple la médecine, l'architecture, les mathématiques, etc...). La modélisation des systèmes est une phase laborieuse pour la conception et la validation des systèmes.

Dans notre thème on s'intéresse à la modélisation.et Dans ce contexte, plusieurs langages de modélisation existent dans la littérature ; par exemple UML (Unified modeling langage), MERISE. Dans notre travail, nous avons appliqué la modélisation en utilisant les diagrammes UML 2.5.

1.2 Uml

1.2.1 Définition UML

UML(Unified modeling language) [1] est un langage de modélisation unifié reconnu actuellement par toute la communauté des chercheurs de l'orienté objet. Cependant un certain nombre de risques peuvent survenir en raison de son inadéquation technique et fonctionnelle du système par rapport aux besoins préalablement définis dans le cahier des charges.

Lorsque l'expression des besoins part comme il se doit d'une action, elle comporte naturellement la description du processus. Toute action se ramène, en effet, à un processus selon lequel, partant d'une situation initiale et par l'application de certaines ressources, on aboutit à un résultat.

La modélisation se construit forcément par étapes successives de plus en plus détaillées. Il est, par conséquent, impossible de produire un modèle représentant quelques milliers de lignes de code sans passer par les phases d'avancement, qui permettent d'organiser et d'exploiter judicieusement le volume d'informations collectées d'où l'intégration de la fouille de données.

1.2.2 Point Fort d'UML [3]

- UML est un langage formel et normalisé :

  Il permet ainsi :

  • un gain de précision

  • un gage de stabilité

  • l'utilisation d'outils

- UML est un support de communication performant :

  Il cadre l'analyse et facilite la compréhension de représentations abstraites complexes. Son caractère polyvalent et sa souplesse en font un langage universel.

1.2.3 Point Faible d'UML [3]

- Apprentissage et période d'adaptation

  Même si l'Espéranto est une utopie, la nécessité de s'accorder sur des modes d'expression communs est vitale en informatique. UML n 'est pas à l'origine des concepts objets, mais en constitue une étape majeure, car il unifie les différentes approches et en donne une définition plus formelle

- Le processus (non couvert par UML)

  L'intégration d'UML dans un processus n'est pas triviale et améliorer un processus est un tâche complexe et longue.

1.3 Les Diagrammes UML

1.3.1 Définition:

Un diagramme UML [1] est une représentation graphique, qui s'intéresse à un aspect précis du modèle. C'est une perspective du modèle, pas "le modèle".

Chaque type de diagramme UML possède une structure (les types des éléments de modélisation qui le composent sont prédéfinis).

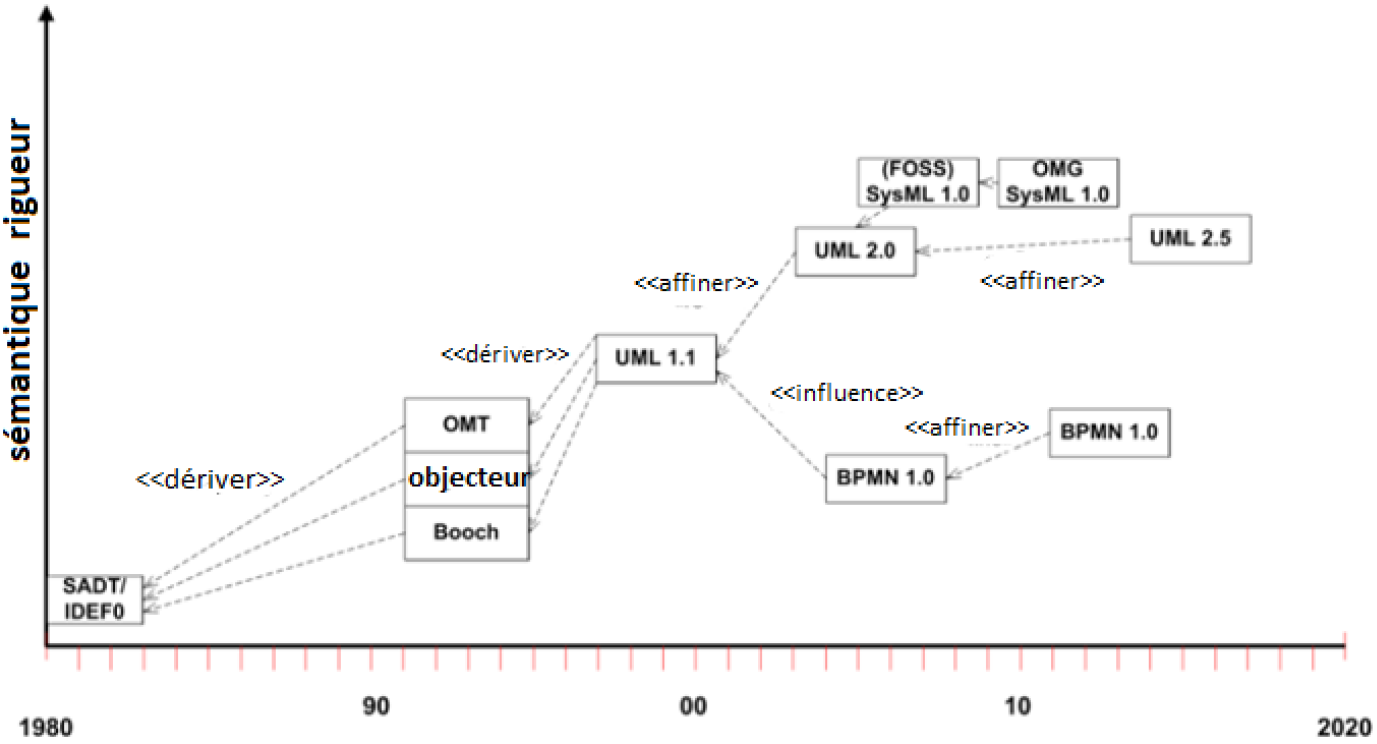

Figure 1.1 – évolution du langage de modélisation d'uml [1]

1.3.2 Caractéristiques fondamentales des modèles

Le caractère abstrait d'un modèle doit notamment permettre :

- de faciliter la compréhension du système étudié : un modèle réduit la complexité du système étudié.

- de simuler le système étudié : un modèle représente le système étudié et reproduit ses comportements

1.3.3 Les différents types de diagrammes UML :

UML 2.5 propose quatorze types de diagrammes pour représenter les déférents points de vues distinctes pour modéliser des concepts particuliers d'un système[1]. Ils se répartissent en deux grands groupes :

1. Diagrammes structurels ou statiques d'un système (Structure Diagram).

2. Diagrammes comportementaux ou dynamiques d'un système (Behavior Diagram).

1.3.3.1 Les Diagrammes structurels ou statiques d'un système [1]

- **Les diagrammes d'objets** : Un diagramme d'objets est l'un des diagrammes UML 2.5 utilisé pour représenter l'aspect statique d'un système. On dit aussi diagramme d'instance. Il est utile pour explorer des exemples du monde réel des objets et les relations entre eux. Il montre des instances au lieu de classes. Ils sont utiles pour expliquer les petits morceaux avec des relations compliquées, en particulier des relations récursives.
- **Les diagrammes de composants** : Un diagramme de composants est l'un des diagrammes UML 2.5 utilisé pour représenter l'aspect statique d'un système. Ils montrent les dépendances entre les composants logiciels, y compris les classificateurs et les objets précis de mise en œuvre ; tels que les fichiers de code source, fichiers de code binaires, les fichiers exécutables, des scripts et des tables
- **Les diagrammes de déploiements** : Un diagramme de déploiements représente une vue statique de la configuration d'exécution de nœuds de matériel et les composants logiciels qui s'exécutent sur ces nœuds. Les diagrammes de déploiement montrent le matériel de votre système, le logiciel qui est installé sur ce matériel, et le middleware utilisé pour connecter les machines disparates les uns aux autres.

- **Les diagrammes de paquetages** : c'est une représentation simplifiée des diagrammes de classes complexes, ils peuvent regrouper les classes en paquets. Un paquet est une collection d'éléments UML logiquement liés. Les paquets sont définis comme des dossiers de fichiers et peuvent être utilisés sur n'importe lequel des diagrammes UML 2.5.

- **Les Diagrammes de structures composites :** Les Diagrammes de structures composites sont utilisés pour explorer les instances d'exécution des instances interconnectées collaboratrices sur des liaisons de communication. Ils montrent la structure interne (y compris les pièces et les connecteurs) d'un classificateur ou d'une collaboration structurée.

- **Les Diagrammes de profils** : est un diagramme de structure permettant l'utilisation de profils pour un métamodèle donné. Apparu avec UML 2.5, ce diagramme fournit une représentation des concepts utilisés dans la définition des profils (packages, stéréotypes, application de profils , etc.).
- **Les diagrammes de classes** : Un diagramme de classe est l'un des diagrammes UML 2.5 utilisé pour représenter l'aspect statique d'un système. Il est composé d'un ensemble de classes et d'associations entre elles. La classe est représentée par un rectangle contenant le nom, les attributs et les opérations. Les associations peuvent être simples, multiples, avec attributs, composition, agrégation et héritage.

1.3.3.2 Les Diagrammes comportementaux ou dynamiques d'un système [1]

- **Les diagrammes de séquence :** Les diagrammes de séquence montrent la collaboration des objets basée sur une séquence de temps. Ils montrent comment les objets interagissent avec les autres dans un scénario particulier d'un cas d'utilisation.
- **Les diagrammes d'états transitions :** Les diagrammes d'états transitions peuvent montrer les différents états d'une entité aussi comment une entité répond à divers événements par le passage d'un état à un autre. L'histoire d'une entité peut être mieux modélisée par un diagramme d'états finis.
- **Les Diagrammes de temps** : Les diagrammes de temps montrent le comportement des objets dans une période de temps donnée. Le chronogramme est une forme particulière d'un diagramme de séquence. Les différences entre un diagramme de temps et un diagramme de séquence sont les axes qui sont inversés de sorte que l'augmentation du temps est de gauche à droite et les lignes de vie sont présentées dans des compartiments séparés disposés verticalement.
- **Les diagrammes de cas d'utilisation** : Les diagrammes de cas d'utilisation décrivent le comportement du système cible à partir d'un point de vue externe. Les cas d'utilisation décrivent des besoins réels. Un cas d'utilisation décrit une séquence d'actions qui offre quelque chose de valeur mesurable pour un acteur et est dessinée comme une ellipse horizontale. Un acteur est une personne, une organisation ou un système externe

qui joue un rôle dans un ou plusieurs interactions avec votre système. Les acteurs sont dessinés comme des chiffres de bâton. Les associations entre les acteurs et les cas d'utilisation sont indiqués par des lignes solides. Une association existe chaque fois qu'un acteur est impliqué dans une interaction décrite par un cas d'utilisation.

- **Un diagramme de vue d'ensemble d'interaction :** est un diagramme d'interaction qui se concentre sur la vue d'ensemble du flux de contrôle , Ce type de diagramme ressemble à un diagramme d'activité, dans le sens où les interactions et/ou les occurrences d'interaction servent de nœuds d'activité. Tous les symboles qui apparaissent sur des diagrammes de séquence et des diagrammes d'activité peuvent apparaître sur des diagrammes de vue d'ensemble d'interaction**.**
- **Les diagrammes d'activités** : Un diagramme d'activités est l'un des diagrammes UML 2.5 utilisés pour représenter l'aspect comportement (fonctionnelle) d'un système. Il est composé d'un ensemble d'activités avec état initial et un état final. ils sont reliés par des transitions simple, conditionnés, synchrones ou par l'objet d'information. Les activités peuvent être des activités simples ou bien des activités spéciaux (par exemple un événement accepté, signal ou un événement d'attente). On trouve aussi les couloirs d'activités qui sont un sous-diagramme. Chaque diagramme d'activités est composé de plusieurs couloirs (sous- diagrammes) dont chacun possède un nom. Les diagrammes d'activités sont une variante des diagrammes d'états-transitions dans laquelle les états représentent à la fois la réalisation des actions ou sous activités et les transitions déclenchées par la finalisation des actions ou sous activités.

- **Les diagrammes de collaboration** : Les diagrammes de collaboration montrent les interactions entre les objets à travers la représentation chronologique d'envois de messages, mais le temps n'est pas représenté implicitement. La chronologie des interactions est indiquée par la numérotation de messages pour indiquer leur ordre d'envoi. Ce diagramme est équivalent au diagramme de séquences. Cependant, l'aspect temporel n'apparaît pas, mais l'aspect chronologique est présent.

- **Les diagrammes de communication** : Un diagramme de communication est l'un des diagrammes UML 2.5 utilisés pour représenter l'aspect comportement (communication) d'un système, c'est une représentation simplifiée de diagrammes de séquences. En plus, on peut construire l'un à partir de l'autre. Un diagramme de communication est une combinaison des diagrammes classes, séquences, cas de

utilisation et objet. Il est composé d'un ensemble d'objets qui communiquent entre eux par des messages. Un lien représente une liaison entre deux objets (émetteur, récepteur) avec un message transitant entre l'émetteur et le récepteur. Un message peut être synchrone, asynchrone, réponse, ou perdu. Leur représentation est sous forme d'un graphe dans lequel les nœuds représentent des objets et les arcs représentent les échanges de messages entre les objets.

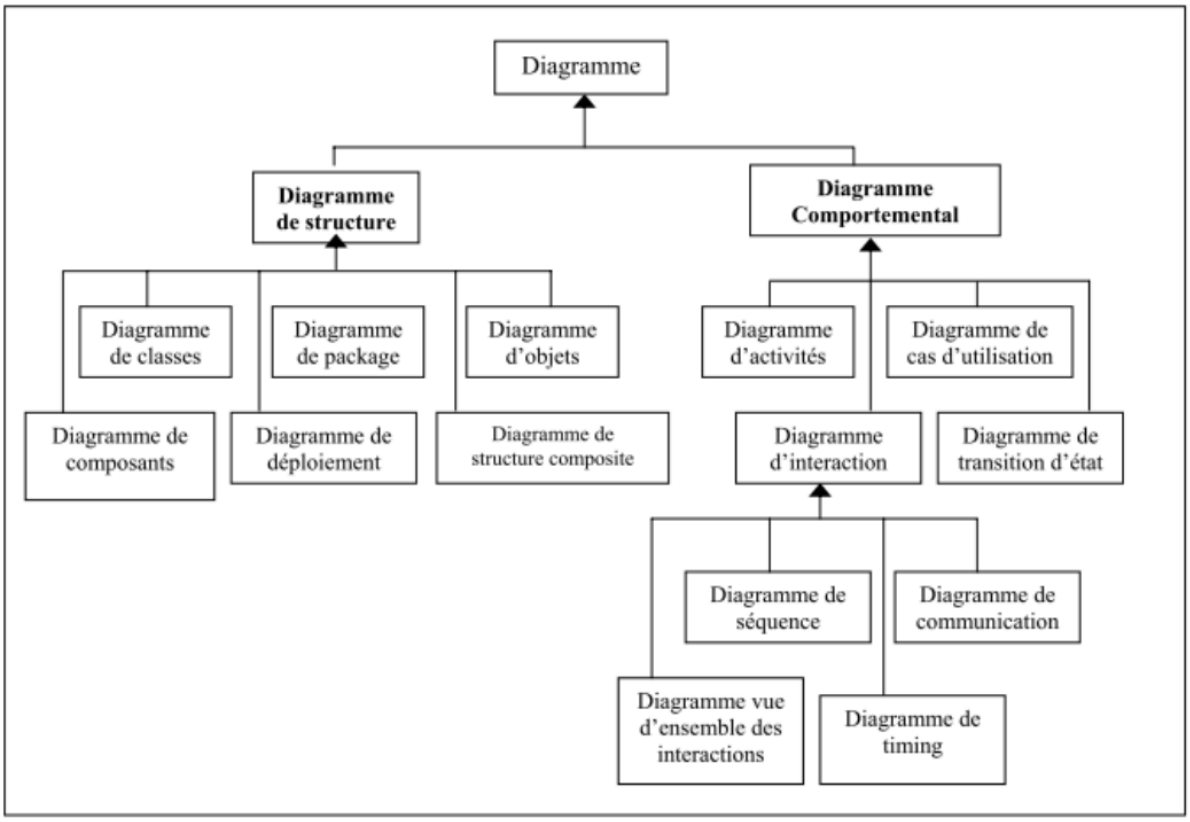

Figure 1.2 – Les Diagrammes Uml [1]

1.4 OCL : Object Constraint Language

1.4.1 : définition

Ocl est un langage informatique d'expression des contraintes utilisé par UML. Développé pour la première fois par IBM en 1995 puis standardisé par l'Object Management Group, OCL permet d'effectuer des requêtes sur des métamodèles.

Ce langage formel est volontairement simple d'accès et représente un juste milieu entre langage naturel et langage mathématique. Il permet ainsi de limiter les ambiguïtés dans la spécification des contraintes logicielles. Sa grammaire simple lui permet d'être interprété par des outils logiciels pour faire de la programmation par contrat et vérifier qu'un logiciel répond à ses spécifications techniques. [2]

1.4.2 : Normalisation de OCL :

- OCL version 1 : fait partie de la norme UML 1.X de l'OMG (Object Management Group)
- OCL version 2.0 : spécification conforme à UML 2.0, mais norme à part entière de l'OMG pouvant s'appliquer sur d'autres modèles que ceux de UML.

1.4.3 :Caractéristiques , forces et faiblesses de OCL [2]

- **Caractéristiques de OCL :**
  - langage d'expression sous ensemble des langages fonctionnels
  - basé sur la théorie des ensembles et la logique des prédicats,
  - repose sur le concept de valeur et d'expression
  - sans effet de bord et fortement typé
  - permet de spécifier, pas de programmer
  - extensible

- **Forces :**
  - d'être parfaitement adapté et intégré à UML
  - notation dite intuitive et proche des langages de programmation, moin rebutante qu'une notation mathématique

- **Faiblesses :**
  - pas toujours défini très proprement
  - pas vraiment lisible pour des contraintes complexes

- pas aussi rigoureux qu'un langage de spécification comme Z ou B => pas de preuves possibles
- puissance d'expression limitée

1.4.4 : Utilisation de OCL dans les diagrammes Uml [2]

OCL permet de spécifier des contraintes ou des expressions associés à un diagramme UML

Contraintes pour :

- **contraindre** les **diagrammes UML**
- **définir** des **invariants** de classe, **précondition** de méthode, ...
- vérifier potentiellement ces contraintes à l'exécution

Expressions pour :

- **indiquer** des **valeurs** ou des collections de valeurs
- **décrire** des **postconditions**
- **spécifier des opérations** : par exemple de type **Query** (OCL se rapproche alors de SQL : requêtes portant sur un diagramme UML)
- **générer éventuellement du code** correspondant à l'évaluation d'une expression

une expression OCL est toujours associée à un élément d'un diagramme : c'est le **contexte de la contrainte** (context)

Une expression OCL permet de spécifier :

- des **Invariants** sur des classes (inv)
- des **Préconditions** sur des **opérations** (pre)
- des **Postconditions** sur des **opérations** (post)
- des **Gardes** sur **transitions** de diagrammes d'états ou de **messages** de diagrammes de
- séquence/collaboration
- des **ensembles d'objets destinataires** pour un envoi de message
- des **attributs dérivés**
- des **stéréotypes**

1.4.5 Types et syntaxe générale des expressions OCL [2]

Différents types :

- **Types provenant d'UML :** classes, association, ...
- **Types de base** : Integer, real, Boolean, string
- **Types énumérés** : {masculin, féminin}
- **Types construits ou Collection**: set (T), séquence (T), bag(T)
- **Méta-types** : OclType, OclAny, OclState, OclExpression

Syntaxe générale des expressions OCL :

- Constantes
- Identificateurs
- **self**
- expr op expr
- exprobjet . propobjet (paramètres)
- exprcollection -> propcollection (paramètres)
- package::package::element
- if cond then expr else expr endif
- let var : type in expr, ...

1.5 Profiles Uml

1.5.1 Définition

Un profil est un package stéréotypé [2] qui contient des éléments de modèle personnalisés pour un domaine ou un objectif spécifique à l'aide de stéréotypes, de définitions de balises et de contraintes, y compris tout autre support ou requis éléments de modélisation. Un profil affine la sémantique standard en ajoutant des contraintes et interprétations supplémentaires qui capturent la sémantique et les éléments de modélisation spécifiques au domaine, mais il n'ajoute pas de nouveaux concepts fondamentaux.

1.5.2 Stéréotype

Les stéréotypes permettent aux concepteurs d'étendre les éléments des diagrammes d'UML [2]. Un stéréotype introduit une nouvelle classe dans le méta-modèle par dérivation d'une classe existante. Comme le montre la figure 1.3, un stéréotype est rendu comme un nom encadré par des guillemets <<>>et placé au-dessus du nom de l'élément stéréotypé

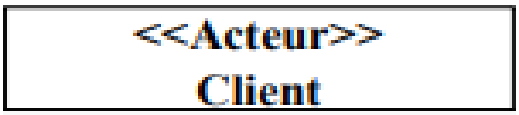

Figure 1.3 – Stéréotype [2]

1.5.3 Etiquette (valeur marquée)

Etiquette (valeur marquée) Une valeur marquée est un méta-attribut que l'on définit comme attribut d'un stéréotype. Elle ajoute une nouvelle propriété à un élément de modélisation. Une valeur marquée a un nom et un type indiqué entre accolades.

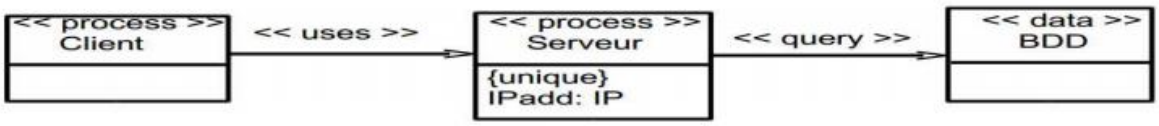

Figure 1.4 : valeur marquée [5]

1.5.4 : Une Contrainte dans les Profiles

Dans UML2.5, une contrainte ajoute une sémantique à un stéréotype. Pour exprimer des contraintes de manière formelle, le langage de contraintes OCL (Object Constraint Language) peut être utilisé. Chaque contrainte est indiquée entre accolades {}. Une contrainte peut être associée à plusieurs éléments par des relations de dépendance [5]. La figure 1.5 présente un exemple d'une contrainte associée au stéréotype Bean.

La figure 1.5 définit un profil UML spécifiant une architecture EJB (Entreprise JavaBeans). Ce profil regroupe les cinq stéréotypes (Bean, Session, Entité, Jar, Eloigné, Accueil). Les flèches noires représentent des extensions. Enumération est un type de données dont les instances proviennent d'une liste de valeurs littérales nommées.

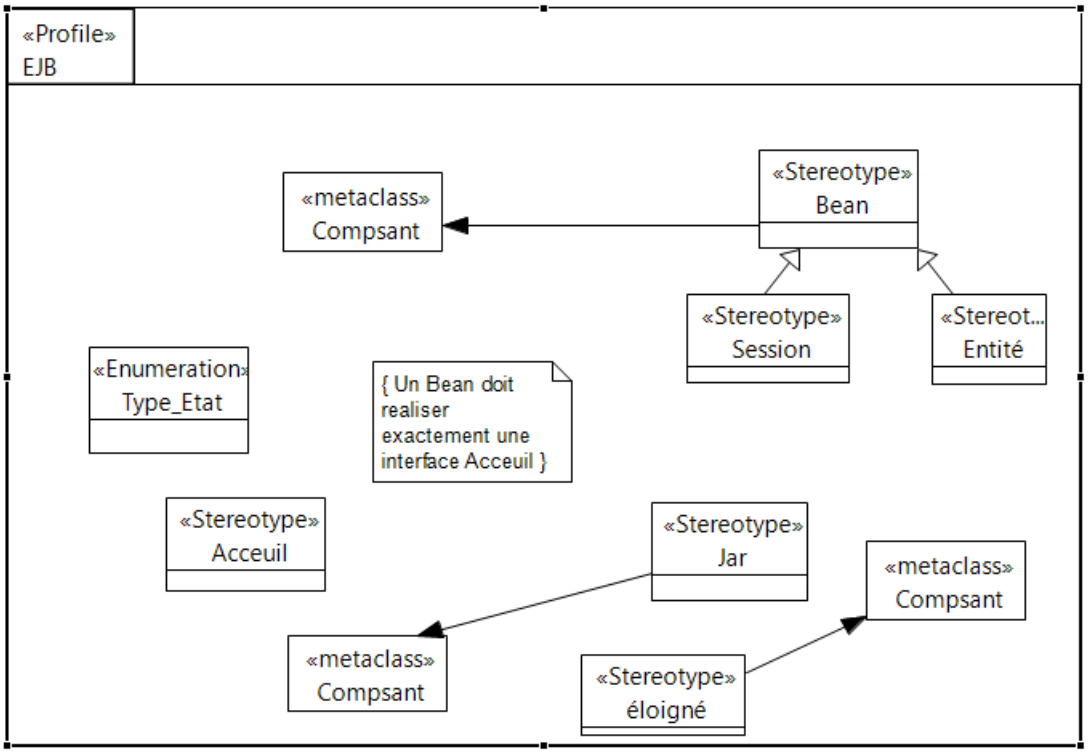

Figure 1.5 – un exemple du profil UML qui utilise une contrainte OCL [6]

1.6 Conclusion

Dans ce chapitre nous avons présenté brièvement la définition de le langage de modélisation unité UML 2.5 avec ces diagrammes. Par la suite nous avons cité quelques avantages et inconvénients de la modélisation orientée objet .en suite nous avons présenté les profils uml et langage de contraints OCL , Le chapitre suivant va introduire la notion de la cohérence uml qui permet de donnez-nous des points sur la Cohérence Uml .

CHAPITRE

2

La Cohérence Uml

Sommaire

2.1 Introduction

Le langage de modélisation unifié (UML) est devenu une norme acceptée par l'industrie pour la modélisation orientée objet de grands systèmes complexes ainsi qu'une base pour les méthodologies de développement de logiciels. Au cours du processus de développement, des artefacts représentant différents aspects du système sont produits. Les artefacts doivent être correctement liés les uns aux autres afin de former une description cohérente du système développé. En particulier, deux types de problèmes concernant la cohérence sont présentés - ceux liés à la cohérence entre les diagrammes au sein d'un modèle donné et nommés comme un problème d'intra-cohérence et ceux concernant la cohérence entre différents modèles et nommés comme un problème d'inter-cohérence.

2.2 Définition

La cohérence est la situation où deux ou plusieurs éléments se chevauchant de différents diagrammes qui décrivent le comportement du système sont conjointement satisfiables. C'est l'un des attributs pour mesurer la qualité du modèle UML. [4]

2.3 Cohérence dans UML

Selon Genero et ses collègues [27], la cohérence UML est la qualité UML qui a reçu le plus d'attention (entre autres) par les chercheurs en termes de nombre d'articles (conférence, revue, etc.) (voir Figure 3)

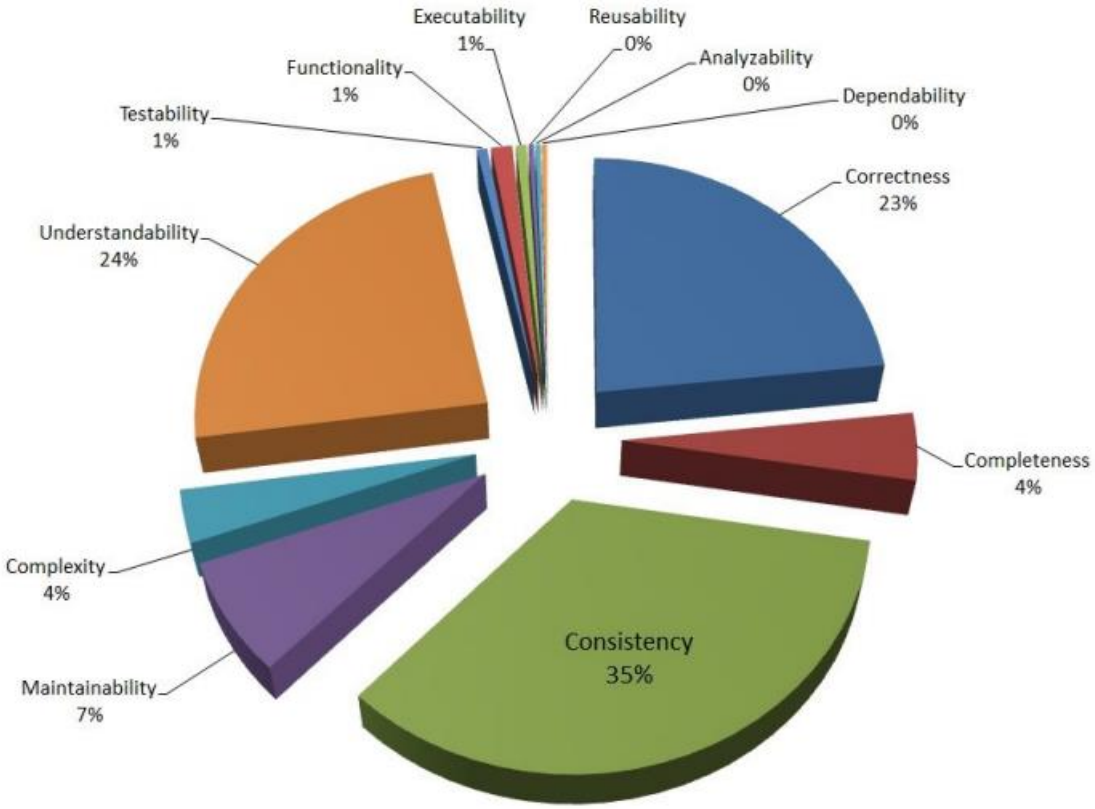

Figure 2.1 – UML qualités[27]

Comme UML n'est pas une notation formelle, des incohérences peuvent survenir dans la spécification UML d'un système complexe lorsqu'une telle spécification nécessite plusieurs diagrammes/vues pour décrire différents perspectives du logiciel/modèle [38]. Dans ce mémoire, nous nous référons aux diagrammes UML comme les différentes vues d'un logiciel (c'est-à-dire un modèle) en cours de développement. Quand les diagrammes UML dépeignent des contradictions ou de sens contradictoire, les diagrammes sont dits incohérents [45]. De telles incohérences peuvent être une source de défauts dans les systèmes logiciels [36, 30]. Il est donc primordial qu'elles soient détectées, analysées et fixées [28].

Même si de nombreux chercheurs ont proposé, explicitement ou non, des règles pour détecter les incohérences, aucun ensemble de règles de cohérence bien accepté et aussi complet que possible n'a été décrit et publié jusqu'à présent. Bien que le standard UML fourni par l'OMG lui-même contienne certaines règles de cohérence, souvent désignées dans les versions précédentes d'UML comme des règles de bonne formation, le standard n'offre pas une liste complète car, par exemple, certaines règles de cohérence peuvent être spécifiques à la façon dont la notation UML est utilisée.

Ce manque de liste bien acceptée de règles de cohérence UML oblige les chercheurs à définir systématiquement les règles de cohérence sur lesquelles ils s'appuient pour leurs propres recherches [38].

2.4 Types de cohérence

il existe cinq principaux types de cohérence définis par Mens et al [44] (voir tableau 2)

Cohérence verticale (inter-modèle)	La cohérence est validée à différents niveaux d'abstraction entre différents diagrammes. Des cohérences syntaxique et sémantique y sont également incluses.
Cohérence horizontale (intra-modèle)	La cohérence est validée à un même niveau d'abstraction entre différents diagrammes.
Cohérence de l'évolution	La cohérence est validée entre différentes versions d'un même diagramme UML.
Cohérence sémantique	La cohérence est validée pour les significations sémantiques des diagrammes UML définies par le méta-modèle UML.
Cohérence syntaxique	La cohérence est validée pour les spécifications des diagrammes UML dans le méta modèle UML.

Tableau 1– Types de cohérence

2.5 Classification de cohérence UML

De nombreuses propositions dans la littérature donnent différentes classifications des incohérences en UML, Dans cette section, toutes les principales classifications seront décrites. le tableau suivant résume ces différentes classifications

Classification	Caractéristiques
Syntaxique vs Sémantique	Des règles de cohérence qui peuvent être exprimées par un langage sont syntaxiques, sinon ils sont sémantiques
Statique vs Dynamique	Règles de cohérence vérifiables sans exécution un modèle sont statiques, sinon ils sont dynamiques
Intra-Model vs. Inter-Model	Les règles de cohérence au sein d'un même modèle sont intra modèle. Ceux qui couvrent les modèles sont inter-modèles
Multi-niveaux	Les règles de cohérence sont regroupées selon la sémantique domaine qu'ils ciblent (cahier des charges, profils, modélisation processus, domaine modélisé …etc)
Nature of Error	Les règles de cohérence sont regroupées en fonction de la nature des erreur (contradiction, incomplétude, ambiguïté…etc)

Tableau 2- Classification de cohérence UML

2.5.1 Syntaxique vs Sémantique

Les travaux publiés sur la cohérence UML ont tendance à aller dans des directions différentes selon la définition utilisée de la cohérence. Cependant, tous conviennent que la cohérence en général implique un manque de contradiction et de conformité aux attentes. Les spécifications du langage [8] introduisent deux premiers niveaux de cohérence, le méta-modèle et les règles de bonne formation. Le méta-modèle est un schéma qui définit précisément les constructions et les règles nécessaires à la création de modèles. Par exemple, une association peut avoir deux ou plusieurs extrémités d'association. La figure 2.2 représente un sous-ensemble (le diagramme de classes) du méta-modèle UML.

Un modèle est incohérent s'il n'est pas conforme au méta-modèle. Cependant, il peut ne pas être cohérent même s'il est conforme au méta-modèle. Ceci est principalement dû à l'expressivité limitée du métalangage UML, un formalisme utilisé pour définir le métamodèle UML. Dans un effort pour compléter le métalangage, l'OMG a proposé OCL [9], un langage logique d'ordre supérieur pour désigner les contraintes de bonne forme en UML. OCL, un langage d'expression pur, a des constructions pour inspecter et parcourir les objets et leur structure et retourner une valeur vraie ou fausse mais cela ne change pas le modèle. La bonne forme, telle qu'exprimée par les contraintes OCL, est généralement une condition préalable à toute autre analyse de cohérence en UML.

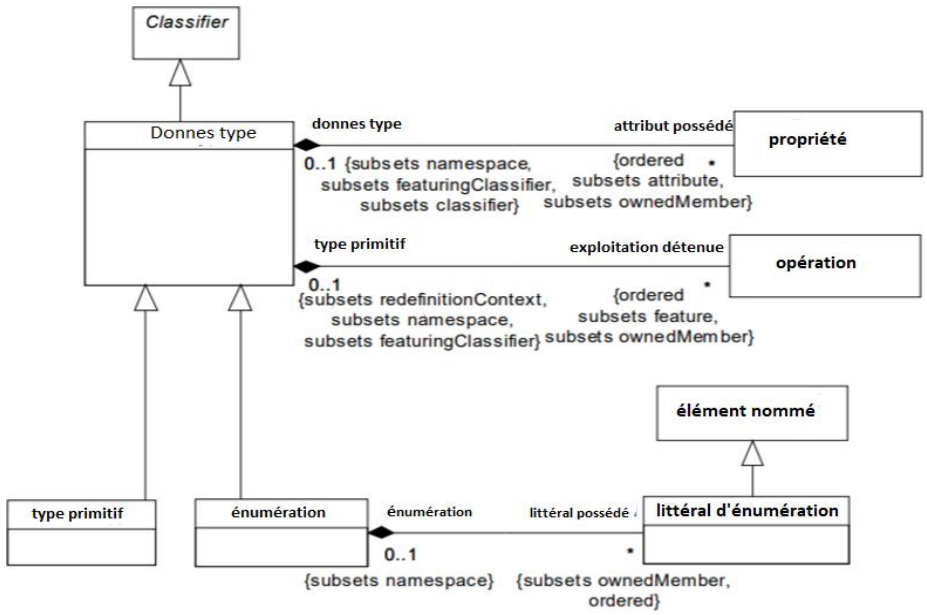

Figure 2.2 – Le sous-ensemble de diagramme de classes du méta-modèle UML [8]

2.5.2 Cohérence statique vs dynamique

La plupart des langages font la distinction entre leur sémantique statique et dynamique . Dans [10], les auteurs expliquent qu'UML ne fait pas exception à cette règle , La sémantique statique, ou la syntaxe, d'UML est formellement décrite en termes de méta-modèle et de contraintes OCL, en plus de quelques descriptions en langage naturel comme " une contrainte attachée à un stéréotype ne doit pas entrer en conflit avec des contraintes sur un stéréotype hérité, ou associé à la classe de base " [11]. Alors que la syntaxe peut généralement être vérifiée par une inspection statique d'un modèle, la sémantique dynamique ne peut pas être complètement vérifiée avant l'exécution. Par exemple, il peut ne pas être possible de vérifier statiquement qu'une condition préalable à une opération est satisfaite avant que l'opération ne soit appelée dans un diagramme d'interaction. Le problème ici est qu'UML n'est pas un langage exécutable, et donc les contraintes dynamiques doivent être intégrées dans un formalisme exécutable (comme le code) qui est traduisible à partir d'UML.

2.5.3 Cohérence intra-modèle et inter-modèle

Une classification récurrente de la cohérence UML dans la littérature [12, 14, 17] distingue entre cohérence intra-modèle et inter-modèle ou entre horizontal et vertical cohérence. La cohérence intra-modèle ou horizontale est une propriété d'un modèle. Cela indique que tous les éléments d'un modèle sont syntaxiquement et sémantiquement corrects .comme mentionné précédemment, UML offre plusieurs points de vue pour modéliser le même système. Ils vont de structurel (ex : diagrammes de classes et d'instances) à comportemental (ex : diagrammes d'interaction et de machine à états). Ces points de vue ou perspectives sont généralement inter dépendants . Par exemple, un message synchrone dans un diagramme de séquence doit correspondre à une opération dans un diagramme de classes. Ce croisement de points de vue conduit à une classe d'incohérences très intéressante qui n'est pas formalisée dans le cadre du cahier des charges . De nombreuses tentatives de recherche [13, 15, 16, 18, 19] ont tenté de définir formellement des contraintes pour vérifier ces incohérences. Si certaines de ces contraintes sont évidentes, d'autres sont principalement heuristiques dans la nature.

D'autre part, la cohérence inter-modèle est une relation entre modèles [22]. Ces modèles sont généralement liés les uns aux autres par une sorte de relation de

transformation. Une transformation décrit l'application de certaines procédures à un modèle pour créer un nouveau modèle [24]. Le nouveau modèle peut être une autre représentation des mêmes informations dans l'ancien modèle, ou une version modifiée du modèle d'origine. En fait, la relation entre transformation et cohérence a été approfondie par les chercheurs [24]. Ils ont caractérisé une transformation comme cohérente si un modèle avant la transformation est cohérent avec le nouveau modèle après la transformation.

2.5.4 Multi-niveaux

Une autre classification de la cohérence est donnée par Sourrouille et Caplat dans [16]. Ils suggèrent que la cohérence peut être gérée à cinq niveaux différents. Le premier niveau, appelé niveau paradigmatique, est basé sur la syntaxe et la sémantique du langage UML lui-même, Le deuxième niveau provient des extensions du méta-modèle UML à travers des profils accompagnés de leurs stéréotypes et contraintes OCL, Par exemple, un stéréotype <<énumération>> est utilisé pour indiquer qu'une classe donnée représente un type énuméré. Par conséquent, un accompagnant la contrainte OCL pourrait affirmer qu'une classe avec ce stéréotype n'a aucune méthode; ses attributs ont des visibilités « publiques » et aucune valeur initiale. L'extension du méta-modèle par l'utilisation de stéréotypes est une pratique couramment utilisée dans de nombreux domaines, comme dans [20].

Un troisième niveau de cohérence basé sur un processus de modélisation. Les spécifications UML décrivent la syntaxe et la sémantique du langage mais ne prescrivent pas de processus pour construire différents diagrammes UML, Les processus de modélisation comblent cette lacune en limitant la portée des expressions autorisées dans UML et fournissant des guides de style pour aider à choisir les représentations appropriées. Des contraintes pourraient être définies pour chaque processus de modélisation afin de maintenir un modèle cohérent avec ce processus. Un processus de modélisation peut être générique comme RUP [21], ou spécifique à un certain groupe, projet, entreprise ou communauté. Par exemple, dans un projet, il pourrait y avoir une contrainte empêchant l'utilisation de classes imbriquées. D'autres contraintes de même niveau proviennent des meilleures pratiques ou des normes de l'industrie. Par exemple, il pourrait y avoir une contrainte vérifiant que toutes les classes déclarent leurs constructeurs privés et fournissent des opérations de création statiques (modèle d'usine) [26].

Un quatrième niveau de cohérence est lié à la plate-forme ou au langage de mise en œuvre spécifique ciblé. Toutes les expressions en UML n'ont pas d'équivalents dans la plate-forme ou le langage de programmation ciblé. Par conséquent, des contraintes sont généralement spécifiées pour limiter l'utilisation d'expressions en UML à celles qui peuvent être traduites dans ces domaines. Ces contraintes sont généralement nécessaires lors du processus de génération de code. Un exemple frappant ici est le cadre de modélisation Eclipse (EMF) [23]. Le Framework permet la génération de code Java de modèles exprimés dans des diagrammes de classes UML. Une caractéristique d'EMF est de permettre la définition de nouveaux types de données dont la sémantique est exprimée en externe par du code java en affectant aux classes le stéréotype prédéfini <<data type>>. cependant, puisqu'il n'y a aucun moyen pour EMF de capturer la structure et le comportement de ces classes, il introduit une contrainte qui garantit que ces classes ne peuvent pas être sous-classées davantage..

Le dernier niveau de cohérence dans la même classification concerne le domaine cible. Un domaine cible est le domaine du monde réel qui est modélisé. Comme l'indiquent les auteurs, les contraintes à ce niveau sont généralement de nature dynamique et exprimées en termes de domaine modélisé spécifique plutôt que de domaine de modélisation UML générique.

2.5.5 Nature de l'erreur de cohérence

Une autre façon de classer les incohérences en UML est en fonction de la nature de l'erreur. Un type d'incohérence est une contradiction, où deux ou plusieurs expressions de modélisation se contredisent. Par exemple, si deux instances de classificateurs différents sont communiquer, alors que les deux classificateurs ne sont pas liés l'un à l'autre, alors c'est une contradiction. Ce type particulier d'incohérence est courant en raison de la redondance inhérente à UML [25]. L'utilisation de plusieurs vues pour définir le même système toujours comporte le risque de contradiction entre ces points de vue.

Un autre type d'incohérence est l'incomplétude, qui survient lorsque certaines informations sont manquantes dans un modèle [8, 16]. Un modèle est complet lorsque tous les diagrammes qui se chevauchent ont des éléments correspondants. Cependant, la spécification de ces éléments correspondants est heuristique et dépendante de la méthodologie de modélisation suivie. Par exemple, un classificateur, défini dans un

diagramme de classes, qui n'a pas d'instance spécifiée dans un diagramme de séquence, pourrait être considéré comme un cas d'incomplétude. Aussi, une classe suffisamment complexe, basée par exemple sur la fréquence de participation de ses instances à différentes interactions, ne peut être complète sans avoir une machine à états correspondante. L'incomplétude pourrait également être inter-modèle.

2.6 Paramètres d'analyse

Cette section fournit un ensemble de paramètres d'analyse utilisés pour analyser les techniques de vérification de cohérence.

2.6.1 Nature : Il identifie les propriétés ciblées d'une application OO telles que statique ou dynamique. Les valeurs possibles sont structurelles, comportementales et toutes deux basées sur les diagrammes UML utilisés.

2.6.2 Basé sur MDA : Il identifie que l'approche se situe dans le domaine MDA ou non. Les valeurs possibles sont oui et non.

2.6.3 Version UML : Il identifie la version d'UML. Les valeurs possibles sont les différentes versions disponibles d'UML telles que 1.0, 1.4, 1.5,1.1.1, 2.0, 2.1.1,2.2,2.5 etc.

2.6.4 Diagrammes UML : Il identifie le ou les diagrammes UML utilisés dans l'approche présentée. Les valeurs possibles sont un ou plusieurs UML diagrammes.

2.6.5 Type de cohérence : Il identifie le type de cohérence pertinent abordé dans l'approche présentée. Les valeurs possibles sont issues de la liste des types de cohérence discutés dans la section 2.3.

2.6.6 Représentation intermédiaire : elle identifie toute représentation temporaire utilisée pour valider la cohérence entre les diagrammes UML. Valeurs possibles toute représentation intermédiaire qui varie d'une technique à l'autre.

2.6.7 Stratégie de cohérence : Il identifie la stratégie utilisée pour valider la cohérence entre les diagrammes UML. Il existe trois types de stratégies disponibles dans la littérature : cohérence par analyse (basée sur un algorithme), cohérence par surveillance (basée sur des règles) et cohérence par construction (génère un artefact à partir d'un autre) [14]. Les valeurs possibles sont les stratégies définies ci-dessus.

2.6.8 Règles fournies : Il identifie le niveau de règles de vérification de cohérence présentées dans un document. Les valeurs possibles sont haute, moyenne et bas qui sont déterminés sur la base de la façon dont les règles sont présentées et discutées dans le document avec leur exactitude logique.

2.6.9 Étude de cas : Il identifie l'utilisation d'exemples réels pour vérifier la validité de l'approche présentée. Les valeurs possibles sont oui et non.

2.6.10 Support d'outil : Il identifie l'outil développé pour automatiser le travail proposé. Le support d'outils permet une validation rapide du travail proposé. Les valeurs possibles sont oui et non.

2.7 Techniques de contrôle de cohérence

Cette section fournit une discussion sur les techniques de vérification de cohérence existantes entre les modèles UML. Cette section analyse également les techniques existantes à l'aide d'une table d'analyse construite à partir des paramètres d'analyse. Les techniques de contrôle de cohérence sont divisées par représentation intermédiaire en trois catégories.

2.7.1 Techniques formellement représentées

Cette sous-section discute et analyse les techniques existantes dans lesquelles la représentation intermédiaire est définie dans n'importe quel langage formel ou dans une notation formelle

2.7.1.1 G. Engels

Engels et al. [40] utilisent des diagrammes de séquence, de collaboration et de diagramme d'états transitions UML 1.3 pour valider la cohérence entre eux. Les auteurs se concentrent sur la représentation formelle des modèles comportementaux UML car il est facile et précis de déterminer les incohérences entre les modèles formels. Les auteurs définissent une approche par étapes pour extraire les problèmes entre les modèles UML. L'approche est proposée pour les applications UML temps réel (UMLRT). La stratégie de surveillance est utilisée à travers laquelle les règles de cohérence sont présentées en termes de définitions et de conditions sur le modèle formellement représenté. Une étude de cas de feux de circulation est également démontrée .Un outil nommé FDR est développé pour mettre en œuvre les travaux proposés.

2.7.1.2 H. Rasch

Rasch et Wehrheim [41] se concentrent sur les diagrammes de classe et de machine à états UML 1.5 pour la validation de cohérence. Les classes et les machines à états sont traduites en CSP-OZ qui est sémantiquement puissant. CSP-OZ est utilisé car il aide à la définition complète de la classe ainsi qu'à l'exécution des méthodes de commande. La traduction de la classe et de la machine à états en Object-Z est également discutée. Des

règles sont présentées pour valider la traduction avec précision. Certaines propositions sont également fournies pour assurer la transformation cohérente de la classe et de la machine à états en Object-Z.

2.7.1.3 S-K. Kim

Kim et Carrington [42] fournissent des règles au niveau du méta-modèle contre la validation de cohérence dans le diagramme d'états transitions. State chart est représenté dans Object-Z pour appliquer des règles de cohérence. L'objectif principal de leur approche est de définir des contraintes de cohérence d'intégrité entre différents modèles UML. Les prédicats sont utilisés pour définir les invariants et les contraintes d'intégrité. Les auteurs fournissent une traduction des constructions de méta-niveau de diagramme d'états transitions vers Object-Z.

2.7.1.4 . Y. Shinkawa

Shinkawa [43] utilise des diagrammes de cas d'utilisation, de classe, de séquence, d'activité et de diagramme d'états transitions UML 2.0 pour vérifier la cohérence. L'article présente la classification des modèles UML et l'auteur inclut un diagramme de chaque classement. L'auteur fournit un mappage pour la conversion des diagrammes UML en réseaux de Petri colorés (CPN). La cohérence est validée par la traduction des diagrammes UML en CPN. Différents exemples sont utilisés pour illustrer la traduction. Les réseaux de Petri colorés (CPN) sont utilisés comme représentation intermédiaire. La cohérence inter-modèle est validée entre les modèles UML. Des règles sont présentées pour traduire chaque diagramme UML en CPN.

2.7.1.5 Z.Liu

Liu et al. [47] basent les diagrammes de classe, de séquence et de diagramme d'états transitions UML 2.0 pour proposer une approche de vérification de cohérence. Le langage de spécification orienté objet (OOL) est utilisé pour présenter la conception de logiciels officiellement. Des preuves théoriques sont fournies pour montrer la puissance d'OOL pour présenter des diagrammes UML. Certaines contraintes sont définies pour assurer la cohérence entre les représentations formelles OOL. Les diagrammes UML sont traduit en OOL par le biais d'une étude de cas de point de vente. Aucun support d'outil n'est fourni. La stratégie d'analyse est mise en œuvre par l'approche.

2.7.2 Techniques de représentation UML étendues

Cette sous-section discute et analyse les techniques présentées dans lesquelles la représentation intermédiaire est définie comme une extension dans le(s) diagramme(s) UML.

2.7.2.1 G. Engles

Engles et al. [35] utilisent des diagrammes de séquence, de collaboration et de diagramme d'états transitions UML 1.4 pour la validation de la cohérence. Des règles de méta-modélisation dynamique (DMM) sont utilisées à cette fin. Les règles sont fournies sous la forme de deux contraintes {new} et {destroyed}. La contrainte {new} lance un appel de création à la classe attachée. La contrainte {destroyer} supprime l'instance attachée de l'espace d'exécution. L'environnement du testeur est également fourni pour valider la cohérence.

2.7.2.2 T. Mens

Mens et al. [44] étendent les méta-modèles de diagrammes de classes, de séquences et de diagrammes d'états UML 2.0 pour inclure la prise en charge de la gestion des versions pour la validation de la cohérence. La logique de description (DL) est utilisée pour détecter et résoudre formellement les incohérences. Cinq stéréotypes tels que <<versioned>><<horizontal>>, <<évolution>>, <<refine>> et <<trace>> sont également inclus dans l'extension UML. OCL est utilisé pour la définition des stéréotypes.

2.7.3 Aucune technique de représentation intermédiaire

Cette sous-section discute et analyse les techniques présentées dans lesquelles aucune représentation intermédiaire.

2.7.3.1 M. Grischick

Grischick [36] utilise le diagramme de classes UML 1.5 pour détecter les incohérences entre deux versions. Un algorithme est proposé à cet effet. Différents codes de couleur sont utilisés pour distinguer les propriétés de la classe diagramme. L'approche indique non seulement la différence entre deux versions, mais fournit également des informations sur la façon dont la différence est produite. Une structure de données est proposée pour la mise en œuvre de l'approche. Évaluation La fonction est utilisée pour comparer deux éléments du diagramme de classes. L'analyse critique est présentée par rapport à l'algorithme proposé

2.7.3.2 B. Graaf

Graaf et Deursen [37] se concentrent sur le scénario UML 2.2(séquence, collaboration) et les diagrammes d'états transitions pour la vérification de cohérence. L'approche génère des diagrammes d'états transitions à partir de scénarios. Transformation des règles sont fournies dans ATL pour générer des diagrammes d'états transitions à partir de scénarios. Les scénarios sont créés à partir de cas d'utilisation. La transformation se fait en quatre étapes. La première étape applique les connaissances du domaine. La deuxième étape génère des diagrammes d'états aplatis. La troisième étape fusionne les diagrammes d'états

transitions aplatis contre leur classe respective. La dernière étape introduit les informations de hiérarchie dans les diagrammes d'états transitions fusionnés.

2.7.3.3 L.C. Briand

Briand et al. [51, 52] utilisent des diagrammes de classe, de séquence et de diagramme d'états transitions UML 1.4 pour évaluer la cohérence. L'approche se concentre sur la gestion du changement dans les modèles UML. L'effet du changement est également l'estimation par l'approche avant la mise en œuvre effective du changement. Une stratégie de surveillance est appliquée à travers laquelle des règles de cohérence sont mises en œuvre dans l'outil. Les expressions OCL sont utilisées pour exprimer les règles. L'ATM est utilisé comme étude de cas pour valider la faisabilité de l'approche. L'analyse expérimentale et les résultats sont également fournis.

2.7.3.4 T-H. Feng

Feng et Vangheluwe [48] utilisent des diagrammes de séquence de classes et de diagrammes d'états transitions UML 1.5 pour la validation de la cohérence. L'approche couvre les applications client-serveur. Les problèmes de cohérence sont résolus entre Composants. Des règles sont présentées pour valider la cohérence. Des traces de sortie sont utilisées à cette fin. La règle se compose de quatre parties en tant que propriété de pré-condition , post-condition, garde (facultatif) et contre-règle(optionnel).

2.7.3.5 A. Egyed

Egyed [50] utilise la classe UML 2.2, les diagrammes de séquence et les diagrammes d'états transitions pour valider la cohérence entre eux. L'approche est appliquée sur les instances d'exécution pour la validation de la cohérence. les problèmes dans les instances d'exécution sont résolus grâce à la portée d'une règle de cohérence et d'un enregistreur. Les valeurs de vérité sont utilisées pour déterminer la portée minimale de la règle de cohérence. Les règles sont fournies dans OCL via une stratégie de surveillance. Une validation est apportée par des études expérimentales. L'outil UML/ANALYZER [31] est développé pour automatiser l'approche. Il contient trois composants : un vérificateur de cohérence, un profileur d'évaluation et un détecteur de règles. L'outil est intégré à IBM Rational Rose pour une utilisation ouverte. Les résultats expérimentaux sont présentés à l'aide de l'outil développé

2.8 conclusion

Dans ce chapitre nous avons présenté brièvement la cohérence uml. Pour cela, nous avons commencé par la définition de la cohérence , par la suite nous avons vu les types de la cohérence et leur définition et exemple . ensuite nous avons présenté la Classification de cohérence UML et les paramètres d'analyse

Enfin nous avons cité quelques Techniques de contrôle de cohérence entre les diagrammes uml , nous présenterons des règles dans le chapitre suivant.

CHAPITRE

3

Règles de Cohérence

Sommaire

3.1 introduction

La cohérence est la situation où deux ou plusieurs éléments se chevauchant de différents diagrammes qui décrivent le comportement du système sont conjointement satisfiables. C'est l'un des attributs pour mesurer la qualité du modèle UML. Même si la recherche sur la cohérence entre les diagrammes est rapidement augmentée, il y a toujours un manque de recherche de cohérence conduite par un diagramme spécifique . Par conséquent, Ce chapitre définira des éléments de diagrammes uml, ainsi que la cohérence entre eux en utilisant une approche logique.

3.2 les règles de cohérence existantes dans UML[87]

nous présenterons un ensemble de règles tirées des recherches des chercheurs en règles de cohérence UML ont généralement défini un certain nombre de règles similaires maintes et maintes fois. Plus précisément, a laide de ces recherches nous a collecté une liste des règles. Après avoir supprimé les règles inexactes, redondantes et déjà présentées dans les standards UML, nous avons obtenu un ensemble final de 116 règles qui sont présentées dans le section 3.5.

3.3 Règles de cohérence UML enter les diagrammes

Le tableau montre que les paires de diagrammes les plus impliquées dans les règles sont CD-SD, CD-COD et SD-DE

CD : diagramme de class SD : diagramme de séquence

COD : diagramme de collaboration DE : diagramme d'état

Cohérence entre 2 diagrammes	règles
Diagramme de classes et diagramme de séquence	26
Diagramme de classes et diagramme d'état	25
Diagramme de classes et diagramme de collaboration	11
Diagramme d'état et diagramme de séquence	9
Diagramme de séquence et diagramme d'activités	9
Diagramme de séquence et diagramme de cas d'utilisation	5
Diagramme d'activité et diagramme de cas d'utilisation	5
Diagramme de classes et diagramme de cas d'utilisation	5

Tableau 3 – Cohérence entre deux Diagrammes

3.4 Règles de cohérence UML d'un seul diagramme

Cohérence dans un seul diagramme	règles
Diagramme de classe	56
Diagramme d'état	52
Diagramme de séquence	10
Diagramme de collaboration	3
Diagramme d'activités	6
Diagramme de cas d'utilisation	8

Tableau 4 – Cohérence de un seul diagramme

3.5 Règles de cohérence UML

Étant donné que les différents diagrammes UML qui sont construits au cours d'un développement logiciel spécifique spécifient différents aspects d'un et d'un seul système, ces diagrammes doivent être cohérents les uns avec les autres. Ceci est vrai que les schémas soient générés à la main (à l'aide d'un outil CASE par exemple) ou synthétisés à partir d'autres schémas. Les techniques de synthèse de diagramme UML dont nous disposons imposent une sorte de cohérence entre les diagrammes d'entrée de la synthèse et les diagrammes en cours de synthèse.

Diagramme de la machine à états	
1	Considérons deux machines à états U' et U d'une classe O' et de sa superclass O, où U' étend la machine à états de U en ajoutant des états et des transitions. Les états initiaux des diagrammes de machine à états U' et U doivent être identiques.[69]

2	Considérons deux machines à états U' et U d'une classe O' et de sa superclasse O, où U' étend la machine à états de U en ajoutant des états et des transitions. Chaque transition de U' qui est déjà dans U a au moins les mêmes états source et puits que dans U.
3	Considérons deux machines à états U' et U d'une classe O' et de sa superclasse O, où U' étend la machine à états de U en ajoutant des états et des transitions. Pour chaque transition t dans U' déjà présente dans U, la condition de garde g'(t) dans U' doit être au moins aussi forte que la condition de garde g(t) pour t dans U : g'(t) → g(t).[79]
4	Considérons deux machines à états U' et U d'une classe O' et de sa superclasse O, où U' étend la machine à états de U en ajoutant des états et des transitions. Une transition de U dans U' ne peut donc recevoir un état source ou un état de puits supplémentaire qui est déjà présent en U. I
5	Considérons deux State Machines U' et U d'une classe O' et sa superclasse O, où U' étend la machine d'état de U en ajoutant des états et des transitions. Une transition ajoutée à U' ne reçoit pas un état source ou un état de puits qui était déjà présent en U.
6	Considérons deux State Machines U' et U d'une classe O' et sa superclasse O, où U' étend la machine d'état de U en ajoutant des états et des transitions. L'ensemble des transitions de U' est un superset de l'ensemble des transitions de U.[70]
7	Considérons deux State Machines U' et U d'une classe O' et sa superclasse O, où U' étend la machine d'état de U en ajoutant des états et des transitions. Une transition en U' qui est déjà présente en U a dans U' tout au plus la source indique que la transition a en U.
8	Considérons deux State Machines U' et U d'une classe O' et sa superclasse O, où U' étend la machine d'état de U en ajoutant des états et des transitions. Pour chaque transition t en U qui est déjà présente en U, U : g(t) g'(t).
9	Considérez deux State Machine U' et U d'une classe O' et sa superclasse O, où U' affine le diagramme d'état de U au moyen d'une fonction de raffinement h qui mappe les transitions sur les transitions et les états de U' et qui mappe les états simples de U' sur les états simples de U. Intuitivement, le concept de la fonction de raffinement signifie que si un état simple s de U a été affiné à un état composite en U", alors h mappe les sous-états de s en U' et les transitions entre ces états à s, h met en correspondance les transitions en U qui sont des incidents aux sous-états de s et en transitions de U qui sont des incidents à s. s est un état en U, donc s Є U (de la règle : "un état simple s de U","s de h(t')") s' est un état raffiné en U', donc s' Є U' (de la règle : "un état s' en U' ","sink state s' de t' en U' ") Source+(t) : t ensemble d'états. Source+(t) = { source de t} Sous-états U*{source de t} s' Є Source+(t) Pour chaque transition t' en S' : pour chaque état source s de h(t'), il existe un état s' en S'

	tel que h(s') = s, et pour chaque état de puits s de h(t') il existe un état de puits s' de t' en S' tel que h(s') = s. Définition informelle de la règle : Une transition t' (de U') qui est raffinée à partir de t (de U) doit avoir s' (expéditeur d'État et destinataire de U'') qui est raffinée à partir de s (de U) Є Source+(t).[72]
10	Considérez deux State Machine U' et U d'une classe O' et sa superclasse O, où U' affine le diagramme d'état de U au moyen de la fonction de raffinement h qui mappe les transitions sur les transitions et les états de U' et qui mappe les états simples de U' sur les états simples de U. Intuitivement, le concept de la fonction de raffinement signifie que si un état simple s de U a été affiné à un état composite en U", alors h mappe les sous-états de s en U' et les transitions entre ces états à s, h met en correspondance les transitions en U qui sont des incidents aux sous-états de s et en transitions de U qui sont des incidents à s. s est un état en U, donc s Є U (de la règle : "un état simple s de U","s de h(t')") s' est un état raffiné en U', donc s' Є U' (de la règle : "un état s' en U' "," sink state s' de t' en U' ") Source+(t) : t ensemble d'états. Source+(t) = { source de t} Sous-états U*{source de t} s' Є Source+(t) Pour chaque état source s' d'une transition t' en S'', où s' et t'' n'appartiennent pas au même état raffiné (c.-à-d. h(s') /= h(t'')), h(s') est donc un état source de h(t'), et pour chaque état de puits s''' d'une transition t' en S'', où s' et t' n'appartiennent pas au même état raffiné, h(s'') est donc un état d'évier de h(t''). Définition informelle des règles : Cette règle concerne les transitions entre les États qui ne sont pas des sous-états d'un État complexe qui est un raffinement d'un État simple. La règle dit que ces transitions ont leurs sources et leurs puits en U' comme raffinements de ceux en U.
11	Utiliser un signal/message sur une transition dans un diagramme d'état qu'aucun objet n'envoie, crée des incohérences structurelles et syntaxiques. L'information sur l'objet d'envoi peut être trouvée dans une autre machine d'état, dans une autre partition de la même machine d'état, dans un diagramme de séquence, ou tout autre diagramme où l'on peut spécifier un objet peut envoyer un signal / message.
12	Une machine d'état devrait être sans impasse
13	A state machine must be deterministic, that is, in every state, only one transition (accounting for the different levels of nested states) should fire on a reception of an event

14	Une opération abstraite ne peut être invoquée dans une machine étatique
Diagramme de classes, diagramme de machine d'état et diagramme d'activité	
15	Il y a incohérence si une condition préalable à une opération est en contradiction avec une machine d'État ou un diagramme d'activité comprenant un appel de cette opération.
Diagramme de séquence et diagramme de cas d'utilisation	
16	Si un cas d'utilisation est précisé par un ou plusieurs diagrammes de séquence, alors chaque scénario décrit dans l'un de ces diagrammes de séquence doit correspondre à une séquence d'étapes dans la description du cas d'utilisation de ce cas.
17	Si un diagramme de séquence décrit tous les comportements requis pour réussir un cas d'utilisation, il s'ensuit que chaque post-condition spécifiée dans la description du cas d'utilisation doit être réalisée par un message (ou un ensemble de messages) dans le diagramme de séquence (y compris les diagrammes référencés) pour ce cas d'utilisation.
18	Si un diagramme de séquence décrit tous les comportements requis pour réussir un cas d'utilisation, il s'ensuit que les résultats obtenus par d'autres flux de messages dans le diagramme de séquence (y compris les diagrammes mentionnés) doivent correspondre à post-conditions spécifiées dans la description du cas d'utilisation.
19	Chaque action spécifiée ou implicite dans une description de cas d'utilisation doit être détaillée dans un message correspondant ou un ensemble de messages dans le diagramme de séquence correspondant à ce cas d'utilisation. Selon la clarté et l'exhaustivité du texte de la description du cas d'utilisation, l'auteur du diagramme de séquence peut devoir déduire certaines des opérations.
20	Un cas d'utilisation est complété par un ensemble de diagrammes de séquence, chaque diagramme de séquence représentant un scénario alternatif du cas d'utilisation
Diagramme d'activité (pas de règle)	
Diagramme de séquence	
21	Les arguments des messages doivent représenter des informations connues de l'expéditeur. Cela inclut les valeurs d'attribut de l'expéditeur, les expressions de navigation commençant par l'expéditeur, les constantes ; Cela devrait tenir compte de l'héritage.

22	Les variables utilisées dans la garde d'un message doivent représenter des informations connues de l'expéditeur. Cela inclut les valeurs d'attribut de l'expéditeur, les expressions de navigation commençant par l'expéditeur, les constantes ; Cela devrait tenir compte de l'héritage.
23	Si SD2 est un diagramme de séquence référencé par une utilisation d'interaction (un diagramme de séquence imbriqué) incorporé dans le diagramme de séquence SD1, alors pour chaque paire de messages appariés vers et depuis SD2 dans SD1, il y a une paire correspondante de messages sans source et de messages sans cible dans SD2.
24	Dans un diagramme séquentiel, si un attribut est affecté à la valeur de retour d'un message, alors les types doivent être compatibles.
25	Les messages de retour des instances Exécution Spécification doivent toujours être affichés.
26	Les arguments des messages doivent toujours être affichés.
Diagramme d'objets et diagramme de collaboration (pas de règle)	
Diagramme d'objets et diagramme de classes	
27	Le nombre d'occurrences d'un lien dans un diagramme objet, instance d'une association dans un diagramme de classe, doit satisfaire aux contraintes de multiplicité spécifiées pour l'association.
Diagramme d'activités et diagramme de classes	
28	Un nom de classe qui apparaît dans un diagramme d'activité apparaît également dans le diagramme de classe
29	Une action qui apparaît dans un diagramme d'activité doit également apparaître dans le diagramme de classe comme une opération d'une classe.
30	Quand un diagramme d'activité spécifie un changement précis d'information, par le contrôle ou le flux de données, entre deux instances différentes, le diagramme de classe devrait spécifier un mécanisme (par exemple, l'association directe) de
Diagramme de classes et diagramme de machine à états	
32	Nageurs (modèle d'activité dans UML 2.0) dans un diagramme d'activité (représenté comme suit : class Name in activity state) doit être présent en tant que classe unique dans un diagramme de classe.
	Lorsqu'une machine à états spécifie le comportement d'une classe, les actions et activités dans la machine à états doivent être des opérations de la classe (dans le diagramme de classes) dont le comportement est spécifié par la machine à états. Une action ou une

32	activité peut faire partie d'une expression de navigation, auquel cas l'expression de navigation doit être légale selon le diagramme de classes et le contexte (classe) du comportement spécifié dans la machine d'état
33	Lorsqu'une machine d'état spécifie le comportement d'une classe, toute propriété (c'est-à-dire un attribut, une navigation) dans la machine d'état doit appartenir à la classe (dans le diagramme de classes) dont le comportement est spécifié par la machine d'état. Une propriété peut faire partie d'une expression de navigation, auquel cas l'expression de navigation doit être légale selon le diagramme de classes et le contexte (classe) du comportement spécifié dans la machine d'état
34	Lorsque le diagramme de machine à états spécifie le comportement d'une classe dans le diagramme de classes, la classe est une classe active avec un classifier comportement , alors les déclencheurs des transitions dans le diagramme de machine à états sont des opérations de la classe active.
35	Aucune opération ne peut être appelée sur une machine à états ou depuis une machine à états si cela enfreint les règles de visibilité du diagramme de classes (public, protected, private).
36	Lorsque le comportement est déclenché à partir d'un diagramme de machine d'état (par exemple, l'appel d'une opération, l'envoi d'un signal) qui décrit le comportement d'une classe et que le comportement déclenché appartient à une autre classe, alors le premier doit avoir un hand le sur le dernier comme spécifié dans le diagramme de classe. Une autre façon de dire cela est que le premier doit avoir de la visibilité pour le second. Un cas particulier de cette situation est lorsque la première classe a une association (éventuellement héritée) avec la dernière classe.
37	Pour une action d'envoi, il doit y avoir une réception dans le classificateur de l'instance de récepteur qui correspond au signal de l'action d'envoi décrivant la réponse comportementale attendue au signal.[66]
38	Pour tous les événements d'appel ou d'envoi spécifiés dans le diagramme de classes pour un classificateur (contexte) dont le comportement est spécifié avec une machine d'état, il devrait y avoir des transitions dans cette machine d'état décrivant le comportement détaillé des événements
Diagramme de séquence et diagramme d'activité	
39	Si un diagramme d'activités présente des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, les différents diagrammes doivent spécifier les mêmes scénarios : par exemple, même séquence de messages/opérations/actions, mêmes conditions de branchement ou de répétition

40	Si un diagramme d'activités montre des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, un flux d'interaction entre les objets d'un diagramme d'activités doit être un flux d'interactions entre les mêmes objets dans un diagramme de séquence
41	Si un diagramme d'activité montre des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, une séquence de messages dans le diagramme de séquence, ininterrompue par des structures de flux de contrôle, doit correspondre à un nœud d'activité dans le diagramme d'activité quel nom est la série de noms de message du diagramme de séquence.
42	Si un diagramme d'activités montre des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, un message synchrone entre les objets exécutés dans différents threads de contrôle dans le diagramme de séquence doit correspondre à un nœud de jointure pour le côté récepteur (thread) dans le diagramme d'activités correspondant , et un nœud de fourche pour la réponse asynchrone.[83]
43	Si un diagramme d'activité présente des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, une création asynchrone d'un objet actif dans le diagramme de séquence doit correspondre à un nœud de fourche dans le diagramme d'activité correspondant : le nœud d'action d'entrée correspond au message de thread appelant ; deux bords sortants doivent sortir du nœud de fourche, un pour le nœud correspondant à la poursuite de l'exécution dans le thread appelant et un pour le nœud correspondant à l'objet actif nouvellement créé.
44	Si un diagramme d'activités montre des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, un message asynchrone envoyé entre deux objets actifs dans le diagramme de séquence doit correspondre, dans le diagramme d'activités correspondant, à un nœud de jointure sur côté récepteur et un nœud de dérivation côté émetteur.[74]
45	Si un diagramme d'activité montre des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, une Interaction Use dans le diagramme de séquence doit correspondre à un nœud d'activité dans le diagramme d'activité qui fait référence au diagramme d'activité correspondant au diagramme de séquence mentionné dans l'Interaction Use
Diagramme de cas d'utilisation et diagramme d'objets (pas de règle)	
Diagramme de la machine à états et diagramme de communication	
	Lorsqu'on spécifie une classe active, c'est-à-dire qui a un comportement basé sur l'état décrit dans un diagramme de machine à états, et qu'une instance de cette classe active est

46	utilisée dans un diagramme de communication, les messages envoyés à cet objet et émis par cet objet comme spécifié dans le schéma de communication doit être conforme au protocole spécifié dans le schéma de la machine d'état [85]
Diagramme de communication et diagramme de classes	
47	Les méthodes utilisées dans un diagramme de communication doivent être déclarées dans le diagramme de classes et leur déclaration, en ce qui concerne les paramètres et les types, doit être correcte
Diagramme de la machine à états et diagramme de séquence	
48	Lorsqu'on spécifie une classe active, c'est-à-dire qui a un comportement basé sur un état décrit dans un diagramme de machine à états, et qu'une instance de cette classe active est utilisée dans un diagramme de séquence, les messages envoyés à cet objet et émis par cet objet comme spécifié dans le diagramme de séquence doit être conforme (par exemple, séquence et types de signaux, récepteurs et émetteurs de signaux) au protocole spécifié dans le diagramme de la machine d'état.[75]
Diagramme de la machine d'état du protocole (pas de règle)	
Diagramme de communication	
49	Si le schéma de communication A est une spécialisation du schéma de communication B, tous les messages présents dans B doivent être inclus dans A
Diagramme de cas d'utilisation et diagramme de classes	
50	Le nom du cas d'utilisation doit être égal au nom d'une classe dans le diagramme de classes.
51	Le verbe du nom du cas d'utilisation doit être égal au nom d'une opération d'une classe dans le diagramme de classes.
52	Les classes d'entités correspondent aux données manipulées par le système comme décrit dans une description de cas d'utilisation
Diagramme d'objets et diagramme d'activités (pas de règle)	
Diagramme de communication et diagramme d'activité	

53	Si un diagramme d'activités spécifie un flux d'interaction entre des objets et que ces objets (ou un sous-ensemble de ces objets) apparaissent dans un diagramme de communication, alors le diagramme de communication doit spécifier le même flux d'interaction.
Diagramme de communication et diagramme de classes	
54	Les objets impliqués dans un diagramme de communication doivent être des instances de classes du diagramme de classes.
55	Pour que les objets échangent des messages dans un diagramme de communication, l'objet émetteur doit avoir un handle vers l'objet récepteur comme spécifié dans le diagramme de classes. Une autre façon de dire cela est que l'expéditeur doit avoir une visibilité sur le destinataire. Un cas particulier de cette situation est lorsque la classe de l'objet émetteur a une association (éventuellement héritée) avec la classe de l'objet récepteur.[67]
56	Chaque classe du diagramme de classes apparaît avec au moins une instance dans au moins un diagramme de communication.
Diagramme de communication et diagramme de cas d'utilisation (pas de règle)	
Diagramme de cas d'utilisation et diagramme d'activité	
57	fa cas d'utilisation U (le cas d'utilisation inclus) inclut le cas d'utilisation V (le cas d'utilisation inclus) dans le diagramme de cas d'utilisation, et les flux du cas d'utilisation U et les flux du cas d'utilisation V sont spécifiés en tant que diagrammes d'activité, puis le diagramme d'activité spécifiant le cas d'utilisation U doit contenir un nœud d'action (probablement un nœud Call Behavior Action) qui fait référence au diagramme d'activité spécifiant le cas d'utilisation V.
58	Chaque cas d'utilisation est décrit par au moins un diagramme d'activité.
59	Chaque événement (flux d'étapes) spécifié ou implicite dans la description du cas d'utilisation doit être détaillé dans un événement correspondant du diagramme d'activité. Cette règle n'est valide que si le diagramme d'activité précise davantage le cas d'utilisation.
60	Un acteur associé à un cas d'utilisation sera une partition d'activité dans le diagramme d'activité décrivant ce cas d'utilisation.[77]
Use Case Diagram	
61	La description de cas d'utilisation d'un cas d'utilisation dans le diagramme de cas d'utilisation doit contenir au moins un événement (flux d'étapes). Même si une description de cas d'utilisation peut être affichée/représentée dans un vue séparée respecter le diagramme de cas d'utilisation, nous considérons qu'un cas d'utilisation est accompagné d'une description.[78]

62	Un événement (flux d'étapes) dans une description de cas d'utilisation d'un cas d'utilisation du diagramme de cas d'utilisation doit être de type basique ou alternatif. Même si une description de cas d'utilisation peut être affichée/représentée dans une vue séparée par rapport au diagramme de cas d'utilisation, nous considérons qu'un cas d'utilisation est livré avec une description
63	Chaque cas d'utilisation dans le diagramme de cas d'utilisation doit avoir une description de cas d'utilisation spécifiant les flux d'événements. Même si une description de cas d'utilisation peut être affichée/représentée dans une vue séparée par rapport au diagramme de cas d'utilisation, nous considérons qu'un cas d'utilisation est accompagné d'une description.
64	Le nom d'un cas d'utilisation doit inclure un verbe et un nom (par exemple « Valider l'utilisateur »). [79]
Diagramme de classes	
65	Un invariant de classe doit être satisfait par toute instanciation non-triviale du diagramme de classes (c'est-à-dire une instanciation qui ne se réduit pas à n'avoir aucune instance d'aucune classe).
66	Le type d'une relation entre deux classes à un (haut) niveau d'abstraction (par exemple, association simple, agrégation, composition, généralisation) doit être le même que le type d'un raffinement de cette relation à un niveau plus concret (bas) de abstraction. Par exemple, une association simple à un faible niveau d'abstraction étant abstraite comme une agrégation (un niveau d'abstraction élevé) dénote une incohérence.
67	Un diagramme de classes est cohérent s'il peut être instancié sans violer aucune des contraintes du diagramme (par exemple, les multiplicités de fin d'association)
68	Deux classes sont équivalentes si elles désignent le même ensemble d'instances chaque fois que les contraintes imposées par le diagramme de classes sont satisfaites. La détermination de l'équivalence de deux classes permet leur fusion, réduisant ainsi la complexité du schéma.
69	Une relation de classe à un (faible) niveau d'abstraction doit avoir une abstraction à un niveau d'abstraction plus élevé, soit une classe soit une relation
70	Si une relation de classe A affine la relation B, A doit avoir les mêmes classes destinations que les classes destinations de l'abstraction de B
71	Si une relation de classe A affine la relation B, A doit avoir le même type que B.
72	Le groupe de relations entre deux classes de haut niveau quelconques doit être identique au groupe de relations entre leurs classes de bas niveau correspondantes : assure les mêmes interactions entre classes de bas niveau et classes de haut niveau.
73	Si une expression de navigation est utilisée dans un contrat d'opération, alors l'expression doit être légale (selon la syntaxe du langage et le diagramme de classes)

74	Ce principe de substitution de Liskov tient
75	Une classe qui réalise une interface doit déclarer toutes les opérations dans l'interface avec les mêmes signatures (y compris la direction des paramètres, les valeurs par défaut, la concurrence, la propriété polymorphe, la caractéristique de la requête)
76	Une opération abstraite ne peut appartenir qu'à une classe abstraite.
77	Si une opération apparaît dans une condition pré ou post, alors sa propriété is Query doit être égale à true.
78	Aucune méthode (publique) d'une classe ne viole, comme indiqué par ses pré et post-conditions, l'invariant de classe de cette classe.
79	Dans une classe, les noms des fins d'association (à l'opposé des associations de cette classe) et les noms des attributs (de la classe) sont différents.
80	Aucune condition préalable ne doit violer l'invariant de classe.
81	Aucune post-condition ne doit violer l'invariant de classe.[80]
82	Une classe qui contient une opération abstraite doit être abstraite.
83	Il ne doit pas y avoir de cycle dans le chemin dirigé des associations d'agrégation : une classe ne peut pas faire partie d'une agrégation dont elle est le tout ; Une classe ne peut pas faire partie d'une agrégation dans laquelle sa superclasse (ou ancêtre) est le tout
84	Une classe ne peut pas faire partie de plus d'une composition - aucune partie composite ne peut être partagée par deux classes composites.
85	Chaque classe concrète, c'est-à-dire qu'elle n'est pas abstraite, doit implémenter toutes les opérations abstraites de sa ou ses super-classes.
86	Si le type d'un attribut est une classe, alors cette classe doit être visible pour la classe contenant l'attribut. Exemple : même package ou il existe un chemin dans le diagramme de classes qui permet à la classe contenant l'attribut d'avoir une emprise sur ce type.[84]
87	Si le type de retour d'une opération est une classe, alors cette classe doit être visible pour la classe contenant l'opération. Exemple : même package ou il existe un chemin dans le diagramme de classes qui permet à la classe contenant l'opération d'avoir une emprise sur ce type
88	Une opération ne peut être remplacée par une classe descendante que si son attribut is Leaf (de la métaclasse RedefinableElement) est défini en conséquence.
89	Une opération statique ne peut pas accéder à un attribut d'instance (comme indiqué par ses conditions pré et post, par exemple).
90	Une opération statique ne peut pas invoquer une opération d'instance (comme indiqué par ses conditions pré et post, par exemple).[82]

91	Si une extrémité d'association a une visibilité privée, alors la classe à cette extrémité n'est accessible via l'association que par la classe à l'autre extrémité d'association (par exemple, comme indiqué par les conditions pré et post d'opérations de la classe à cette autre extrémité de l'association).
92	Si une extrémité d'association a une visibilité protégée, alors la classe à cette extrémité n'est accessible que via l'association, par la classe à l'autre extrémité d'association et ses descendants (par exemple, comme indiqué par les conditions pré et post des opérations de la classe à cette autre extrémité de l'association).
93	La plage de multiplicité d'un attribut doit être respectée par tous les éléments (contrats d'exploitation, conditions de garde) qui y accèdent
94	Pour que les opérations de la classe A utilisent une autre classe B, comme indiqué par les contrats en A, il doit y avoir un moyen (par exemple, sous la forme d'un chemin impliquant des associations, des généralisations et/ou des dépendances) dans le diagramme de classes pour que A prenne la main sur B[83]
95	Une classe à un bas niveau d'abstraction affine au plus une classe à partir d'un niveau d'abstraction supérieur.
96	Une classe à un haut niveau d'abstraction est raffinée par au moins une classe d'un niveau d'abstraction inférieur
97	Il ne devrait pas y avoir de chemins sémantiquement redondants entre deux classes dans le graphe du diagramme de classes, à moins que cela ne soit spécifié avec précision par une contrainte (par exemple, spécifié dans OCL). Par exemple, à partir de la classe A, il peut être possible de naviguer en tant que self . the A . the B, avec self . the C. the B. La question est alors de savoir si les deux collections self. The A .the B et self. The C. the B sont identiques. [85]
Diagramme d'interaction et diagramme de classes (pas de règle)	
Diagramme de cas d'utilisation et diagramme d'interaction	
98	Chaque diagramme d'interaction correspond à un cas d'utilisation dans le diagramme de cas d'utilisation, et chaque cas d'utilisation dans le diagramme de cas d'utilisation est spécifié par un diagramme d'interaction
99	Si un cas d'utilisation est en outre spécifié par un ou plusieurs diagrammes d'interaction, alors chaque scénario décrit dans l'un de ces diagrammes d'interaction doit correspondre à une séquence d'étapes dans la description de cas d'utilisation de ce cas d'utilisation.
Composite Structure Diagram	
100	Un connecteur de délégation connecte un port sur la limite d'un classificateur à un port sur la structure interne (ou des parties) du classificateur. Il délègue essentiellement les signaux ou les appels d'opération arrivant sur le port frontière vers le port interne, ou ceux provenant du port interne vers le port frontière. Par conséquent, ces ports à l'extrémité du connecteur de délégation doivent avoir les mêmes (ou compatibles) interfaces. Si les deux ports nécessitent l'interface, la direction de la délégation va du

	port limite au port interne. Si les deux ports fournissent l'interface, la direction de la délégation va du port interne au port frontière.[86]
101	Un connecteur de délégation connecte un port sur la limite d'un classificateur à un port sur la structure interne (ou des parties) du classificateur. Il délègue essentiellement les signaux ou les appels d'opération arrivant sur le port frontière vers le port interne, ou ceux provenant du port interne vers le port frontière. Par conséquent, ces ports à l'extrémité du connecteur de délégation doivent avoir les mêmes interfaces (ou compatibles). Si les deux ports ont l'interface, la direction de la délégation va du port limité au port interne. Si les deux ports fournissent l'interface, la direction de la délégation va du port interne au port frontière.
102	Si un connecteur est typé avec une association, le sens de l'association doit être conforme au sens du connecteur dérivé du sens des ports à ses extrémités (association navigable de la classe A à la classe B si le connecteur entre A et B indique que A a besoin des services que B fournit). Étant donné que la direction des associations et des connecteurs (cependant, elle est calculée) pourrait être codée en utilisant l'ordre de leur collection memberEnd' et 'end', respectivement, la règle dit essentiellement que cette direction devrait être la même dans les deux cas, si un connecteur est typé par une association.
103	Si un lien sortant d'un port est typé statiquement avec une association, alors l'association doit être navigable vers une interface qui fait partie de l'ensemble des interfaces et le type indiqué par l'association doit appartenir à l'ensemble des interfaces transportées pour ce lien.[29]
104	Si un lien sortant d'un port est typé statiquement avec une association, alors l'association doit être navigable vers une interface qui fait partie de l'ensemble des interfaces et le type indiqué par l'association doit appartenir à l'ensemble des interfaces transportées pour ce lien.
105	L'ensemble des interfaces transportées par un lien ne doit pas être vide.
106	Si plusieurs connecteurs non typés partent d'un même port, alors les ensembles d'interfaces transportés par chacun de ces connecteurs doivent être disjoints par paire.
107	L'union des ensembles d'interfaces transportés par chacun des connecteurs en provenance d'un port P doit être égale à l'ensemble d'interfaces fourni/requis par P[63]
Diagramme de séquence et diagramme de classes	
108	Le type d'une ligne de vie (type de l'élément connectable de la ligne de vie) dans un diagramme de séquence ne doit pas être une interface ni une classe abstraite.
109	Dans le cas où un message dans un diagramme de séquence fait référence à une opération, cette opération ne doit pas être abstraite.
110	Si un message dans un diagramme de séquence fait référence à une opération, via la signature du message, alors cette opération doit appartenir, selon le diagramme de

	classes, à la classe qui type la ligne de vie cible du message.
111	Les interactions entre les objets dans un diagramme de séquence, en particulier le nombre de types d'objets en interaction, doivent respecter les restrictions de multiplicité spécifiées par le diagramme de classes (par exemple, les multiplicités de fin d'association)[34]
112	Pour que les objets échangent des messages dans un diagramme de séquence, l'objet d'envoi doit avoir un handle vers l'objet de réception comme spécifié dans le diagramme de classes. Une autre façon de dire cela est que l'expéditeur doit avoir une visibilité sur le destinataire. Un cas particulier de cette La situation est lorsque la classe de l'objet émetteur a une association (éventuellement héritée) avec la classe de l'objet récepteur.
113	La sémantique comportementale d'une association de composition ou d'agrégation dans le diagramme de classes doit être déduite dans les diagrammes de séquence. Par exemple, dans une relation tout-partie (composition), la partie ne doit pas survivre au tout
114	Chaque méthode publique dans un diagramme de classes déclenche un message dans au moins un diagramme de séquence.
115	Chaque classe du diagramme de classes doit être instanciée dans un diagramme de séquence.[7]
116	Aucune opération ne peut être utilisée dans un message d'un diagramme de séquence si cela enfreint les règles de visibilité du diagramme de classes (public, protected, private).
Diagramme de séquence et diagramme de communication (pas de règle)	
Diagramme de classes, diagramme de machine à états et diagramme de communication (pas de règle)	
Diagramme de classes, diagramme de séquence et diagramme de cas d'utilisation (pas de règle)	

Tableau 5 – les règles de cohérence

3.6 Conclusion

Ces dernières années, un grand nombre de règles de cohérence UML ont été proposées par les chercheurs afin de détecter les incohérences entre les diagrammes UML. Cependant, aucune étude précédente n'a, à notre connaissance, résumé et analysé ces règles de cohérence UML en suivant un processus systématique. le chapitre 3 a présenté les résultats obtenus après avoir suivi un examen systématique protocole, dont le but était d'identifier, de présenter et d'analyser un ensemble consolidé de 116 règles de cohérence UML issues de la littérature, Nous programmerons dans ocl les règles énoncées dans ce chapitre dans le chapitre suivant.

CHAPITRE

4

Utilisation et réalisation des règles

Sommaire

4.1 Introduction

l'étape suivante consiste à valider l'ensemble de règles. En effet, même si nous collectons très soigneusement les règles de cohérence UML, il n'y a aucune garantie complète que nous n'avons pas fait une erreur de règles manquées. Dans le domaine du génie logiciel empirique, cela est considéré comme une menace pour la validité de nos résultats. Nous devons donc vérifier nos résultats, c'est-à-dire valider l'ensemble des règles de cohérence UML que nous avons obtenues. Pour les besoins de cette recherche, nous avons la principale question de validation suivante : comment vérifier si les règles de cohérence UML sont pertinentes ? Pour atteindre cet objectif, notre méthodologie de validation sera présentée dans ce chapitre

4.2 Décorateur pattern

Dans la programmation orientée objet, le modèle de décorateur est un modèle de conception qui permet d'ajouter un comportement à un objet individuel, de manière dynamique, sans affecter le comportement d'autres objets de la même classe.[62] Le modèle de décorateur est souvent utile pour adhérer au principe de responsabilité unique, car il permet de répartir les fonctionnalités entre des classes avec des domaines de préoccupation uniques.[49] L'utilisation du décorateur peut être plus efficace que le sous-classement, car le comportement d'un objet peut être augmenté sans définir un objet entièrement nouveau.

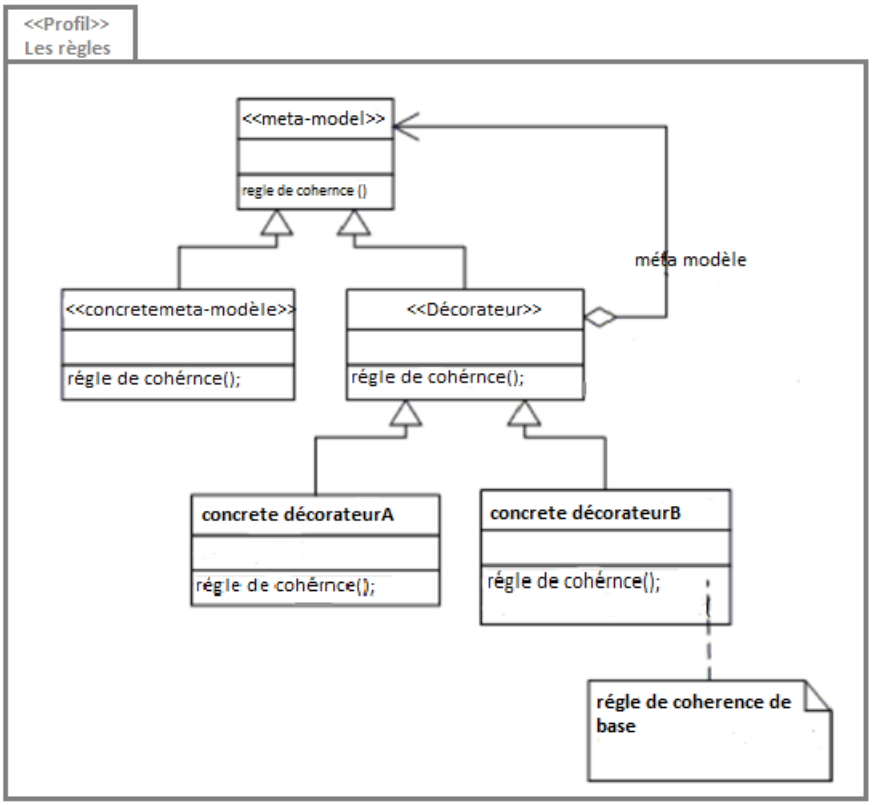

Figure 4.1– noter model (Décorateur Pattern)

- Composant est une interface qu'implémentent à la fois l'objet décoré et les objets qui le décorent
- Concret Composant représente l'objet décoré (qui implémente Component)
- Décorateur représente l'interface des décorateurs On trouve ensuite les décorateurs concrets (Concrete décorateurA , etc.)

4.3 Environnement de logiciel

Eclipse Papyrus :

Eclipse Papyrus est un outil d'édition graphique pour UML 2 tel que défini par OMG. Eclipse Papyrus vise à implémenter 100% de la spécification OMG !Eclipse Papyrus fournit des éditeurs pour tous les diagrammes UML :

- Diagramme de classes
- Diagramme d'objet
- Diagramme de paquet
- Diagramme de structure Composite
- Diagramme des composants
- Diagramme de déploiement

- Diagramme de profil
- Diagramme de cas d'utilisation
- Diagramme d'activité
- Diagramme de la machine à états
- Diagramme de communication
- Diagramme de séquence
- Diagramme de timing
- Aperçu de l'interaction Diagramme

Eclipse Papyrus fournit également un support complet à SysML[106] afin de permettre l'ingénierie système basée sur des modèles. Des éditeurs tabulaires et graphiques spécifiques requis pour SysML sont également fournis :

- Diagramme de définition de bloc
- Schéma fonctionnel interne
- Diagramme des exigences
- Diagramme paramétrique
- Tableau des exigences
- Tableau de répartition

4.4 Architecture de vérification des diagrammes uml

Le System de noter thèse de mémoire de vérification des diagrammes uml C'est qu'il n'y a parfois pas de cohérence dans les diagrammes uml , La question est maintenant de savoir comment traiter et résoudre ce problème d'incohérence.

La solution que nous avons proposée et essayée de résoudre le problème d'incohérence est l'ocl , Grâce à quoi nous pouvons établir des règles de cohérence qui guident l'utilisateur et l'empêchent de tomber dans les erreurs d'incohérence dans les diagrammes uml , C'est notre objectif dans ce mémoire .

Aussi, notre objectif est de qui consiste à étudier dans quelle mesure les règles de cohérence UML les plus pertinentes que nous avons trouvées (voir chapitre3) sont satisfaites dans les modèles UML réels. Cela montrera comment un concept spécifique, tel qu'il est couvert par une règle OCL spécifique.

dans la figure 4.2 , nous présentons brièvement les trois principales étapes que nous avons suivies pour réaliser l'étude de cas rapportée dans ce mémoire de master :

1) Sélection et encodage des règles de cohérence UML en OCL .
   Dans cette étape , les règles définies sont converties en un code ocl.
2) Collecte des projets de modèles UML .
   Dans cette étape , nous collectons de projets modelés uml , puis sélection de projets de papyrus pour choisir des projets appropriés
3) Vérification des règles OCL sur les projets de modèles UML .
   Dans cette étape , nous effectuons un processus d'importation de modèles et règles pour la vérification des règles ocl sur le modèle.

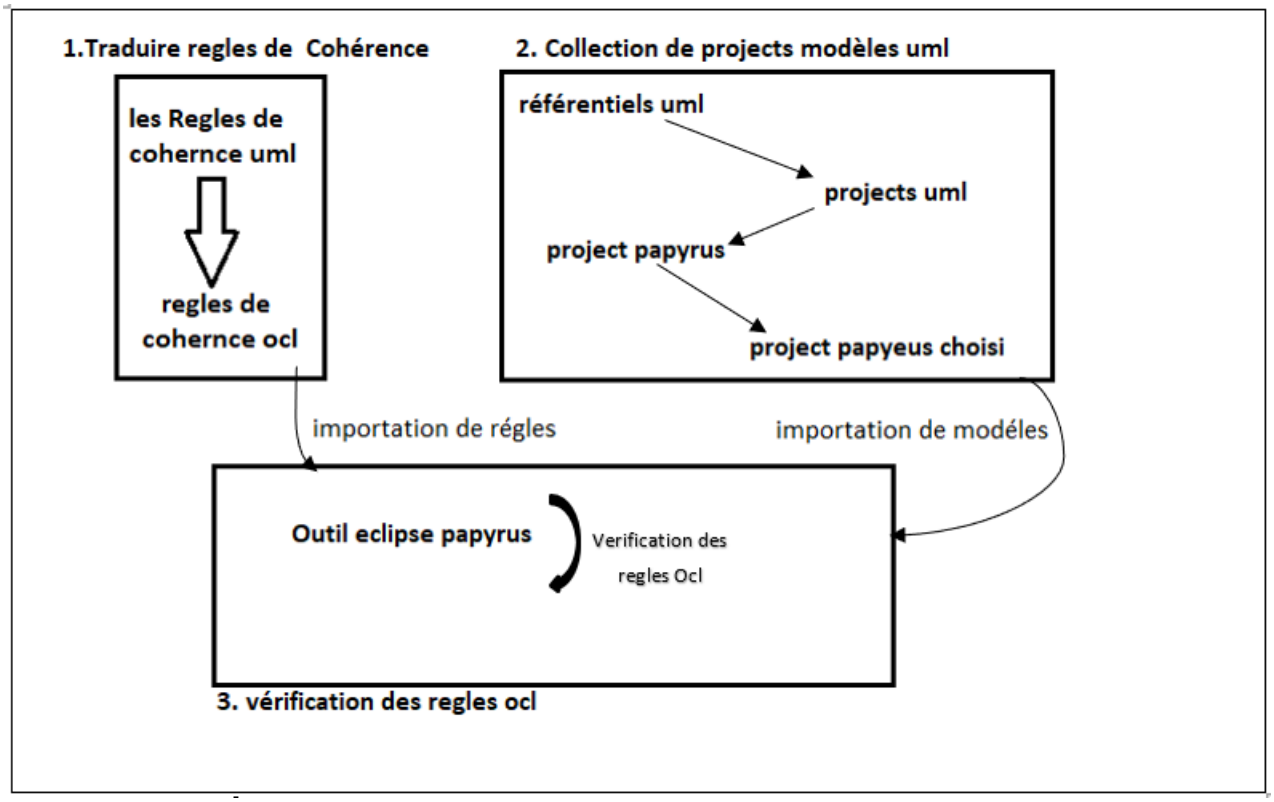

Figure 4.2– les étapes pour Vérifier les règles ocl dans Eclipse papyrus [65]

4.5 Implémentation

Le but de cette application est d'aider les utilisateurs à ne pas faire d'erreurs dans la modélisation des erreurs d'incohérence dans les diagrammes. Pour atteindre notre objectif, nous appliquons les règles de cohérence mentionnées dans le troisième

chapitre, en les convertissant en règles ocl qui alertent l'utilisateur des erreurs qu'il a commises, bien sûr les erreurs d'incohérence.

4.6 Règles de cohérence UML

En commençant par l'ensemble des règles de cohérence UML identifiées dans le chapitre 3 , le principal objectif de recherche de cette section est d'étudier dans quelle mesure les modèles UML existants satisfont à ces règles. Pour ce faire, dans cette section, nous décrivons une étude de cas exploratoire flexible indirecte [32] pour évaluer les règles de cohérence UML sur les modèles UML où nous rapportons :

1) l'encodage des règles de cohérence UML [39] comme invariants dans le langage OCL (Object Constraint Language) [33]. A partir de maintenant dans cette section, les invariants de cohérence OCL seront simplement appelés règles OCL.

2) la sélection des modèles UML à utiliser.

3) la méthodologie utilisée pour importer les modèles UML dans l'Outil Eclipse Papyrus pour les vérifier avec les règles OCL .

4) les résultats obtenus en vérifiant les modèles UML avec les règles OCL.

Dans cette étude de cas, nous effectuons un type de vérification de la cohérence UML où nous concentrons sur la détection d'incohérences entre (ou dans un seul) diagramme UML d'un modèle UML .

4.6.1 Travaux connexes

A notre connaissance, nous n'avons connaissance d'aucun travail antérieur présentant une étude de cas impliquant des règles de cohérence UML, qui ont été systématiquement collectées dans le troisième chapitre, évaluées par des experts de l'industrie et du monde universitaire, et encodées en OCL et vérifiées systématiquement sur un nombre relativement important de modèles UML open source. Néanmoins, en tant que travail connexe, nous avons décidé de rassembler des articles qui discutent des règles OCL utilisées pour vérifier la cohérence entre les diagrammes UML. Notez que nous omettons donc les travaux connexes qui traitent de la cohérence (règles) des modèles (UML). Pour une discussion plus complète de ce dernier sujet, dans le tableau 6 ,nous présentons un ensemble de travaux qui correspondent à la description des travaux connexes pour les travaux présentés dans cette section. Pour chacun, le tableau indique l'année de publication de l'ouvrage, si l'article implique une étude de cas, le nombre de règles OCL discutées dans l'article, et enfin le nombre de projets modèles concernés.

Auteurs	OCL règles	Modèles
Reder et Egyed [51]	20	29
Gogolla et al. [52]	10	18
Czarnecki et Pietroszek [53]	1	1
Liu et al.[54]	1	aucun
Sapna et Mohanty[55]	1	aucun

Tableau 6 – Résume de travaux connexes

Reder et Egyed [81] ont présenté une approche pour identifier comment les règles de conception provoquent une incohérence donnée et quels éléments du modèle sont impliqués. Ils ont démontré que leur approche identifie correctement les causes en validant 29 modèles UML par rapport à un ensemble de 20 règles de conception OCL, où trois règles sur 20 ont également été prises en compte dans l'ensemble des 52 règles de cohérence.

Gogolla et al. [66] rapportent une étude de cas de taille moyenne, dans laquelle la cohérence d'un modèle de classe UML est vérifiée. Les restrictions du modèle de classe sont exprimées par des contraintes de multiplicité UML et règles OCL explicites et non triviales. Leur approche construit automatiquement un état système valide qui montre si un modèle de classe peut être instancié et l'échec de l'instanciation du modèle tout en satisfaisant les règles OCL est une indication d'incohérence.

Liu et al. [73] décrivent une solution basée sur des règles pour détecter un problème d'incohérence de conception logicielle. Ils ont caractérisé les classes d'incohérence qui se produisent dans la conception de logiciels. Ils ont défini un langage de système de production et des règles spécifiques aux conceptions logicielles modélisées en UML. En utilisant cette approche, ils détectent les incohérences, informent les utilisateurs, recommandent des résolutions et corrigent automatiquement les incohérences pendant le processus de conception.

Sapna et Mohanty [75] traitent les incohérences structurelles entre les diagrammes de cas d'utilisation, d'activité, de collaboration, de machine à états et de classes en utilisant des règles OCL converties en déclencheurs SQL applicables dans les tables qui stockent les diagrammes UML.

Czarnecki et Pietroszek [83] utilisent OCL pour définir des règles de bonne formation pour la vérification des modèles de modèles basés sur les caractéristiques qui sont analysés par un solveur SAT. Les règles peuvent être évaluées par rapport à un modèle à l'aide d'une interprétation de modèle pour OCL. Ils ont présenté un prototype d'outil qui a été testé sur un modèle économique pour une plateforme de commerce électronique.

4.6.2 Etude d'un cas

Le processus de recherche d'une étude de cas peut être caractérisé comme fixe ou flexible [56, 57]. Dans un processus de conception fixe, tous les paramètres sont définis au lancement de l'étude, tandis que dans un processus de conception flexible, les paramètres clés de l'étude peuvent être modifiés au cours de l'étude [65]. Dans cette section, nous décrivons une étude de case car nous voulons savoir ce qui se passe dans ce contexte (c'est-à-dire comment nos règles OCL sont vérifiées sur les modèles UML), rechercher de nouvelles informations concernant les incohérences dans différents types de diagrammes UML et générer des idées et des hypothèses pour de nouvelles recherches ;b) flexible car les paramètres clés de notre processus de conception ont été modifiés au cours de l'étude, l'incohérence est vérifiée en se basant sur les machines à états et les diagrammes des classes.

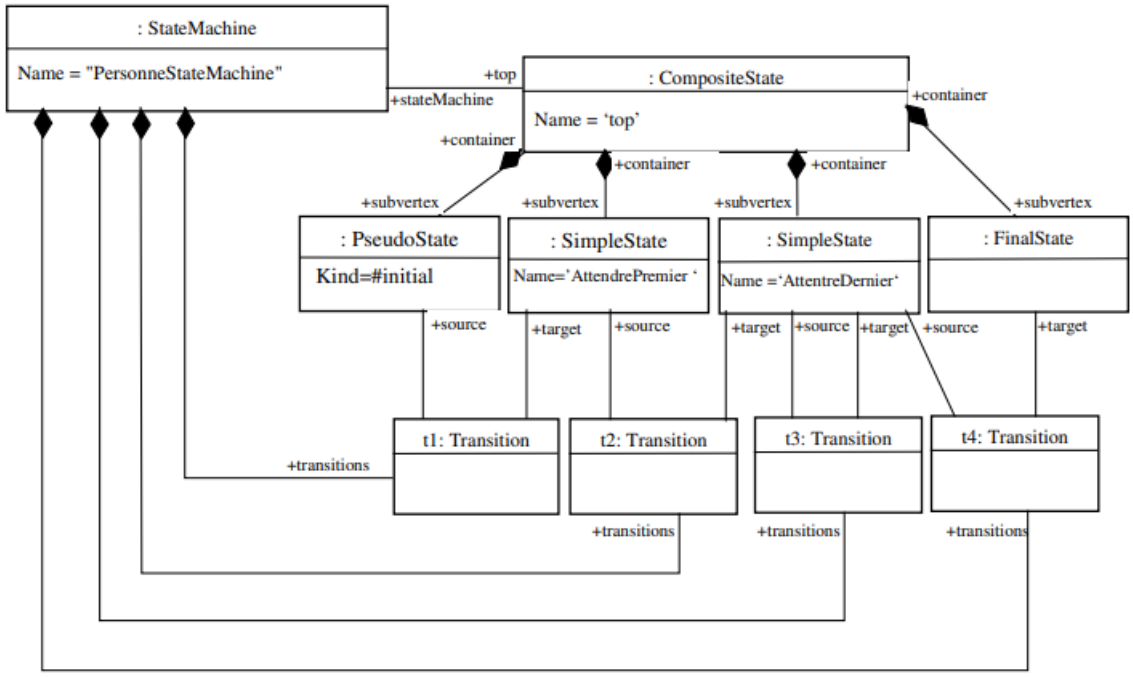

Figure 4.3– instance de métamodèle uml

4.6.2.1 Encodage des règles de cohérence UML en OCL.

L'ensemble de cohérence UML que nous avons présenté dans le chapitre 3 a été évalué par des experts mais nous n'avons pas pu encoder quelques-unes de ces règles en OCL pour différentes raisons :

a) OCL n'est pas assez expressif pour coder toutes ces règles (par exemple, nécessite beaucoup de traitement) ;

b) dans certains cas, La norme UML n'inclut pas les informations nécessaires pour vérifier les règles (par exemple, nécessite des informations spécifiques dans les descriptions de cas d'utilisation).

c) Certaines des règles que nous avons exclues étaient :

1) une machine à états doit être sans blocage ;

2) pour une action d'envoi, il doit y avoir une réception dans le classificateur de l'instance de récepteur qui correspond au signal de l'action d'envoi décrivant la réponse comportementale attendue au signal ;

d) les classes d'entités correspondent aux données manipulées par le système telles que décrites dans une description de cas d'utilisation ;

d) le nom d'un cas d'utilisation doit inclure un verbe et un nom (par exemple « Valider l'utilisateur »). en particulier ,Ci-dessous, nous présentons un exemple des règles encodées dans OCL.

Règle 77 : Une opération abstraite ne peut appartenir du'a une classe abstraite

```
inv Rule77:
    (postcondition -> includes (self) or self.precondition -> includes (self)) implies
        self.isQuery=true
```

Figure 4.4 – ocl règles de cohérence / règle 77

Règle 39 : Si un diagramme d'activités présente des scénarios d'une opération et que cette opération apparaît dans un diagramme de séquence, les différents diagrammes doivent spécifier les mêmes scénarios : par exemple, même séquence de messages/opérations/actions, mêmes conditions de branchement ou de répétition.

```
inv Rule39: (
   (self.name = UML::Action.name) and (self.name = UML::Message.name)
   implies UML::Message.name = UML::Action.name
)

context uml::Message
```

Figure 4.5 – ocl règles de cohérence/ règle 39

Règles 112 et 55 : Pour que les objets échangent des messages dans un diagramme de séquence, l'objet d'envoi doit avoir un handle vers l'objet de réception comme spécifié dans le diagramme de classes. Une autre façon de dire cela est que l'expéditeur doit avoir une visibilité sur le destinataire. Un cas particulier de cette La situation est lorsque la classe de l'objet émetteur a une association (éventuellement héritée) avec la classe de l'objet récepteur.

```
inv Rule_112_55: (
    self.receiveEvent.oclAsType(InteractionFragment).covered->exists (
        let rc:Class=represents.type.oclAsType(Class)
        in self.sendEvent.oclAsType(InteractionFragment).covered->exists(let sc:Class=represents.type.oclAsType
        in sc.ownedAttribute->exists(association<>null implies type=rc)
        )
        )
    )
```

Figure 4.6 – ocl règles de cohérence / règle 55_112

Règle 109 : Dans le cas où un message dans un diagramme de séquence fait référence à une opération, cette opération ne doit pas être abstraite.

```
inv Rule109 : (
   if(self.name = uml::Operation.name)
   then (not uml::Operation.isAbstract)
   else false
   endif
)
```

Figure 4.7 – ocl règles de cohérence règle 109

Règles 46_48 : Lorsqu'on spécifie une classe active, c'est-à-dire qui a un comportement basé sur un état décrit dans un diagramme de machine à états, et qu'une instance de cette classe active est utilisée dans un diagramme de séquence, les messages envoyés à cet objet et émis par cet objet comme spécifié dans le diagramme de séquence doit être conforme (par exemple, séquence et types de signaux, récepteurs et émetteurs de signaux) au protocole spécifié dans le diagramme de la machine d'état.

```
inv Rule_46_48: (
  if (InteractionFragment.name = UML::Class.name) and  (InteractionFragment.name = UML::State.name)
      then self.receiveEvent.oclAsType(InteractionFragment).covered->exists (
          let rs:State=represents.type.oclAsType(State)
              in self.sendEvent.oclAsType(InteractionFragment).covered->exists(let ss:State=represents.
              in ss->exists(ss.oclIsKindOf(ProtocolStateMachine) and rs.oclIsKindOf(ProtocolStateMachi
              )
      )
else false
endif
)
```

Figure 4.8 – ocl règles de cohérence règle 46_48

Règle 13 : A state machine must be deterministic, that is, in every state, only one transition (accounting for the different levels of nested states) should fire on a reception of an event

```
inv Rule13a: (
    (self->forAll(s|s.Transition.source->size() = 1)) and (self->forAll(t|t.Transition.target->size() = 1))
)

context uml::State

inv Rule13b: (
    (self->forAll(s|s.entry->size()=1)) and (self->forAll(s|s.exit->size()=1))
)

context uml::UseCase
```

Figure 4.9 – ocl règles de cohérence / règle 13

Règle 9 : Considérez deux State Machine U' et U d'une classe O' et sa superclasse O, où U' affine le diagramme d'état de U au moyen d'une fonction de raffinement h qui mappe les transitions sur les transitions et les états de U' et qui mappe les états simples de U' sur les états simples de U. Intuitivement, le concept de la fonction de raffinement signifie que si un état simple s de U a été affiné à un état composite en U'', alors h mappe les sous-états de s en U' et les transitions entre ces états à s, h met en correspondance les transitions en U qui sont des incidents aux sous-états de s et en transitions de U qui sont des incidents à s.

```
inv Rule9:(
    self.extendedStateMachine->notEmpty()
        implies ((let esm = self->selectByKind(uml::StateMachine) in esm->exists(sm:StateMachine
            | esm.extendedStateMachine.Transition.source = sm.Transition.source
        ))
                and (let esm = self->selectByKind(uml::StateMachine) in esm->exists(sm:StateMachine
                    | esm.extendedStateMachine.Transition.target = sm.Transition.target
                )))
)
```

Figure 4.10 – ocl règles de cohérence / règle 9

Cette étape de sélection et d'encodage des règles a finalement fourni 33 règles OCL pour vérifier la cohérence entre les différents diagrammes UML. Les règles OCL ont été enregistrées dans une bibliothèque appelée « règles de cohérence uml . ocl », qui était incluse dans le répertoire Eclipse Papyrus « Règles Validation » que nous avons dédié à ce travail (voir Figure 4.11).

Certaines des règles (voir les règles dans le chapitre 3) ont été fusionnées dans une règle OCL unique lorsqu'elles ont été codées en OCL. Par exemple, la règle 54 (les objets impliqués dans un diagramme de communication doivent être des instances de classes du diagramme de classes) et la règle 115 (chaque classe du diagramme de classes doit être instanciée dans un diagramme de séquence), impliquent le diagramme de communication (COMD) et la classe Diagramme (CD), et le diagramme de classes (CD) et la séquence Diagramme (SD) respectivement. Dans Eclipse Papyrus, ces deux règles se concentrent toutes les deux sur UML::Class (pour CD) et UML::Interaction Fragment (pour COMD et SD). Produire deux règles OCL distinctes aurait conduit à deux règles identiques. Pour cette raison, les deux règles ont plutôt été fusionnées en une seule règle OCL.

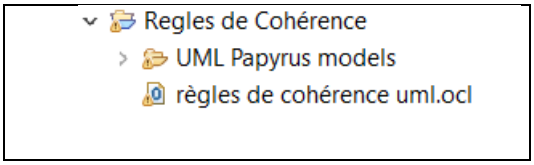

Figure 4.11 – règles de cohérence uml fichier

De plus, pour que certaines des 33 règles (voir les règles dans le chapitre 3) soient encodées en OCL, nous avons dû résumer (c'est-à-dire simplifier) leur sens original fourni en anglais simple lorsque nous les avons encodées en OCL car d'OCL comme langage de contraintes. Bien que de nouvelles opérations et même de tout nouveaux types de types puissent être ajoutés à OCL, les mécanismes ont leurs limites. On ne peut pas changer les fondements sous-jacents de la langue. Il n'est pas possible de faire en sorte que les expressions OCL aient un effet secondaire, ou dicter un algorithme à appliquer lorsqu'une certaine expression est évaluée . Un exemple plus simple de la procédure d'abstraction/simplification que nous appliqué est pour la règle 115 qui spécifie que chaque classe du diagramme de classes doit être instanciée dans un diagramme de séquence.

Pour pouvoir coder la règle 115, nous l'avons simplifiée (c'est-à-dire que nous ne prenons pas en compte le noms qualifiés) en OCL comme suit :

```
inv Rule_115: (
    self->exists(self.name = UML::Class.name)
)
```

Figure 4.12 – ocl règle de cohérence / règle 115

Notez que la règle 115 est censée s'appliquer aux modèles complets ; d'autre part, les modèles que nous avons collectés dans cette étude de cas peuvent ne pas être complets. Nous pensons que la règle suppose également que l'on vérifie un type spécifique de modèles. Plus précisément, il n'est pas réaliste pour un modèle de conception, qui montre dans le diagramme de classes comment les modèles de conception sont utilisés, d'indiquer comment les classes qui sont spécifiques aux modèles de conception interagir dans un diagramme de séquence : un modèle de conception n'est pas censé satisfaire la règle. D'un autre côté, nous pensons qu'il est plus logique qu'un modèle d'analyse satisfasse à la règle.

.

4.6.2.2 Vérification des règles OCL

L'outil UML que nous avons utilisé est Papyrus 2.0.X (développé sur Eclipse Papyrus 2021), qui était la version Eclipse recommandée à utiliser avec Papyrus . l'installation du plug-in supplémentaire suivant était requise :

- OCL Classic 2 SDK.
- Ecore/UML Parsers.
- Evaluator et Edit (version 5.2).

Cette pile d'outils (Eclipse Papyrus + Plug-in) a été choisie car elle offre une vue de validation spécifique qui permet de vérifier les résultats des règles OCL. Les utilisateurs peuvent également générer un plug-in à partir d'un profil qui intègre les règles OCL créées .Cela dit, les trois étapes suivantes résument et décrivent comment nous avons vérifié plusieurs projets de modèles UML avec nos règles OCL.

1. nous ouvrons le fichier « .uml » de chaque modèle UML Papyrus (voir Figure 4.12)

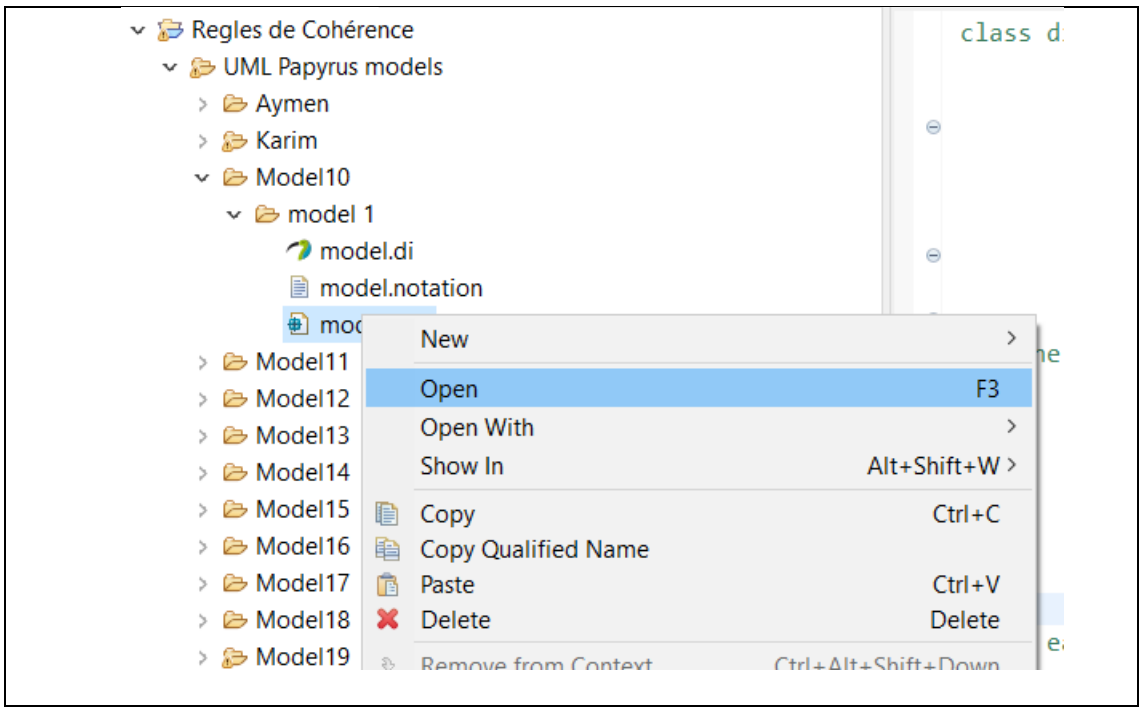

Figure 4.12 – étape 1 ouvrir le model

2. puis depuis le nœud racine du modèle « .uml », nous avons sélectionné dans le menu contextuel OCL->Charger le document (voir Figure 4.13).

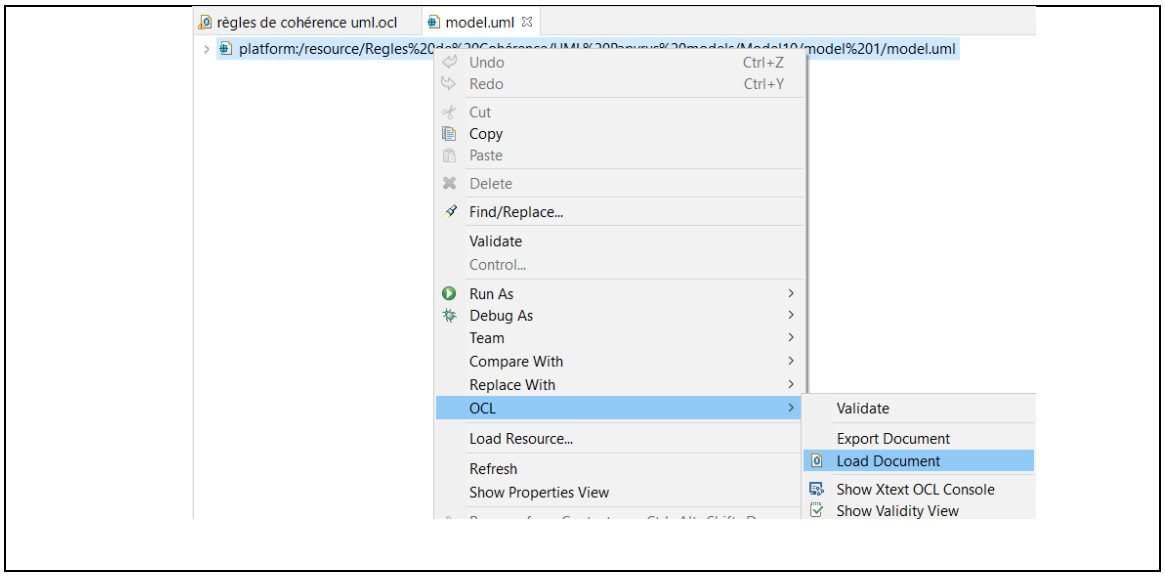

Figure 4.13 – étape 2 charger le document de règles ocl

3. sélectionné notre bibliothèque ‘règles de cohérence uml.ocl’ (voir Figure 4.14) Cette étape a ajouté les règles OCL définies dans notre bibliothèque à l'ensemble des règles de bonne formation UML déjà implémentées dans Eclipse (basées sur la spécification UML).

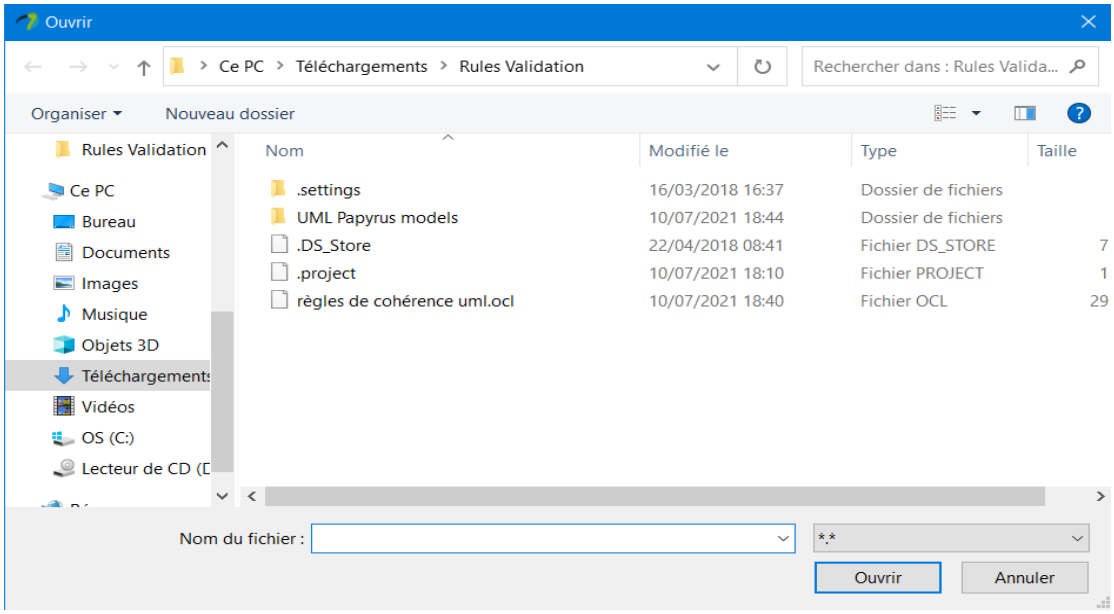

Figure 4.14 – étape 4 Sélectionné notre fichier ocl

4. enfin, pour exécuter les règles OCL, c'est-à-dire valider le modèle chargé, à partir du nœud racine du modèle « .uml », nous avons sélectionné Valider (voir Figure 4.15). Cette dernière étape a exécuté toutes les règles de bonne formation du standard UML ainsi que nos règles OCL.

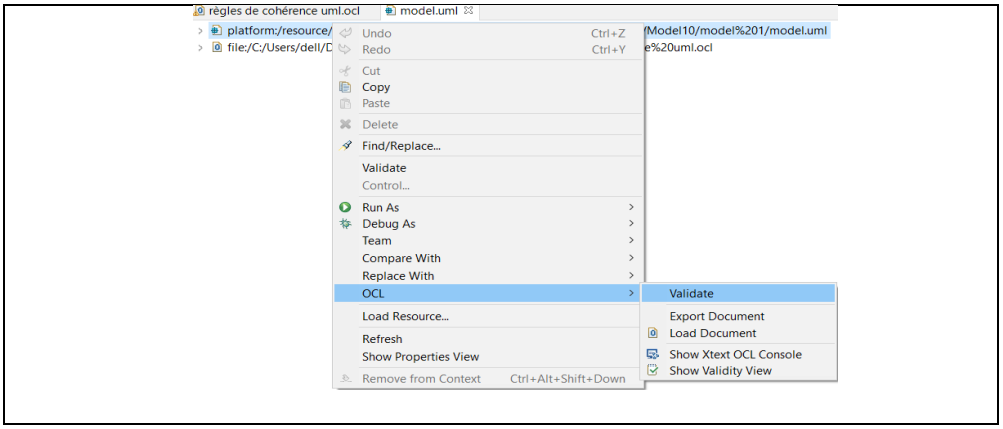

Figure 4.15 – étape 4 Validation de modèle avec les règles ocl

4.7 Conclusion

Ce travail présente les résultats d'une étude de cas impliquant un ensemble de règles OCL qui ont été utilisées pour vérifier la cohérence des diagrammes dans des modèles UML Papyrus. A partir de l'ensemble des UML règles de cohérence identifiées dans ce chapitre, l'objectif de cette étude de cas était d'étudier dans quelle mesure les modèles UML existants satisfont à ces règles. Après l'exécution des règles OCL sur les modèles, nous nous sommes rendu compte que seulement 30 règles OCL et 31 modèles UML Papyrus étaient finalement utilisés,. Une première conclusion que nous pouvons tirer est que, malgré le fait que les règles OCL que nous avons utilisées ont été validées par des experts dans le domaine qui ont indiqué que les règles doivent toujours être appliquées, c'est-à-dire que tout modèle UML doit satisfaire aux règles, nous observons que les incohérences sont nombreuses en open source modèles que nous avons utilisés. Les principales contributions de cette section sont la description du processus de vérification des règles OCL sur modèles UML dans Papyrus et évaluation de 33 de ces règles pour détecter différentes incohérences entre les diagrammes de modèles UML. Ce travail peut ainsi être considéré comme une contribution à la définition des meilleures pratiques de l'ingénierie logicielle Model-Driven basée sur UML. Les concepteurs peuvent utiliser les informations fournies par ce travail pour mieux comprendre les incohérences UML.

Conclusion Générale

A l'issue de ce travail qui vise à aider à préparer une application qui vérifié et détecte les erreurs de cohérence.

Pour la réalisation de ce système, il nous a fallu adopter une méthodologie de travail. nous avons utilisé la technique de modélisation objet UML pour modéliser de façon standard le problème sous étude. Il en va de même du premier chapitre, dans lequel nous nous sommes initialement concentrés sur UML et ses propriétés, avantages et diagrammes, puis avons mentionné certains des avantages et des inconvénients de la modélisation orientée objet. Ensuite, nous avons introduit les définitions uml et le langage de contraintes OCL.

Dans le deuxième chapitre, nous nous sommes concentrés sur la cohérence, Pour cela, nous avons commencé par la définition de la cohérence , par la suite nous avons vu les types de la cohérence et leur définition et exemple , ensuite nous avons présenté la Classification de cohérence UML et les paramètres d'analyse ,et nous avons cité quelques Techniques de contrôle de cohérence .et pour contrôlé la cohérence de les diagrammes uml.

Dans le troisième chapitre nous avons présenté les règles de cohérence et leur importance et leur nombre dans chaque Diagramme et entre Diagrammes, et nous avons donné un ensemble de 116 règles de cohé8rence dont nous pouvons bénéficier et chaque Diagramme nous avons donné ses propres règles et nous avons également donné les règles entre les diagrammes. Dans le quatrième chapitre, en général, nous avons utilisé les règles de cohérence à travers notre profil UML et nous avons converties en un code OCL sous forme de conditions et de contraints ,Nous avons également présenté comment vérifier le modèle à travers les règles de cohérence programmées.

Bibliographie

[1] J. Gabay et D. Gabay, UML2 Analyse et Conception, Groupe Dunod, ISBN : 978-2-10-053567- 5,2008

[2] Sinan Si Alhir , Guide to Applying the UML .

[3] Eskandar Kouicem, Azza Dridi, Med Nadir Boukelal, Université de Constantine 2 - Licence en informatique 2016..

[4] Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. World Scientific Pub. Co., New Jersey (2001).

[5] P. A. Muller et N. Gaertner, Modélisation objet avec UML, Groupe Eyrolles, ISBN : 2-212- 11397- . 8, 2004

[6]. James Rumbaugh, Ivar Jacobson,Grady Booch : The Unified Modeling Language Reference Manual, Second Edition ,2004

[7] M. Hilsbos and I.-Y. Song, "Use of Tabular Analysis Method to Construct UML Sequence Diagrams,".

[8] OMG. UML 2.0 Infrastructure Spécification, September, 2003. http://www.omg.org/docs/ptc/03- . 09-15.pdf

[9] OMG. UML 2.0 OCL Specification, October, 2003 http://www.omg.org/docs/ptc/03-10-14.pdf

[10] J.L. Sourrouille and G. Caplat. A Pragmatic View on Consistency Checking of UML Models.

[11] B. Hnatkowska, Z. Huzar, L. Kuzniarz and L. Tuzinkiewicz. Refinement relationship between collaborations.

[12] L. Kuzniarz, G. Reggio, J. Sourrouille and Z. Huzar. Workshop on Consistency Problems in UML-based . Software Development. UML 2002. Blekinge Institute of Technology. Research Report 2002:06.

[13] H. Gomaa and D. Wijesekera. Consistency in Multiple-View UML Models: A Case Study.

[14] L. Kuzniarz, G. Reggio, J. Sourrouille and Z. Huzar and M. Staron. Workshop on Consistency Problems in UML-based Software Development II. UML 2003. Blekinge Institute of Technology. Research Report 2003:06.

[15] L. Kuzniarz and M. Staron. Inconsistencies in Student Designs.

[16] T. Feng and H. Vangheluwe. Case Study: Consistency Problems in a UML Model of a Chat Room.

[17] B. Hnatkowska, Z. Huzar, L. Kuzniarz and L. Tuzinkiewicz. A systematic approach to consistency within . . UML .based software development process.

[18] Y. Shaham-Gafni and S. Kremer-Davidson. Consistency in UML 2.0 – Classes vs. Objects. IBM Haifa . Research Lab.

[19] S. Kremer-Davidson and Y. Shaham-Gafni. UML 2.0 Model Consistency – The Rule of Explicit and . Implicit Usage Dependencies, IBM Haifa Research Lab.

[20] H. Gomaa. Designing Concurrent, Distributed and Real-Time Applications with UML. Addison-Wesley . . Object Technology Series, ISBN: 0-201-65793-7, August 2000.

[21] P. Kruchten. The Rational Unified Process: An Introduction (2nd Edition). Addison Wesley Longman Inc1999.

[22] J.L. Sourrouille and G. Caplat. A Pragmatic View on Consistency Checking of UML Models.

[23] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick and T. Grose. Eclipse Modeling Framework. Addison-. Wesley Professional, August 2003.

[24] J. Hausmann, R. Heckel, and S. Sauer. Extended Model Relations with Graphical Consistency . Conditions.

[25] W. Liu, S. Easterbrook, and J. Mylopoulos. Rules Based detection of Inconsistency in UML Models.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, AddisonWesley, ISBN: 0-201-63361-2,1994.

[27] M. Genero, A. M. Fernández-Saez, H. J. Nelson, G. Poels, and M. Piattini, "A Systematic Literature . . Review on the Quality of UML Models

[28] G. Spanoudakis and A. Zisman, "Inconsistency management in software engineering: Survey and open . research issues," in Handbook of Software Engineering and Knowledge Engineering, S. K. Chang, Ed ed Singapore: World Scientific Publishing Co., 2001, pp. 329-380.

[29] J. Chanda, A. Kanjilal, S. Sengupta, and S. Bhattacharya, "Traceability of Requirements and Consistency Verification of UML Use case, Activity and Class Diagram: A Formal Approach,".

[30] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille, "Consistency problems in UMLbased software . . development," presented at the International Conference on UML Modeling Languages and .Applications, Lisbon, Portugal, 2005

[31] A. Egyed : UML/ANALYZER: A Tool for the Instant Consistency Checking of UML Models, In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), pp. 793-796, May 2007, Min

[32] P. Runeson, M. Host, A. Rainer, and B. Regnell., Case Study Research in Software Engineering: Guidelines and Examples 1st ed.: Wiley Publishing, 2012.

[33] OMG. (2016). Object Management Group - Object Constraint Language (OCL).

[34] M. Hilsbos and I.-Y. Song, "Use of Tabular Analysis Method to Construct UML Sequence Diagrams,".

[35] G. Engles, J. H. Hausmann, R. Heckel, and S. Sauer, Testing the Consistency of Dynamic UML diagrams

[36] M. Grischick : Difference Detection and Visualization in UML Class Diagrams, Department of Computer Science on Metamodeling and its Application,

[37] B. Graaf and A-V. Deursen, Model-Driven Consistency Checking of Behavioral Specifications

[38] N. Ibrahim, R. Ibrahim, M. Z. Saringat, D. Mansor, and T. Herawan, "Consistency rules between UML use case and activity diagrams using logical approach.

[39] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for UML diagrams," Journal of Systems and Software, vol. 144, pp. 121- 142, 2018.

[40] G. Engels, J. M. Kuster, and L. Groenewegen : A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models

[41] H. Rasch and H. Wehrheim : Consistency Checking in UML Diagrams: Classes and State Machines

[42] S-K. Kim and D. Carrington : A Formal ObjectOriented Approach to defining Consistency Constraints for UML Models

[43] Y. Shinkawa; Inter-model Consistency in UML Based on CPN Formalism, In Proceedings of the 13th Asia Pacific Software Engineering Conference

[44] T. Mens, R. V-D. Straeten, and J. Simmonds : A Framework for Managing Consistency of Evolving UML Models, In H. Yang, editor, Software Evolution with UML and XML, chapter 1. Idea Group Inc., March 2005.

[45] J. Simmonds, R. V. Straeten, V. Jonkers, and T. Mens, "Maintaining Consistency between UML Models . . using Description LogicZ," RSTI – L'Object LMO'04, vol. 10, pp. 231-244, 2004.

[46] G. Reggio, M. Leotta, F. Ricca, and D. Clerissi, "What are the used UML diagrams? A Preliminary Survey," presented at the In Proceedings of 3rd International Workshop on Experiences and Empirical Studies in Software Modeling - CEUR Workshop Proceedings (EESSMod 2013), Miami, Florida - USA, 2013.

[47] Z. Liu, X. Li, J. Liu, and J. He : Integrating and refining UML models. Technical Report 295, UNU/IIST, PO Box 3058, Macao SAR China, 2004. Presented at UML 2004 Workshop on Consistency Problems in UML-based Software Development, October 10-15, 2004, Lisbon, Portugal.

[48] T-H. Feng and H. Vangheluwe : Case Study: Consistency Problems in a UML Model of a Chat Room, In Proceedings of the 6 th International Conference on the Unified Modeling Language (UML'03), October 2003. San Francisco, USA.

[49] Transitioning Systems Thinking to Model-Based Systems Engineering: Systemigrams to SysML Models

[50] A. Egyed : Instant Consistency Checking for the UML, In Proceedings of the 28th International Conference on Software Engineering (ICSE'06), May 2006, Shanghai, China. ACM Press.

[51] A. Reder and A. Egyed, "Determining the Cause of a Design Model Inconsistency," IEEE Trans. Softw. Eng. , vol. 39, pp. 1531-1548, November 2013 2013.

[52] M. Gogolla, L. Hamann, F. Hilken, and M. Sedlmeier, "Checking UML and OCL Model Consistency: An Experience Report on a Middle-Sized Case Study," presented at the International Conference on Tests and Proofs, 2015.

[53] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against wellformedness OCL constraints.

[54] W. Liu, S. Easterbrook, and J. Mylopoulos, "Rule-Based Detection of Inconsistency in UML Models,"

[55] P. G. Sapna and H. Mohanty, "Ensuring consistency in relational repository of UML models,

[56] J. W. Anastas et M. L. MacDonald, Design de recherche pour le travail social et les services sociaux : Lexington, 1994.

[57] C. Robson, Real World Research, 2nd ed.: Blackwell, 2002.

[58] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: data collection techniques for software field studies. ," Empirical Softwtware Engineering vol. 10, pp. 311–341, 2005.

[59] A. Egyed and A. Reder. (2016, last access on August 2016). Examples of Design Rules. Available: http://goo.gl/MJkJnB

[60] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," Empirical Software Engineering,

[61] M. Siikarla, J. Peltonen, and P. Selonen. "Combining OCL and Programming Languages for UML Model Processing. Electron". Notes Theor. Comput. Sci. vol. 102, pp. 175-194. 2004.

[62] Gamma, Erich; et al. (1995). *Design Patterns*. Reading, MA: Addison-Wesley Publishing Co, Inc. pp. 175

[63] L. Fryz and L. Kotulski, "Assurance of System Consistency During Independent Creation of UML Diagrams,".

[64] P. Runeson, M. Host, A. Rainer, and B. Regnell., Case Study Research in Software Engineering: Guidelines and Examples 1st ed.: Wiley Publishing, 2012

[65] T. Maier and A. Zndorf, "The Fujaba Statechart Synthesis Approach," presented at the Workshop on Scenarios and State Machines (ICSE '03), Portland, Oregon, USA, 2003.

[66] T. Ziadi, L. Helouet, and J.-M. Jezequel, "Revisiting Statechart Synthesis with an Algebraic Approach ".

[67] P. Selonen, K. Koskimies, and M. Sakkinen, "How to Make Apples from Oranges in UML," presented at the 34th Annual Hawaii International Conference on System Sciences (HICSS '01), 2001.

[68] J. Schumann, "Automatic debugging support for uml designs," presented at the 4th International Workshop on Automated Debugging, 2000

[69] P. Selonen, K. Koskimies, and M. Sakkinen, "Transformations between UML diagrams,"

[70] 11, pp. 102-107, 2005. [187] J. Whittle and P. K. Jayaraman, "Synthesizing hierarchical state machines from expressive scenario descriptions," ACM Trans. Softw. Eng. Methodol., vol. 19, 2010.[

[72] S. Kang, H. Kim, J. Baik, H. Choi, and C. Keum, "Transformation Rules for Synthesis of UML Activity Diagram from Scenario-Based Specification " presented at the IEEE 34th Annual Computer Software and Applications Conference (COMPSAC '10), 2010.

[73] I. Khriss, M. Elkoutbi, and R. K. Keller, "Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams,"

[74] [85] M. Hilsbos and I.-Y. Song, "Use of Tabular Analysis Method to Construct UML Sequence Diagrams,".

[75]] J. Yang, Q. Long, Z. Liu, and X. Li, "A predicative semantic model for integrating UML models,".

[76] L. Quan, L. Zhiming, L. Xiaoshan, and J. He, "Consistent code generation from UML models,".

[77] H. Il-Kyu and K. Byung-Wook, "Meta-validation of UML structural diagrams and behavioral diagrams with consistency rules,".

[78] X. Xia, J. Shi, Z. Fan, Z. Ai, and Y. Dong, "A Consistency Checking Approach for System Architecture,".

[79] J. Kuster and J. Stroop, "Consistent design of embedded real-time systems with UML-RT," presented at the 4th International Symposium on Object-Oriented Real-Time Distributed Computing, Magdeburg, Germany, 2001.

[80] L. C. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of UML models,".

[81] S.kang,H kim ,j,Baik, H. Choi and shc “transformation rules between Sythesis ”

[82] L. Telinski, W. Agner, I. W. Soares, P. C. Stadzisz, and J. M. Simão, "A Brazilian survey on UML and Model-Driven practices for embedded software development,".

[83] M. Petre, "UML in practice," presented at the 35th International Conference on Software Engineering, San Francisco, CA, USA, 2013.

[84] A. M. Fernández-Sáez, M. Genero, D. Caivano, and M. R. V. Chaudron, "On the Use of UML Documentation in Software Maintenance: Results from a Survey in Industry,".

[85] P. G. Sapna and H. Mohanty, "Ensuring consistency in relational repository of UML models,".

[86] N. Ibrahim, R. Ibrahim, and M. Z. Saringat, "Definition of Consistency Rules between UML Use Case and Activity Diagram,".

[87] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for UML diagrams," Journal of Systems and Software, vol. 144, pp. 121- 142, 2018.