N° d'ordre : ……/ Faculté des Sciences/ UMBB 2014

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE M'HAMED BOUGARA – BOUMERDES

FACULTE DES SCIENCES

Mémoire de Magister

Présenté par : SAMI Sihem

Filière : Systèmes Informatiques et Génie des Logiciels

Option : Spécification Logiciels et Traitement de l'Information

Gestion de la cohérence sémantique lors de l'évolution des modèles appliquée aux ADL

Devant le Jury:

Mr.Mohamed Mezghiche	Prof. (Univ. Boumerdès)	Président
Mr. TARI Abdelmalek	(**MCA**. A/Mira de Bejaia)	Examinateur
Mr.Mohamed Ahmed Nacer	Prof. (USTHB)	Encadreur
Me.Assia Hachichi	MCB. (USTHB)	Examinatrice

Année Universitaire : 2013/2014

Abstract

Systems are increasingly complex. They are likely to evolve, in order to add new features, to modify existing features or to adapt to new technological needs. This implies a very high cost of Maintenance and development. To make the evolution of system less complex, it is necessary to Take in consideration the level of abstraction in the system specification using the Model. There are several types of models, we use two, oriented objects and oriented component. So the model is used to explain a system, the evolution of the system involves the evolution of the model. However, the question that arises is: after an evolution, is our model still consistent? For this reason, our main objective is to maintain consistency of the models regardless of their Meta Model and in particular the semantic consistency.

We presented our IMoSCM model as a solution for the problems related to the management of The evolution of models regardless of Meta Model, ensuring semantic consistency of the models. In IMoSCM the model praxis represents all software architecture (oriented component, object-oriented) likely evolved. We introduced this concept for modeling any software architecture in the form of elementary actions based on the Praxis formalism. Which involves a management of evolution regardless of Meta model of the architecture to be evolved. In IMoSCM semantic inconsistency detection rules are predefined, allowing the detection of semantic inconsistencies after evolution using the semantic properties proposed in SAEV [9]. In order to correct inconsistencies found, IMoSCM provides a suite of elementary action as a repair plan, from the list of inconsistencies found, and an automatic correction of inconsistencies. For this we have introduced the semantic properties defined at connector level, to manage the evolution of component -based software architectures.

We implemented our solution for two different Meta models, to prove that our solution is independent of any Meta model. For this, the implementation is divided into two parts: one is independent from the meta-model and the other one is dependent from it. A performance calculation is done for both ACME ADL [36] and xADL [49, 51] on a client - server architecture example from 10 to 100 architectural elements. We have presented the parameters that can have an impact on the execution time as well as the result of the consistency check.

Keywords: Model, Meta model, specification of evolution, strategy of evolution, inconsistency detection rules, inconsistency correction rules, semantic properties, repair plan, Praxis, evolution model, ADL.

ملخص

الانظمة البرمجية تزداد تعقيدا يوما بعد يوم . فمن المرجح أن تتطور أو يضاف اليها ميزات جديدة لتعديل الميزات الموجودة أو للتكيف مع الاحتياجات التكنولوجية الجديدة . وهذا ينطوي على تكلفة عالية للصيانة والتطوير. لجعل التغييرات في الأنظمة أقل تعقيدا ، فمن الضروري رفع مستوى التجريد في مواصفات النظام باستخدام النموذج . وهناك عدة أنواع من النماذج، ونحن في هذه الدراسة نستخدم نموذجين ، l'orienté composant و l'orienté objet

بحيث يتم استخدام النموذج لشرح النظام وتطور هذا الأخير ينطوي على تطور النموذج. مع ذلك ، فإن السؤال الذي يطرح نفسه هو : بعد التغيير ، هل النموذج لا يزال متسقا ؟ لهذا السبب، هدفنا الرئيسي هو الحفاظ على اتساق النماذج بغض النظر عن ميتا نموذج وعلى وجه الخصوص الاتساق الدلالي .

قدمنا نموذجنا IMoSCM كحل للمشاكل المتعلقة بإدارة تطوير النماذج بغض النظر عن méta modèle ،مع ضمان الاتساق الدلالي للنماذج . في IMoSCM، النموذج Praxis يمثل اي هندسة البرمجيات (orienté objet، orienté composant يمكنها ان تتغير . قدمنا هذا المفهوم لنمذجة أي هندسة برمجيات على شكل إجراءات اولية حسب Praxis. والذي يسمح إدارة التغيير بغض النظر عن méta modèle.

في IMoSCM قواعد الكشف عن التناقض الدلالي محددة سلفا ، للسماح للكشف عن التناقضات الدلالية بعد التطوير باستخدام الخصائص الدلالية المقترحة في SAEV [9] . من أجل تصحيح التناقضات ، يوفر IMoSCM مجموعة من الإجراءات الابتدائية على اساس PRAXIS [12] كخطة إصلاح،استنادا على قائمة التناقضات التي وجدت ، و تصحيح اوتوماتيكي للتناقضات . لهذا قدمنا خصائص الدلالية محددة على مستوى connecteur، لإدارة تطور هندسة البرمجيات à base de composant

نفذنا اقتراحنا لنموذجين مختلفين، لإثبات أن الحل مستقل عن أي méta modèle . لهذا ، ينقسم التنفيذ إلى قسمين: جزء مستقل لل méta modèle وجزء يعتمد على méta modèle.

تمت عملية تقييم الأداء لكلا من ADL [36] ACME و 49] xADL،[51 على سبيل المثال لهندسة - client - serveur من 10-100 عنصر هندسي . لقد قدمنا الاعدادات التي يمكن أن يكون لها تأثير على وقت التنفيذ وكذلك نتيجة التدقيق التناسقي .

مفتاح الكلمات :نموذج, متا مودل ,قواعد الكشف عن التناقظ الدلالي, الخصاءص الدلالية, قواعد حل التناقض, خطة اصلاح, نموذج التطوير ,ADL,PRAXIS, تحديد التطور,خطة التطور

Résumé

Ce travail s'inscrit dans le domaine de l'évolution des modèles et la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs Méta Modèle, en assurant leurs cohérences sémantiques.

Les systèmes sont amenés à évoluer soit pour ajouter de nouvelles fonctionnalités, pour modifier les fonctionnalités existantes ou bien pour s'adapter aux nouveaux besoins technologiques. Ce qui implique un cout de maintenance et de développement très élevé. Afin de rendre l'évolution des systèmes moins complexe, il est nécessaire d'élever le niveau d'abstraction dans la spécification du système en utilisant le **Modèle.** Un modèle est une description et une spécification partielle d'un système, comme exemple les modèles relationnels, qui permettent de spécifier la structure des bases de données. Le modèle sert à expliquer un système, ainsi l'évolution de ce dernier implique celle du modèle. Cependant, la question qui se pose est: après une évolution, notre modèle est-il encore cohérent?

Notre principale problématique est liée à l'évolution statique et structurelle au niveau modèle. Proposer une solution automatique afin de gérer l'impact engendré par les changements, établir le lien entre le modèle de départ et le modèle d'arrivé et assurer une cohérence sémantique indépendamment de tout méta modèle. Un modèle nommé IMoSCM (***I**ndependent Model **S**émantique **C**onsistency **M**anagement)* est proposé pour une gestion automatique de la cohérence sémantique indépendamment de tout méta modèle lors de l'évolution des modèles. Cette contribution est validée par une application développée en JAVA en utilisant ECLIPS. Une illustration est présentée au travers de deux ADL ACME et xADL.

Mots-clés : Modèle, Méta modèle, spécification de l'évolution, stratégie d'évolution, règle de détection d'incohérence, règle de solution d'incohérence, propriétés sémantiques, plan de réparation, Praxis, Modèle d'évolution, ADL.

Sommaire

Chapitre 1 :
Introduction générale

Contexte

Ce travail s'inscrit dans le domaine de l'évolution des modèles et la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs Méta Modèle, en assurant leurs cohérences sémantiques.

Les systèmes sont de plus en plus complexes et font face à de nombreuses difficultés telles que la gestion de la réutilisabilité, de l'interopérabilité, ou encore de la flexibilité. Ils sont amenés à évoluer soit pour ajouter de nouvelles fonctionnalités, pour modifier les fonctionnalités existantes ou bien pour s'adapter aux nouveaux besoins technologiques. Ce qui implique un coût de maintenance et de développement très élevé. Afin de rendre l'évolution des systèmes moins complexe, il est nécessaire d'élever le niveau d'abstraction dans la spécification du système en utilisant le **Modèle.** Un modèle est une description et une spécification partielle d'un système, comme exemple les modèles relationnels, qui permettent de spécifier la structure des bases de données. Aussi chaque modèle nécessite un formalisme de modélisation comme une légende pour la carte géographique. Un formalisme de modélisation est un langage qui permet d'exprimer des modèles. Il définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles. Ces derniers offrent de nombreux avantages, le plus important est la spécification de *différents niveaux d'abstraction* afin de présenter l'architecture générale d'une application.

Dans l'ingénierie dirigée par les modèles (MDE : Model Driven Engeneering) [13] l'objet de réflexion central du processus de développement est le modèle : l'idée c'est de se concentrer sur les concepts et les liens entre concepts, et ce, à divers niveaux d'abstraction, les aspects liés aux contingences d'une plateforme d'exécution cible n'apparaissent que tardivement dans le processus (comme le code source des composants par exemple). Cette approche prend tout son sens dans le cadre des architectures logicielles dirigées par les modèles utilisant des standards tels que le **MDA (Model-Driven Architecture) [2]** proposé par l'OMG. Elle a pour but d'apporter une nouvelle façon de concevoir des applications en séparant la logique métier de l'entreprise, de toute plate-forme technique. La démarche MDA propose à terme de définir un modèle métier indépendant de toute plate-forme technique et de générer automatiquement du code vers la plate-forme choisie. Donc L'approche MDA permet de réaliser le même modèle sur plusieurs plates-formes et permet aux applications d'inter opérer

en reliant leurs modèles et supporte l'évolution des plates-formes. La mise en œuvre du MDA est entièrement basée sur les modèles et leurs transformations.

Il existe plusieurs type de modèle, nous retenons que deux, les modèles à objets « l'OO » qui ont été créés pour modéliser le monde réel. "Dans un modèle à objets, toute entité du monde réel est un objet, et réciproquement, tout objet représente une entité du monde réel". L'intérêt principal de l'OO réside dans le fait que l'on ne décrit plus par le code des actions à réaliser de façon linéaire mais par des ensembles cohérents appelés objets. Le deuxième type de modèle est l'orienté composant qui nous intéresse pour la suite de nos travaux. Le paradigme orienté composant est apparu pour compléter les lacunes de l'orienté objet. Avec l'orienté composant la construction d'application se fait par assemblage et réutilisation d'entités existantes appelées *composant*, Aussi un formalisme est utilisé afin de décrire le déploiement et les interactions entre composants. Les travaux relatifs aux modèles de composant, appelés aussi *architectures logicielles*, sont complémentaires et tendent à se rapprocher. Trois principaux concepts existent dans le paradigme de la modélisation par composant : *composant, connecteur et configuration* et qui sont généralement acceptés comme essentiels [6].

Donc le modèle sert à expliquer un système, ainsi l'évolution de ce dernier implique celle du modèle. L'évolution des modèles et la gestion de la cohérence deviennent une activité cruciale pour faire face aux changements de tout système logiciel. Pour cette raison, des ateliers annuels internationaux ont été mise en place depuis 2007. Leurs principal objectif est d'explorer et de renforcer l'interaction et la synergie entre les domaines de recherche relatif à l'évolution des logiciels, Co -évolution, gestion de la cohérence et l'ingénierie dirigée par les modèles.

Problématique

En informatique l'évolution des logiciels survient, suite à des changements technologiques, l'apparition de nouveaux besoins fonctionnels et non fonctionnels, et évolution des modèles pour rester en adéquation avec les logiciels qu'ils modélisent. L'évolution des systèmes est une nécessité incontournable. Elle permet d'éviter que les systèmes ne restent figés et soient obsolètes par rapport aux besoins en perpétuel changement. En effet, l'évolution permet non seulement d'allonger la durée de vie des systèmes mais également de prendre en compte de nouveaux besoins ou des fonctionnalités plus complexes. Un modèle exprimant un système

doit donc pouvoir être modifié pour rester utilisable, disponible et robuste auprès de ses utilisateurs. L'évolution d'un modèle se reflète par les différents changements dans sa structure ou son comportement. Elle peut être réalisée à l'étape de spécification ou de conception du système qu'il décrit. On parle alors d'*évolution statique*. Elle peut également être réalisée à l'exécution de ce système. On parle alors d'*évolution dynamique*. Dans chacun de ces cas, il faut considérer que tout élément du modèle peut être amené à évoluer [2]. Cependant, la question qui se pose est: après une évolution, notre modèle est-il encore cohérent? Pour cette raison, notre principal objectif est de maintenir la cohérence des modèles indépendamment de leurs Méta Modèle et plus particulièrement la cohérence sémantique.

Dans la littérature nous avons constaté que tous les travaux étudiés, relatifs à la gestion de l'évolution et vérification de cohérence, traitent l'évolution statique d'une façon automatique en se basant généralement sur des contraintes et invariants. Quelques un utilisent des techniques de versionnement et des règles de transformation de modèle. La plus part s'intéressent à la cohérence structurelle. Par contre SAEV [9], un modèle d'évolution pour les architectures logiciels, aborde la cohérence sémantique mais uniquement dans le contexte des ADL [6]. Nous avons remarqué aussi que seul PRAXIS propose une solution traitant la cohérence indépendamment de tout méta modèle mais ne traite pas la cohérence sémantique. Notre proposition vient compléter ce manque d'une part en traitant la cohérence sémantique et d'une autre part proposer une solution indépendamment de tout méta modèle.

Notre principale problématique est donc liée à l'évolution statique et structurelle au niveau modèle. Proposer une solution automatique afin de gérer l'impact engendrer par les changements, établir le lien entre le modèle de départ et le modèle d'arrivé et assurer une cohérence sémantique indépendamment de tout méta modèle [3] (un ensemble de concept pour décrire un modèle. Comme les diagrammes de classes pour UML).

Contribution

D'après notre état de l'art différent types de cohérence existent tels que la cohérence syntaxique, structurelle, statique, sémantique et autres. Parmi les travaux qui ont étudiés la cohérence : PRAXIS [12] et SAEV [9]. Dans SAEV la cohérence sémantique est étudiée, mais ce travail ne porte que sur le contexte des ADL [6] (Langage de description

d'architecture). L'autre travail PRAXIS traite différents types de cohérence mise à part la cohérence sémantique et ceci indépendamment de tout méta modèle.

Dans notre travail nous allons étudier la gestion de l'évolution des modèles indépendamment de leurs Méta Modèle, en assurant une cohérence sémantique des modèles résultants. Pour cela, nous proposons un modèle nommé IMoSCM (***I**ndependent Model **S**émantique **C**onsistency **M**anagement)* pour une gestion automatique de la cohérence sémantique indépendamment de tout méta modèle lors de l'évolution des modèles.

Dans IMoSCM le modèle en praxis représente toute architecture logicielle (*orientée composant, orientée objet*) susceptible d'évoluer. Nous avons introduit ce concept pour pouvoir modéliser toutes architectures logicielles, sous forme d'actions élémentaires en se basant sur le formalisme de Praxis. Ce qui permet une gestion de la cohérence après évolution indépendamment du méta modèle de l'architecture à faire évoluer. Dans IMoSCM des règles de détection d'incohérence sémantique sont prédéfinies, afin de permettre la détection des incohérences sémantiques après évolution en utilisant les propriétés sémantiques proposées dans SAEV [9]. Dans le but de corriger les incohérences trouvées, IMoSCM propose une suite d'action élémentaire sous forme de PRAXIS, comme un plan de réparation, à partir de la liste des incohérences trouvées, ainsi qu'une correction automatique de l'incohérence.

Notre objectif est de maintenir une cohérence sémantique indépendamment de tout méta modèle, notre contribution consiste donc à intégrer la cohérence sémantique dans Praxis. Elle apporte principalement un plan de réparation automatique en se basant sur des invariants et des propriétés sémantiques afin de garder un modèle cohérent à chaque évolution. De plus nous considérons tout modèle comme un ensemble de nœuds connectés via des arcs, ce qui permet à notre solution d'être indépendante de tout méta modèle. Nous validons cette contribution par une application développée en JAVA [50] en utilisant ECLIPS [50]. Nous présentons une illustration au travers de deux ADL [6] ACME [36] et xADL [49 ,51].

Organisation du document

Ce manuscrit est organisé comme suit :

Chapitre 1 : Introduction générale

Dans ce chapitre nous présentons les notions liés au domaine de l'évolution des modèles, représentant ainsi le contexte dans lequel s'inscrit notre travail. Nous exposons aussi notre problématique liée à la gestion de l'évolution des modèles indépendamment de leurs Méta Modèle, en assurant leurs cohérences sémantiques. Nous terminons en présentant notre contribution, un modèle IMOSCM pour une gestion de la cohérence sémantique indépendamment de tout méta modèle.

Chapitre 2 : Etat de l'art

Dans ce chapitre nous présentons les différents travaux effectués sur la gestion d'évolution et vérification de la cohérence des modèles, ils sont classés selon deux types de modélisation : orientée *objet* et orientée *composant*. Nous terminons par un bilan composé de trois parties : la première partie représente deux comparaisons, une sur les différents travaux relatifs à la gestion d'évolution des modèles et une autre sur les différents travaux relatifs à la gestion de la cohérence des modèles. La deuxième partie représente un ensemble de constatations sur les limites des différents travaux étudiés. La troisième partie et la dernière présente nos objectifs à atteindre.

Chapitre 3 : Modèle d'évolution IMOSCM

Nous présentons dans ce chapitre notre modèle IMoSCM (***I****ndependent Model* ***S****émantique* ***C****onsistency* ***M****anagement)*, comme solution à la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs méta modèle, en assurant une cohérence sémantique des modèles. Nous commençons par une présentation des objectifs attendus et les différents concepts de notre modèle IMoSCM. Nous présentons ensuite la gestion d'évolution et la gestion de cohérence par le modèle IMoSCM. Nous terminons par un bilan qui positionne notre modèle par rapport aux limites des travaux étudiés dans le chapitre précédent, et les objectifs fixés dans le début de ce chapitre.

Chapitre 4 : Modélisation UML du système d'évolution

Dans ce chapitre nous présentons une conception fonctionnelle du modèle IMoSCM. Nous présentons le diagramme de classe ainsi que le diagramme de cas d'utilisation pour chaque étape du processus du modèle IMoSCM : l'application de l'évolution et la vérification de la cohérence & application du plan de réparation. Les détails d'implémentation ainsi que la validation du modèle IMoSCM sont présentés dans le chapitre suivant.

Chapitre 5 : Validation

A travers ce chapitre nous présentons une validation de nos travaux sur la vérification de cohérence sémantique. Nous donnons premièrement une présentation générale de la manière de valider. Dans la deuxième partie, nous présentons l'implémentation de la partie indépendante du méta modèle. Dans la troisième partie, nous présentons l'implémentation de la partie dépendante du méta modèle. Un exemple appliqué pour chacun des deux ADLs ACME [36] et xADL [49 ,51] est donné dans la quatrième partie. Dans la sinquième partie et la dernière nous présentons le calcul de performance fait sur un ensemble d'architecture client-serveur. Nous présentons pour cela des paramètres qui peuvent avoir un impact sur le temps d'exécution ainsi que sur le résultat de la vérification de la cohérence : taille du modèle, nature du modèle, nature d'opération d'évolution, nombre d'opération. Nous terminons par donner une analyse des performances par rapport aux paramètres d'impact cités précédemment.

Chapitre 6 : Conclusion et perspective.

Partie 1 :
Etat de l’art

Chapitre 2 :
Etat de l’art

2.1 Introduction:

Le travail présenté dans ce manuscrit s'intéresse à la gestion de l'évolution des modèles en assurant leur cohérence sémantique après évolution. Deux concepts liés à la modélisation existes: modélisation *orientée objet* où les systèmes sont uniquement constitués d'entités appelées *objets* et modélisation *orientée composant* où les systèmes se réduisent à un assemblage d'entités prédéfinies appelées *composants,* présentés dans la section 2.2. L'évolution et la cohérence des modèles sont abordées dans les sections 2.3 et 2.4 respectivement. Pour chaque type de modélisation sont présentés les travaux relatifs à la gestion d'évolution des modèles et la gestion de la cohérence des modèles, présentés dans les sections 2.5 et 2.6. Un Bilan est présenté dans la section 2.7 en donnant deux comparaisons sur les différents travaux relatifs à la gestion de l'évolution des modèles et la gestion de cohérence des modèles, suivi d'un positionnement par rapport à l'état de l'art afin de définir notre objectif sur lequel nous nous appuierons tout au long de ce manuscrit.

2 .2 Modélisation :

Un modèle est une description et une spécification partielle d'un système, c'est l'abstraction de ce qui est intéressant d'une réalité, afin de faciliter sa compréhension et la simulation de son fonctionnement [1]. Comme exemple les modèles relationnels, qui permettent de spécifier la structure des bases de données. Dans l'ingénierie dirigée par les modèles (MDE : Model Driven Engeneering) [13] l'objet de réflexion central du processus de développement est le modèle : l'idée c'est de se concentrer sur les concepts et les liens entre concepts, et ce, à divers niveaux d'abstraction, les aspects liés aux contingences d'une plateforme d'exécution cible n'apparaissent que tardivement dans le processus (comme le code source des composants par exemple). Aussi chaque modèle nécessite un formalisme de modélisation comme la légende pour la carte géographique présentée dans la figure 2.1.

2.2.1 Formalisme de modélisation :

Un formalisme de modélisation est un langage qui permet d'exprimer des modèles. Un formalisme définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles. Dans l'exemple de la figure 2.1 deux modèles de carte géographique *région sud* et *région nord* sont établis en respectant le formalisme présenté sous forme de légende.

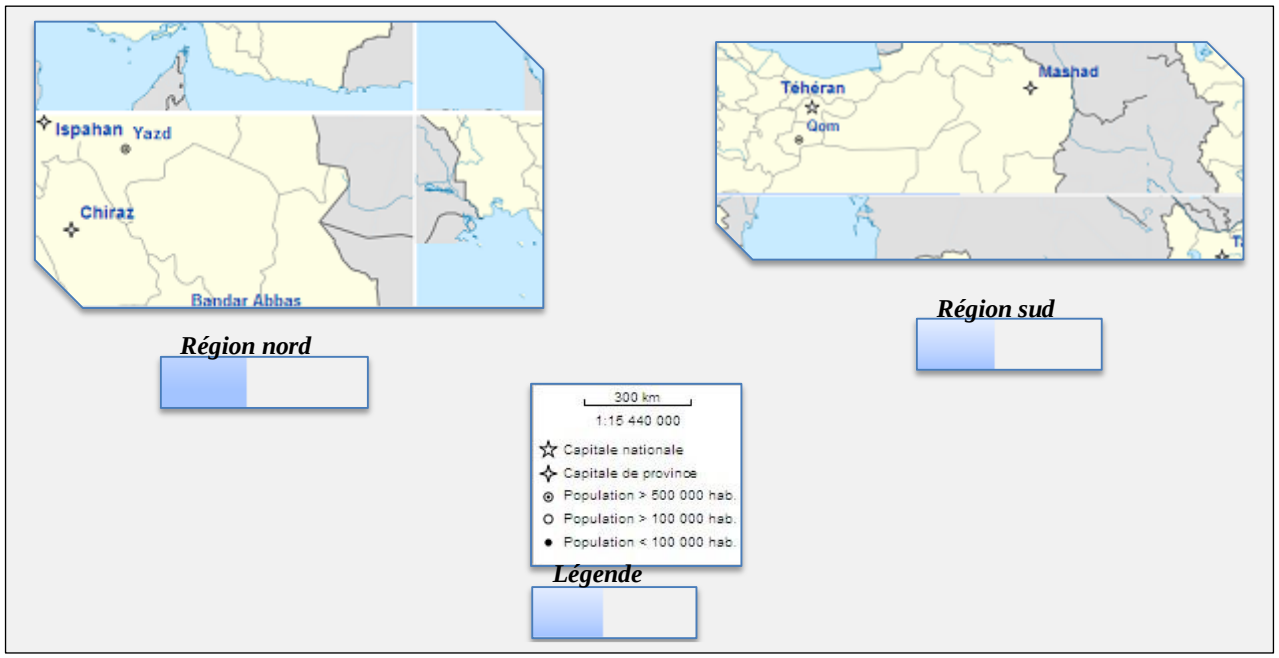


Figure 2.1: Exemple de formalisme de modélisation

2.2.2 Méta modélisation :

Un méta modèle définit la structure que doit avoir tout modèle conforme à ce méta modèle. Autrement dit tout modèle doit respecter la structure définie par son méta modèle [2]. Par exemple, le méta modèle *UML* définit que les modèles *UML* contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc. Les métas modèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. Dans l'exemple de la figure 2.2 la grammaire *JAVA* [50] définit que les programmes écrits en *JAVA* doivent respecter qu'une instruction de condition *IF* doit commencer par le mot clé *if* et une expression, suivi d'une instruction ensuite du mot clé *else* suivi d'une autre instruction.

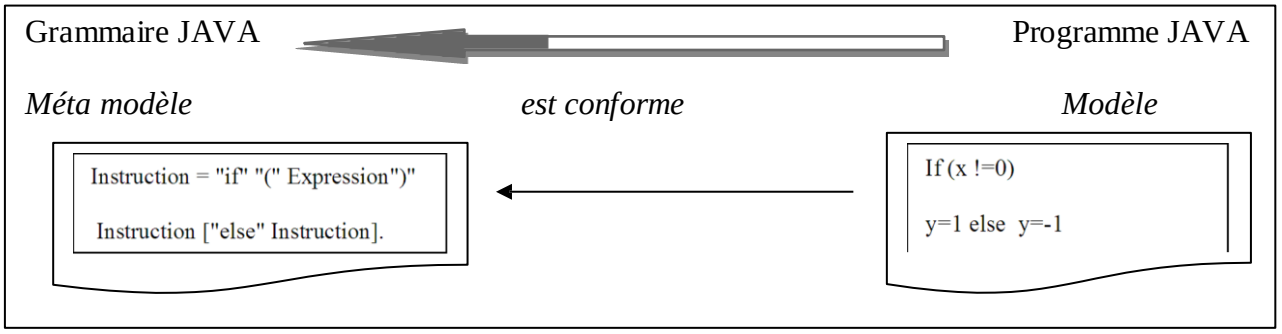


Figure 2.2: Exemple du méta modélisation

Les notions de modèles et de formalisme de modélisation ne sont pas suffisantes. Il est indispensable de travailler non pas uniquement au niveau des modèles, mais aussi au niveau des formalismes de modélisation. Il faut exprimer des liens entre les concepts des différents formalismes. Par exemple, il faut pouvoir exprimer que le concept de classe *UML* doit être transformé dans le concept de classe *JAVA*. L'objectif est de disposer d'un formalisme permettant l'expression de modèles de formalismes de modélisation. Un tel formalisme est appelé un *méta formalisme*, et les modèles qu'il permet d'exprimer sont appelés des *métas modèles*.

2 .2.3 Type de modélisation :

Il existe plusieurs type de modèle, nous retenons que deux, l'orienté *objet* qui est très utilisé et l'orienté *composant* qui nous intéresse pour la suite de nos travaux.

2.2.3.1 Modélisation orientée objet :

Les modèles à objets ont été créés pour modéliser le monde réel. "Dans un modèle à objets, toute entité du monde réel est un objet, et réciproquement, tout objet représente une entité du monde réel". L'intérêt principal de l'OO réside dans le fait que l'on ne décrit plus par le code des actions à réaliser de façon linéaire mais par des ensembles cohérents appelés objets.

Dans l'approche orientée objet, les systèmes sont uniquement constitués d'entités appelées *objets*. Un objet est « *une abstraction d'une chose du problème reflétant la capacité du système à garder de l'information sur cette chose et/ou à interagir avec elle* ». Ces objets sont définis par des *types* qui définissent de façon syntaxique et sémantique les propriétés que présenteront les objets du type ; ces propriétés permettent de délimiter, de qualifier les objets et de savoir comment on peut communiquer avec eux.

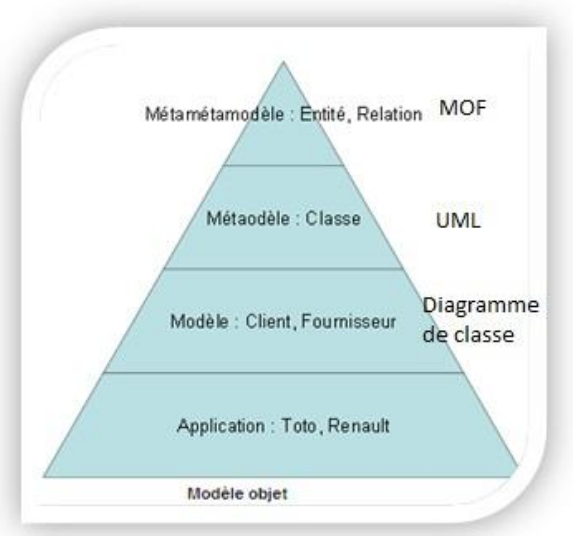


Figure 2.3: Niveau d'abstraction du modèle objet

Dans MDA (Model-Driven Architecture) [2], il n'existe qu'un seul méta formalisme, le MOF (Meta Object Facility). Aussi appelé *méta méta modèle,* le MOF permet d'exprimer des formalismes de modélisation, ou méta modèles, permettant eux-mêmes d'exprimer des modèles. Un méta méta modèle est un modèle qui définit le langage pour exprimer un méta modèle. La relation entre un méta méta modèle et un méta modèle est analogue à la relation entre un méta modèle et un modèle [3,4]. Le méta méta modèle s'auto définit et est à lui-même son propre méta modèle. De ce fait, il n'existe que les niveaux modèle, méta modèle et méta méta modèle, aussi appelé modèle MOF. [2] illustré par la figure 2.3.

2.2.3.2 Modélisation orientée composant :

Le paradigme orienté composant est apparu pour compléter les lacunes de l'orienté objet. Avec l'orienté composant la construction d'application se fait par assemblage et réutilisation d'entités existantes appelées *composant,* Aussi un formalisme est utilisé afin de décrire le déploiement et les interactions entre composants. Les travaux relatifs aux modèles de composant, appelés aussi *architectures logicielles*, sont complémentaires et tendent à se rapprocher. Trois principaux concepts existent dans le paradigme de la modélisation par composant : *composant, connecteur et configuration* et qui sont généralement acceptés comme essentiels [6].

Un *composant* est une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. Le composant est un lieu de calcul et possède un état. Il peut être simple ou composé; on parle alors dans ce dernier cas de composite. Sa taille peut aller de la fonction mathématique à une application complète. Le *connecteur* correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions. Les connecteurs sont donc des éléments essentiels pour la maintenance et la réutilisation des composants. Composants et connecteurs peuvent être assemblés à partir de leurs interfaces pour former une *configuration*. Cette dernière décrit l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que leurs interactions. Elle permet de gérer la complexité de l'application en raffinant peu à peu l'architecture en un ensemble hiérarchique de modules logiciels.

Plusieurs langages de description d'architecture à base de composant (dits ADL [6] : Architecture Description Languages) sont proposés pour aider à la spécification de ces concepts de base.

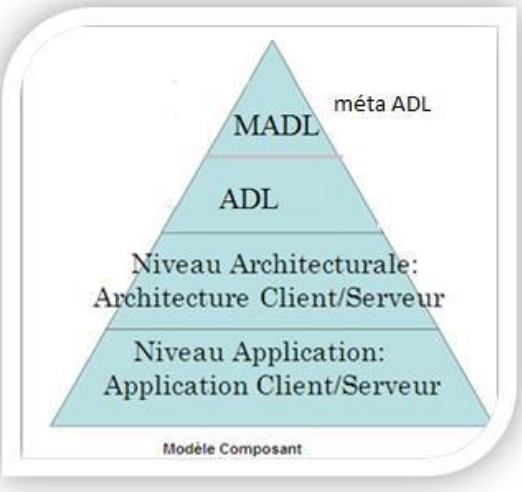


Figure 2.4: Niveau d'abstraction du modèle composant

Contrairement à l'orienté objet, plusieurs métas ADL existent et aucune unification n'existe. De manière générale [47], [48] la description d'une architecture logicielle peut être établie au travers de trois niveaux d'abstraction: le **niveau Méta**, le **niveau Architectural** et le **niveau Application** illustrés par la figure 2.5 :

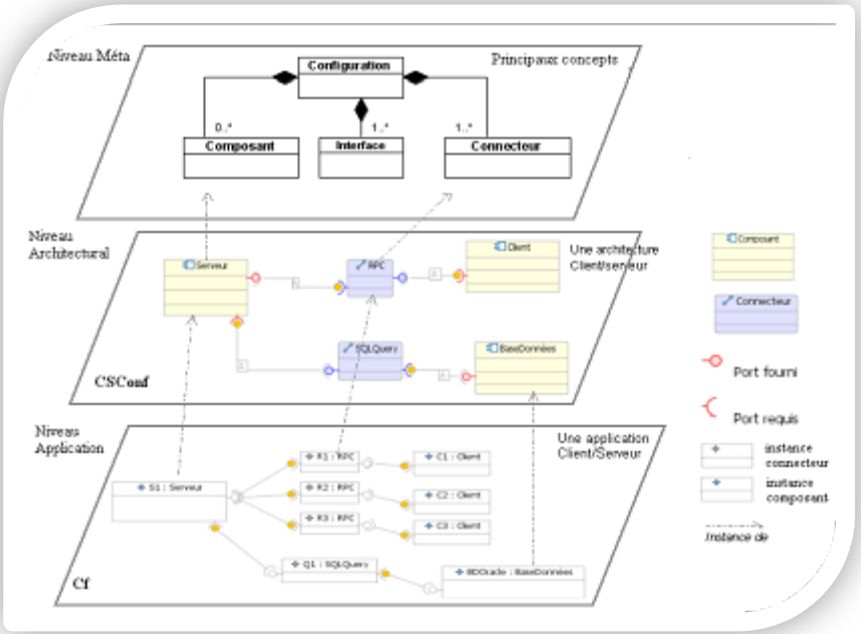


Figure 2.5: Niveaux d'abstraction de description d'architectures logicielles [9]

Le niveau méta représente le niveau de définition de tous les concepts architecturaux qu'un ADL peut proposer pour la description d'une architecture logicielle. Chaque ADL peut être décrit au niveau méta au travers des concepts qu'il propose et des différents liens entre ces concepts.

Le niveau architectural représente le niveau de description d'architectures logicielles en utilisant un ADL donné, lui-même défini auparavant au niveau méta. A ce niveau plusieurs types de composants, de connecteurs, de configurations peuvent être définis. Ces types doivent être conformes aux concepts de l'ADL définis au niveau méta.

Le niveau application représente le niveau de description des applications construites conformément à leurs architectures décrites au niveau *Architectural*. Chaque élément architectural du niveau Application est une instance d'un élément type du niveau Architectural. Par exemple à partir de l'architecture type *Client serveur* de la figure 2.5, on peut construire l'application suivante composée de la configuration *Cf* : Instance de *CSConf* ; de trois composants *C1, C2, C3* : instances du composant Client ; du composant *BDoracle* : instance du composant Base données ; *S1* : instance du composant serveur ; de trois connecteurs *R1,R2,R3* : instances du connecteur *RPC ; Q1* : instance du composant *SQLQuery*.

2.3 Evolution des modèles :

Un modèle doit évoluer à travers des modifications et des transformations, afin de rester utilisable et de répondre aux nouveaux besoins en perpétuelle changement. Des définitions sur l'évolution, coévolution et transformation sont présentées dans les sous sections suivantes afin de se positionner par rapport au terme évolution et transformation utilisés dans ce manuscrit.

2.3.1 Définition évolution :

Dans le *dictionnaire français* [14] le terme évolution désigne une transformation progressive, une suite de transformations graduelles ou une suite de petits changements successifs. Par exemple, dans le domaine de la médecine, ce terme indique les différents stades par lesquels passe une maladie. En biologie c'est une transformation des espèces vivantes qui se manifeste par des changements de leurs caractères génétique pouvant aboutir à de nouvelles espèces. En informatique on parle d'évolution des logiciels, suite à des changements technologiques, l'apparition de nouveaux besoins fonctionnels et non fonctionnels, et évolution des modèles pour rester en adéquation avec les logiciels qu'ils modélisent.

Dans les travaux de [7] trois écoles de pensée sont distinguées concernant le terme évolution. Dans la *première* le terme évolution désignait l'étude des changements, au fil du temps, des propriétés d'un logiciel (Taille du logiciel, efforts, nombre des changements, coûts, etc.). Dans la *seconde école*, ce terme a été repris comme un sous ensemble des activités de maintenance : par exemple l'ajout, la suppression ou la modification de propriétés fonctionnelles. *La dernière école* celle adoptée par les travaux [7] considère que l'évolution est un terme plus général et plus approprié pour d´écrire la problématique de la vie d'un logiciel après sa mise en service. Ce terme doit donc se substituer à celui de maintenance.

Pour l'évolution des systèmes, les travaux dans [8] distinguent deux types d'évolution. Les *évolutions internes* à l'application (création ou destruction d'instances, création ou destruction de liaisons) appelée aussi *évolution structurelle*, et les *évolutions externes* à l'application qui permettent d'ajouter de l'information sur une préoccupation de l'application (déploiement, sécurité …etc.) au niveau de la spécification du schéma d'instance appelée aussi *évolution comportementale*. Dans les travaux [9] évoque que dans la littérature, certains auteurs utilisent le terme *maintenance,* d'autres parlent d'*adaptation* pour désigner l'évolution logicielle. Elle peut être réalisée à l'étape de spécification ou de conception du système «*évolution statique »*. Elle peut être réalisée à l'exécution de ce système « *évolution dynamique »*

Dans ce document le terme *évolution* est considéré comme un terme plus général qui désigne l'ensemble des changements d'un modèle tout au long de son cycle de vie et les deux autres activités de maintenance et d'adaptation comme étant des cas particuliers de l'évolution.

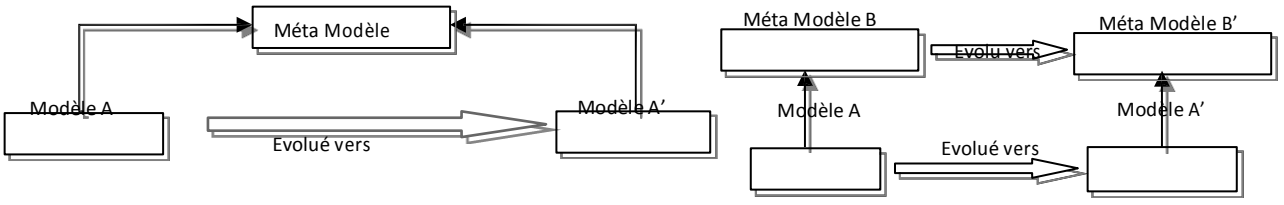
Un modèle doit toujours être conforme à son méta modèle. Par conséquent un modèle A qui évolue vers un modèle A' doit être conforme au même méta modèle comme montrer dans la figure 2.6 (a).

2.3.2 Définition coévolution :

Dans le *dictionnaire français* [14], le terme coévolution désigne une évolution simultanée et/ou interdépendante de deux entités de même espèce. Par exemple, les plantes à fleurs et les insectes qui en assurent la pollinisation.

Dans [10] définit la coévolution, dans son sens le plus large, comme une adaptation évolutive qui se produit chez plusieurs éléments (gènes ou espèces) à la suite de leurs influences réciproques.

Dans la modélisation, en prenant l'exemple de la figure -6- (b), si le méta modèle évolue le modèle A' doit aussi évoluer à fin de rester conforme au nouveau méta modèle B'. Donc le modèle A, dans ce cas, doit évoluer afin d'atteindre le modèle A' de plus doit aussi évoluer pour être conforme au nouveau méta modèle B'. Ce qui est appelé une *coévolution*.



Evolution (a) Coévolution(b)

Figure 2.6: Evolution & Coévolution

2.3.3 Définition transformation :

Dans le dictionnaire le mot transformation désigne un changement d'une forme vers une autre. Dans les transformations de modèle ce terme est une opération qui prend un modèle source en entrée, fournit un modèle cible en sortie au moyen d'une spécification de transformation. A son tour, une spécification de transformation englobe l'ensemble des règles nécessaires à la transformation. Enfin, chaque règle décrit comment transformer une instance source vers une instance cible correspondante.

D'après [11] une taxonomie des transformations est donnée :

Transformation endogène :
Une transformation est dite endogène si les modèles d'entrée et de sortie sont conformes au même méta modèle et sont dans le même espace technologique.

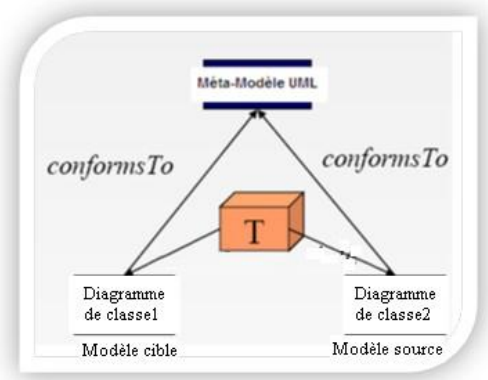


Figure 2.7: Exemple de la transformation T d'un modèle *UML* en un autre modèle *UML*

Dans l'exemple de la figure 2.7 le *diagramme de classe1* conforme au méta modèle *UML* est transformé via la transformation *endogène* T vers le *diagramme de classe2* qui reste conforme au même méta modèle *UML*.

Le but de cette transformation pourrait être, l'amélioration des performances d'un système tout en maintenant la sémantique, par exemple le changement dans la structure pour améliorer certains aspects de la qualité du logiciel tels que la compréhension, la maintenance, la modularité et la réutilisation sans en changer le comportement observable ou tout simplement la réduction de la complexité syntaxique.

Transformation exogène :

Lorsque les modèles sources et cibles sont conformes à des métas modèles différents et sont entre deux espaces technologiques différents.

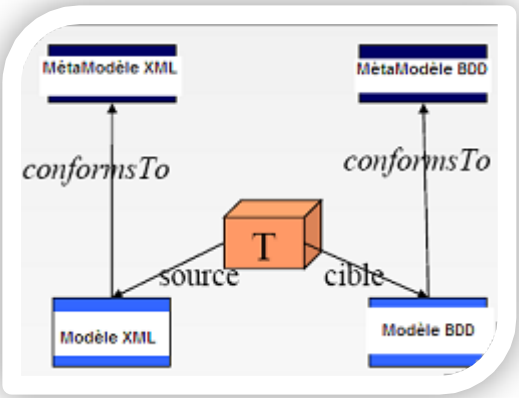


Figure 2.8: Exemple de la transformation T d'un fichier *XML* en schéma de *BDD*

Dans l'exemple de la figure 2.8 le *modèle XML* conforme au méta modèle *XML* est transformé via la transformation *exogène* T vers un *modèle BDD* qui est conforme à un autre méta modèle *BDD*. Le but de cette transformation pourrait être, le passage d'un niveau d'abstraction vers un autre niveau d'abstraction moins/plus élevé « ***Transformation horizontale*** », par exemple la génération du code, « *dans [2] des stéréotypes, tagged-values et contraintes sont présentés qui permettent de construire des modèles UML modélisant des applications EJB afin de générer du code à partir de tels modèles* ». Ou bien dans le même niveau d'abstraction « ***Transformation vertical*** », par exemple la transformation d'un programme écrit dans un langage vers un autre langage (du langage *C#* vers *Java* par exemple).

2.3.4 Transformation & évolution :

La transformation peut être faite de deux manières : transformer un modèle sans le faire évoluer (transformation exogène), transformer un modèle à fin de le faire évoluer (transformation endogène).

Par définition une transformation désigne une opération qui prend un modèle source en entrée, fournit un modèle cible en sortie au moyen d'une spécification de transformation [44].

Dans ce document nous considérons l'évolution comme une transformation endogène appliquée sur un modèle source à fin de fournir un modèle cible évolué tout en restant conforme au même méta modèle.

2.4 Cohérence des modèles :

Après une évolution la question qui se pose est: le modèle évolué reste-t-il conforme au même méta modèle ? Respecte-t-il certaine contrainte …etc. On parle alors de cohérence.

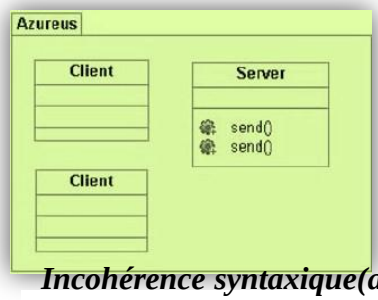
2.4.1 Définition cohérence :

Dans le dictionnaire le terme cohérence désigne un rapport d'harmonie ou d'organisation logique entre des éléments. En informatique, la **cohérence** est la capacité pour un système à pouvoir assurer une absence de contradiction entre ses éléments le constituant après un changement appliqué sur ces derniers.

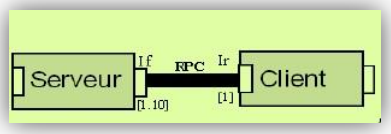
Dans l'évolution des modèles, afin d'éviter d'atteindre un modèle incohérent après évolution, la cohérence est maintenue en étudiant l'impact d'évolution d'un élément du modèle sur les autres éléments. Différents type de cohérence existent dans l'évolution des modèles [12,25] :

2.4.1.1 Cohérence syntaxique / sémantique :

Le système est basé sur des règles syntaxique et sémantique. La vérification des règles syntaxiques est appelé cohérence syntaxique, la vérification des règles sémantiques est appelé cohérence sémantique.



Incohérence syntaxique(a)



Règle sémantique (b)

Figure 2.9: Cohérence syntaxique / sémantique

La règle sémantique ci-dessus 2.9 (b) indique qu'il est possible d'attacher au maximum 10 instances du composant *Client* à une même instance du composant *Serveur*, par contre une instance de composant *Client* est reliée uniquement à une seule instance du composant *Serveur*.
La figure 2.9 (a) présente un exemple d'incohérence syntaxique provoquée par la présence de deux opérations portant le même nom *send*, et deux classes portant le même nom *Client*.

2.4.1.2 Cohérence structurelle / méthodologique:

Une nouvelle catégorie de cohérence est définie dans [12], qui est la cohérence méthodologique. Ce type de cohérence concerne le suivi d'un processus de modélisation. Il ne cible pas l'exactitude des modèles eux-mêmes, mais plutôt vérifier qu'ils ont été construit d'une manière correcte. Elle s'oppose à la cohérence structurelle, qui elle vise l'exactitude d'un état particulier d'un modèle suite aux différents changements dans sa structure et/ou dans celle de ses éléments constitutifs.
Exemple d'une *règle méthodologique* :
Si deux classes sont reliées par une relation d'agrégation, une règle méthodologique pourrait énoncer que ces deux classes appartiennent au même paquetage, ce qui permet d'augmenter le degré d'encapsulation [47].

2.4.1.3 Cohérence inter-modèle / intra-modèle :

La cohérence inter-modèle cible les incohérences qui peuvent être détectées par la vérification de chacun des modèles séparément qui peut être de la vérification de la cohérence sémantique, dynamique, structurelle…etc. Contrairement à la cohérence entre les modèles intra-modèle qui s'attaque aux incohérences découlant des relations entre les différents modèles, où les modèles peuvent être exprimés en utilisant différents méta modèles.
Comme exemple de cohérence *intra-modèle*, un modèle *UML* constitué de plusieurs types de diagrammes, un diagramme statique décrivant la structure du système, et un ou plusieurs diagrammes exprimant sa dynamique. Ces diagrammes n'étant pas indépendants, la modification du diagramme de classe nécessite d'examiner la propagation des modifications engendrées afin de maintenir la cohérence globale du système en cours de développement [46].

2.5 Gestion d'évolution des modèles :

Généralement, un système est appelé à évoluer pour plusieurs raisons, la prise en compte des nouveaux besoins par exemple mais aussi pour s'adapter à des nouveaux usages ou de nombreux autres raisons. Cette évolution entraine forcément l'adaptation des modèles représentant le système. Dans ce travail la gestion d'évolution des modèles est divisée en deux grands groupes : gestion d'évolution des modèles orientés objet et gestion d'évolution des modèles orientés composant.

2.5.1 Gestion d'évolution des modèles orientés objet :

2.5.1.1 Coévolution et gestion de version dans le contexte MDE :

Dans ce travail [44] la gestion des versions est étudiée dans l'environnement *MDE* [13] en français *IDM* (**I**ngénierie **D**irigée par les **M**odèles). L'objectif est de traiter la problématique qui se pose lorsque des métas modèles, des modèles ou des transformations sont manipulés où évolués. Le but est de mettre en évidence la nécessité d'une représentation de différence entre les modèles suite à un changement pour proposer un support des changements dans le contexte de *MDE* [13]. Le but dans ce travail est de gérer l'évolution par la transformation et le versionnement en trouvant une représentation convenable des différences entre modèles.

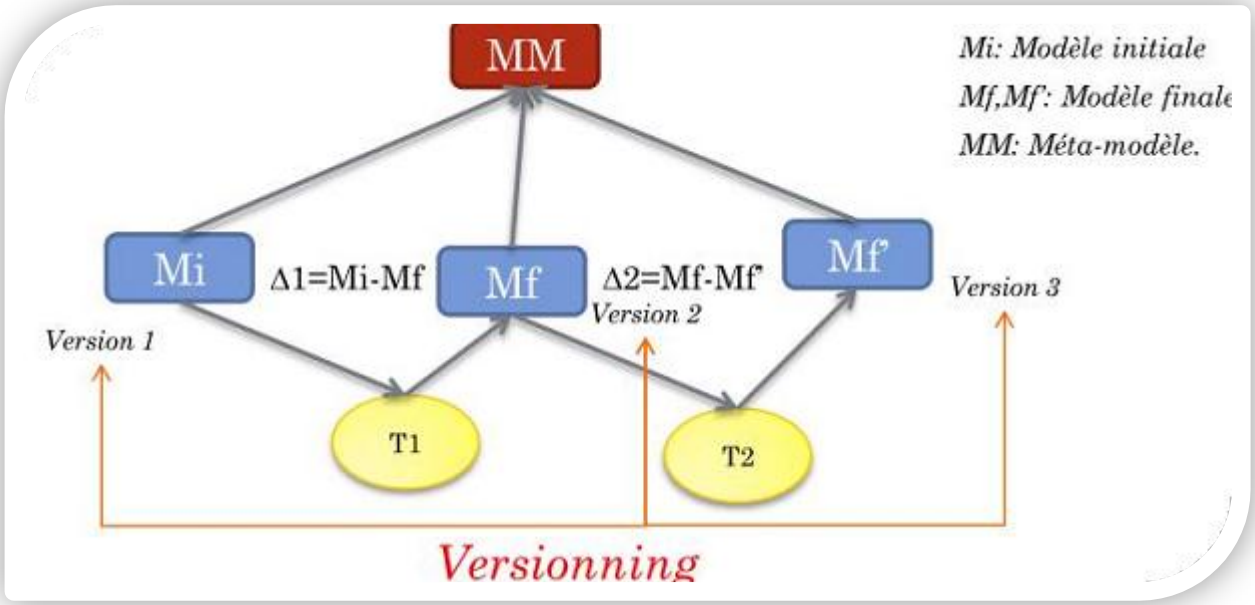


Figure 2.10: représentation de différence.

La technique de représentation proposée dans ce travail permet de valider la vision de *MDE* « tout est un modèle » car la différence entre deux modèles ***(Mi, Mf)*** est représenter par un modèle appelé *delta modèle* $\Delta =$ ***Mf – Mi*** comme montré dans la figure 2.10. Ce modèle permet :

- D'enregistrer uniquement les changements survenus,

- D'assurer que toutes données nécessaires sont disponibles pour d'une part pouvoir obtenir le modèle finale ***Mf*** à partir de ***Mi*** en appliquant une transformation automatique $\mathbf{T_{\Delta}(Mi)=Mf.}$ Et d'autre part annuler le changement en revenant vers le modèle initiale ***Mi*** par application de l'opération d'annulation $\mathbf{T^{-1}(Mf) = Mi.}$

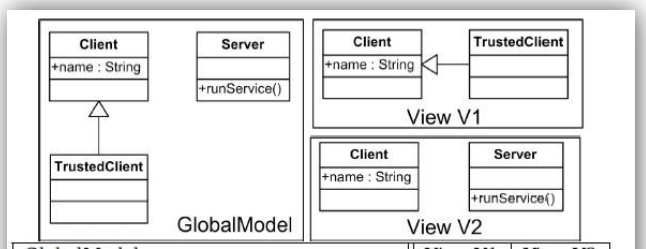
- Garantir la possibilité de construire un seul modèle qui combine les différentes manipulations : *Composition séquentielle* correspond à la fusion des deux deltas modèles. $\Delta = \Delta_1 ; \Delta_2$ où ';' est un opérateur de composition séquentiel. Le résultat est un modèle de différence contenant uniquement les modifications qui n'ont pas été remplacées par les modifications ultérieures. La *Composition parallèle* correspond aux modifications qui sont parallèles et indépendantes, le delta est obtenu par la fusion des deux delta modèles $\Delta = \Delta_1 \| \Delta_2$. Dans le cas où les modifications ne sont pas indépendantes, une intervention manuelle est proposée pour la résolution des conflits.

2.5.1.2 Praxis :

Dans ce travail [12] les modèles sont représentés par des séquences de six actions élémentaires (*Create, Delete, AddProprety, RemoveProprety, AddReference, RemoveReference*) nécessaires à la construction de chaque élément du modèle, voir l'exemple de la figure 2.11. Chaque action est marquée avec un temps qui indique le moment où elle a été exécutée par l'utilisateur. Ce formalisme appelé Formalisme de Praxis rend la solution proposée dans ce travail indépendante de tout méta modèle.

Figure 2.11: Exemple de Formalisme de Praxis

Dans ce travail une vue d'un modèle est présentée comme un sous-modèle du modèle dont elle est la vue. Le sous modèle est appelé *vue* et le modèle est appelé le *modèle global* voir la figure 2.12. Le modèle global et les vues sont représentés par des séquences d'action en formalisme de Praxis, nommés respectivement la séquence globale et les séquences de site.



GlobalModel	View V1	View V2
create(c1,class,T1)	X	X
addProperty(c1,name, 'Client',T2)	X	X
create(a1,attribute,T3)	X	X
addProperty(p1,name,'name',T4)	X	X
addProperty(p1,type, 'String',T5)	X	X
addReference(c1,attribute,a1,T6)	X	X
create(c2,class,T7)	X	
addProperty(c2,name,'TrustedClient',T8)	X	
addReference(c2,super,c1,T9)	X	
create(c3,class,T10)		X
addProperty(c3,name,'Server',T11)		X
create(o1,operation,T12)		X
addProperty(o1,name,'runService',T13)		X
addReference(c3,operation,o1,T14)		X

Figure 2.12: Représentation du modèle globale et les vues

Les vues sont représentées par des séquences de site, les modifications apportées sur une vue sont des ajouts de nouvelles actions dans les séquences de site correspondantes. Quand un changement concerne un élément de modèle, il est seulement propagé aux sites qui le partagent. La liste suivante définit le protocole de propagation des changements :

Créer (me, mc): Quand un nouvel élément de modèle est ajouté à une vue, l'élément de modèle est automatiquement considéré comme faisant partie du modèle globale parce que les vues sont incluses dans le modèle global. En outre, cette création n'a aucun impact sur les autres vues.

Delete (me): Quand un élément de modèle 'me' est supprimé d'une vue, l'élément de modèle est automatiquement considéré comme supprimé du modèle global et des vues qui le partagent.

AddProperty (me, p, v): Quand une nouvelle valeur 'v' de la propriété 'p' est ajoutée pour un élément de modèle 'me' dans une vue, l'élément global est automatiquement modifié ainsi que les vues qui partagent cette élément.

RemProperty (me, p): Quand une propriété 'p' est supprimée pour un élément de modèle 'me' dans une vue, l'élément global est automatiquement modifié ainsi que les vues qui partagent cette élément.

AddReference (me, r, met): Quand une nouvelle valeur 'met' de référence 'r' est ajoutée pour un élément de modèle 'me' dans une vue, l'élément global est automatiquement modifié ainsi que les vues qui partagent cette élément.

RemReference (me, r): Quand une référence 'r' est supprimée pour un élément de modèle 'me' dans une vue, l'élément global est automatiquement modifié ainsi que les vues qui partagent cette élément.

Le résultat de deux modifications peut donner lieu à des conflits. Dans ce travail, quand deux actions sont en conflit, l'action (marquée avec un temps qui indique le moment où elle a été exécutée par l'utilisateur) la plus ancienne est conservée et l'action ressente est annulée sauf s'il s'agit d'une action de suppression pour laquelle les conséquences sont trop importantes. La manière dont les conflits sont identifiés et résolus est présenté comme suit:

Une valeur de propriété différente: Ce conflit est détecté si une nouvelle valeur de propriété est ajoutée pour un élément de modèle, qui possède déjà une autre valeur de la même propriété, et la propriété est *single-valued*. Si l'action reçue est plus ancienne que l'action présente dans le modèle, l'action reçue est ajoutée à la séquence de site, sinon elle est ignorée.

Ajouter et supprimer une valeur d'une propriété: Ce conflit est détecté si une nouvelle valeur de propriété est ajoutée pour un élément où la valeur de la propriété a été déjà enlevée. Ce conflit est résolu comme la précédente.

Différentes valeurs de référence: Ce conflit est détecté si une nouvelle valeur de référence est ajoutée pour l'un des éléments de modèle, qui possède déjà une autre affectation de référence. Ce conflit est résolu, comme la précédente.

Ajouter et supprimer des valeurs de référence: Ce conflit est détecté si l'action AddReference (respectivement remReference) est ajoutée pour un élément de modèle, où

l'affectation de cette référence a déjà été supprimé (respectivement AddReference). Ce conflit est résolu, comme le précédent.

Supprimer un élément de modèle et ajouter une valeur de référence de l'élément supprimé: Ce conflit est détecté suite à une action de suppression d'un élément de modèle qui est référencé localement (définition 1 : empêcher un élément de modèle à être supprimés s'il est référencé). Ce conflit est résolu comme suit : Les actions qui suppriment toutes les références existantes pour l'élément à supprimer sont ajoutées à la séquence du site (et ne seront pas propagées) avant l'action de suppression de l'élément lui-même.

Supprimer un élément de modèle et ajouter / supprimer une valeur de référence pour l'élément supprimé: Ce conflit est détecté suite à une action de suppression d'un élément de modèle, et la présence d'actions d'ajout / suppression des références à partir de cet élément de modèle. Ce conflit est résolu en ignorant les changements de référence qui se produisent après l'action de suppression.

Supprimer un élément de modèle et ajouter / supprimer une valeur de propriété de l'élément supprimé: Ce conflit est détecté suite à une action de suppression d'un élément de modèle, et la présence d'actions qui ajoutent / suppriment des valeurs de propriétés pour cet élément de modèle. Si ces actions sont plus anciennes que l'action de suppression, alors ils sont ignorés.

2.5.1.3 BDD :

Trois approches de gestion d'évolution existent selon le domaine d'application et la nature des informations utilisées [15] :

- *Réorganisation des données [16]* où les modifications sont directement et automatiquement appliquées sur le schéma de base de données avec un risque de perte d'information. Dans cette approche aucun versionnement de schéma et de suivi de l'évolution n'est possible.

-*Versions historiques de schémas [17]* où des versions d'un schéma sont faites en conservant les anciennes versions sous forme d'historiques accessibles qu'en consultation. Dans cette approche le suivi de l'évolution de schéma par versions historiques se fait par cloisonnement des données associées à un schéma et autant de copies de la base que de versions de schémas doivent alors être gérées.

-*Versions parallèles de schémas [18, 19, 20] :* trois types de versionnement peuvent être utilisés. Version de schéma où une nouvelle version d'un schéma est générée lors de toute

modification de ce schéma. Version de type où toute modification d'un type génère une nouvelle version de ce type. Version de vues où l'entité de modification est un schéma partiel qui définit une vue du schéma global. A partir d'une vue, il est possible d'en faire dériver plusieurs autres par modification du schéma de la base. Dans cette approche différentes versions d'un schéma coexistent, évoluent en parallèle et opèrent sur un même ensemble de données. La cohérence structurelle d'un schéma est assurée par des invariants avant et après toutes modification. Alors que la cohérence structurelle des objets est assurée par la réorganisation qui est basée sur le vérsionnement des types permettant aux anciennes applications d'utiliser l'ancien schéma.

-Dans [15] l'existence simultanée de plusieurs *points de vue* d'un objet est présentée :

- *des vues d'instances objets :* permettre des interprétations différentes des objets à partir des versions de leur type.
- *des vues de la base d'objets :* a tout schéma correspond une vue de la base où l'ensemble des instances des types définis dans un schéma au niveau **méta base** constitue la vue associée à ce schéma au niveau **Base**.
- *des vues de la méta base :* les vues du méta base sont les schémas de la base d'objets à partir de la **méta schéma**.

-Dans l'approche DB-MAIN [20] propose un outil d'aide à l'évolution, cet outil permet de générer automatiquement les programmes de conversion de la base de données après une modification appliquée aux différents niveaux de conception. Cette solution consiste à la propagation automatique de ces modifications entres les différentes couches conceptuelles. Ainsi que sur les données et les traitements. En analysant l'historique d'évolution permet de repérer les modifications opérées et l'analyse différentielle des conceptions logique et physique fournit les informations nécessaires pour en dériver les structures physiques à supprimer, modifier ou ajouter.

2.5.2 Gestion d'évolution des modèles orientés composant :

2.5.2.1 SAEV :

Le modèle SAEV [9] prend en entrée une architecture logicielle, sur laquelle il répercute les différents changements structurelle exprimés par le concepteur et leurs impacts induits. Il offre ainsi des concepts définis au sein de son méta modèle permettant à la fois la

spécification et la gestion de l'évolution d'une architecture logicielle ainsi qu'un processus opératoire décrivant les étapes à suivre pour mener une évolution.

Les concepts de base du modèle SAEV :

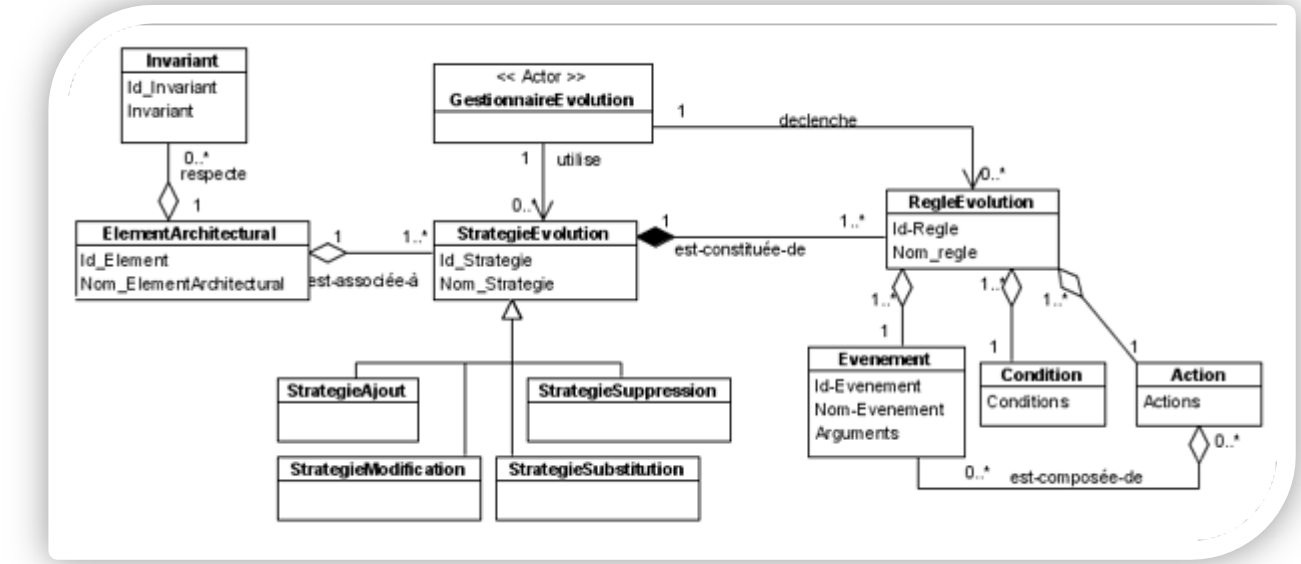


Figure 2.13: Méta modèle de SAEV

- Elément Architecturale : représente tout élément significatif dans une architecture logicielle (*Configuration, un Composant, un Connecteur, une Interface)* et à chaque niveau d'abstraction.

- Invariant : contrainte structurelle sur un élément architectural qui doit être respecté tout au long du cycle de vie de l'élément architectural. Comme exemple : le nombre de *Clients* reliés à un *Serveur* ne doit pas dépasser 30.

- Stratégie d'évolution : regroupe toutes les règles permettant de spécifier l'évolution d'un élément architectural. Stratégie d'ajout, de suppression, de modification, de substitution.

- Règles d'évolution : décrit l'exécution des opérations d'évolutions (*ajout/suppression/modification/ substitution*) sur un élément architectural modélisées par le formalisme ECA (*évènement/ Condition/ Action*. Ce formalisme est détaillé dans la section 2.6.2.1.

- Gestionnaire d'évolution : il intercepte tout évènement d'évolution, identifie la stratégie d'évolution correspondante, déclenche la ou les règles d'évolution associées à l'évènement reçu, propage les impacts des règles d'évolution exécutées et à la fin vérifie le maintien des invariants.

Processus de SAEV [9] est composé de deux phases : la spécification de l'évolution et l'exécution d'une évolution. Dans la première, SAEV offre les concepts nécessaires pour spécifier une évolution ainsi que ses impacts. Spécifier une évolution dans SAEV revient à spécifier les éléments de l'architecture à faire évoluer, selon le niveau d'abstraction considéré. Pour chaque élément architectural, il faut spécifier : ses invariants, ses stratégies d'évolution, ses règles d'évolution associées à chacune des stratégies d'évolution. Dans la deuxième phase le mécanisme opératoire du modèle SAEV pour exécuter une évolution est composé de cinq étapes :

1. Interception des événements : Le gestionnaire d'évolution intercepte deux types d'évènements : les évènements provenant du concepteur et les évènements provenant des règles d'évolution :

   • Les évènements provenant du concepteur : pour réaliser une évolution, le concepteur sélectionne l'élément architectural qu'il souhaite faire évoluer ainsi que l'opération à exécuter sur cet élément architectural. Le message représentant ainsi les choix du concepteur est intercepté par le gestionnaire d'évolution comme étant l'évènement de l'évolution.

   • Les évènements provenant des règles d'évolution : l'exécution d'une règle d'évolution peut invoquer d'autres évènements pour propager l'impact qu'elle provoque. Ces évènements sont également interceptés par le gestionnaire d'évolution.

2. Recherche de la stratégie d'évolution : A l'interception d'un évènement d'évolution, le gestionnaire d'évolution recherche la stratégie d'évolution correspondante à l'évènement intercepté. La stratégie d'évolution est identifiée via les champs "Elément Architectural" et "Opération Evolution" de l'évènement reçu.
3. Recherche de la règle d'évolution à exécuter : Une fois la stratégie d'évolution identifiée, le gestionnaire d'évolution recherche la ou les règles d'évolution à exécuter. Toutes les règles, dont la partie évènement, correspondent à l'évènement reçu, et dont la partie Condition est évaluée à vrai, sont sélectionnées.

4. Exécution des règles d'évolution et propagation des impacts : Le déclenchement d'une règle revient à l'exécution de sa partie action, séquentiellement action par action. Si une action correspond à un évènement de propagation, ce dernier sera alors intercepté, et de la même façon une nouvelle règle sera exécutée. Le processus se réitère pour chaque règle exécutée. Chaque action d'une règle est exécutée ainsi que ses propagations, avant de passer à l'action suivante.

2.5.2.2 TranSat :

TranSAT[22] propose la définition d'un nouveau mécanisme d'assemblage au sein d'une architecture logicielle, le tissage de plan. Il permet de structurer au sein d'un plan les informations relatives à une préoccupation. Il définit alors comment ce plan est intégré à une architecture existante. TranSAT est considéré comme un canevas de conception d'architecture qui permet de maitriser l'évolution dans une architecture logicielle par transformation de cette dernière en ajoutant une dimension dans la structuration d'une architecture logicielle : la dimension des préoccupations. Cette dimension permet de construire pour chaque préoccupation un plan correspondant à un assemblage de composants dont le but est de réaliser les services liés à cette préoccupation. TranSAT fournit alors une approche par transformation afin de réaliser le tissage de ce plan avec le plan de base. L'architecture de base est spécifiée à l'aide de SafArchie [22], reposant sur cinq concepts architecturaux (composant composite, composant primitif, liaison, port, opération) ainsi que sur la notion de contrat qui décrit un ensemble de contraintes sur le lieu sur lequel le nouveau plan peut être intégré.

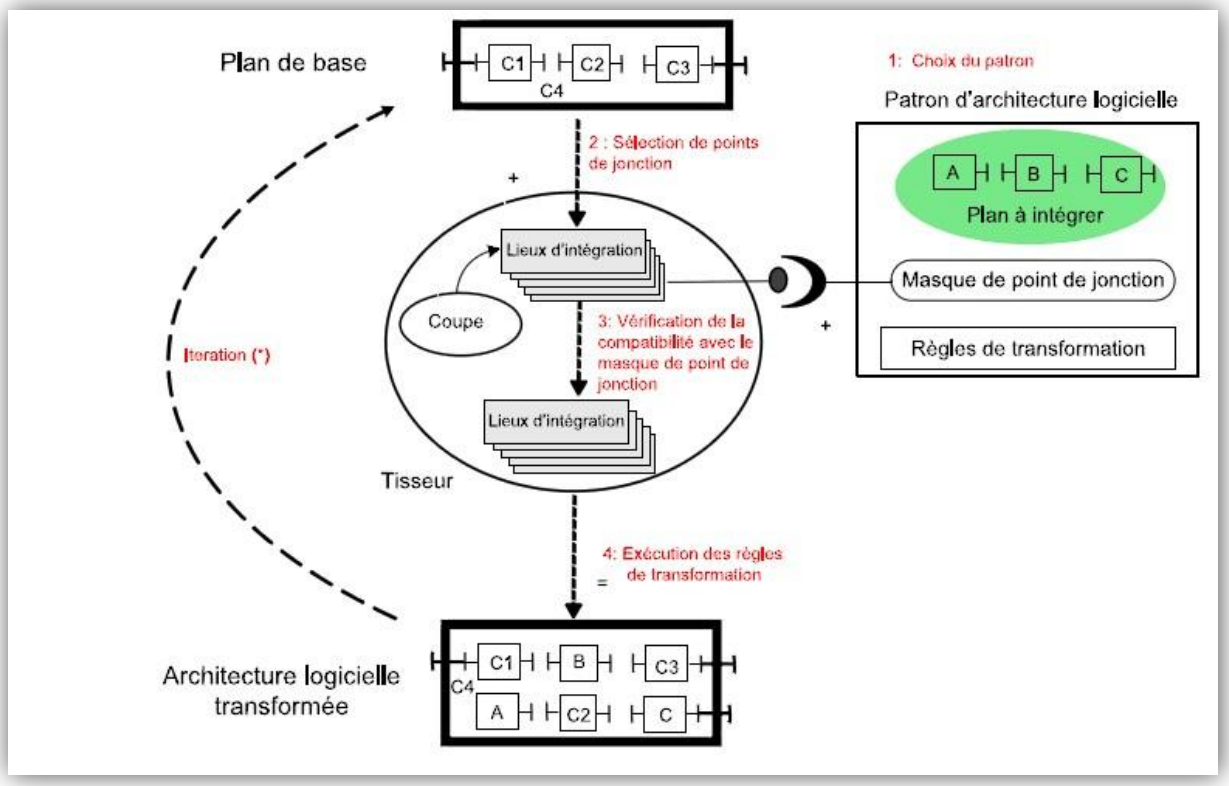


Figure 2.14: Vue générale de TranSAT

L'évolution dans TranSAT inspiré par les technologies des aspects, la notion de patron d'architecture est introduite pour structurer les différentes préoccupations transverses d'une architecture. Ce patron comprend les éléments à intégrer, les transformations à apporter à l'architecture de base, mais aussi un ensemble de contraintes génériques sur les éléments d'une architecture cible sur laquelle le patron peut être intégré. En particulier, le langage offert pour exprimer les règles de transformation adresse l'évolution structurelle à travers un ensemble de douze primitives. Il s'agit d'opérateurs d'ajout, de suppression et de déplacement appliqués aux composants (primitifs ou composites) et aux liaisons, ainsi qu'aux ports et à leurs opérations.

TranSAT permet de formuler l'évolution, mais ne gère pas les impacts, pas plus qu'il ne garde la trace de l'évolution. Il travaille au niveau de la spécification d'une architecture.

L'architecture de TranSAT est constituée de :

-Un *plan* représente un assemblage de composants mettant en œuvre les services liés à une préoccupation,

– Un *masque de point de jonction* décrit un ensemble de contraintes sur le lieu sur lequel le nouveau plan peut être intégré. Le masque de point de jonction a un double rôle dans TranSAT : premièrement, c'est le contrat entre le plan de base qui est modifié et le patron d'architecture. Il précise sous quelles conditions ce plan de base et le plan contenu dans le patron peuvent être tissés. Deuxièmement, ses éléments sont utilisés pour décrire les modifications à apporter pour intégrer une nouvelles architecture,

– *les règles de transformation* spécifient les modifications à apporter au niveau du plan de base et du plan de ce patron afin de permettre leur tissage créant une architecture logicielle enrichie de la nouvelle préoccupation.

Le résultat d'une transformation est une nouvelle architecture logicielle contenant l'ensemble des éléments du plan de base et du nouveau plan. Cette architecture peut alors servir à nouveau de plan de base pour l'intégration d'une nouvelle architecture. Une nouvelle itération peut alors être effectuée pour intégrer de manière incrémentale d'autres préoccupations.

2.5.2.3 SAEM :

SAEM (Style-based ArchitecturalEvolution Model – SAEM) [23] est un modèle d'évolution à base de styles qui sont dotés de qualités descriptives et prescriptive de l'évolution. La première qualité sert à véhiculer des connaissances et partager un vocabulaire d'évolution commun entre les architectes. La seconde qualité sert à guider l'architecte dans son activité d'évolution, en contraignant l'espace des solutions possibles qui prend en compte la problématique de l'évolution structurelle des architectures logicielles, indépendamment de tout ADL.

Dans ce modèle, une évolution est considérée comme un style et attachée à chaque élément architectural, en tant que partie intégrante de sa description tout en étant distincte.

Les concepts de base pour la gestion d'évolution de ce modèle sont définis comme suit :

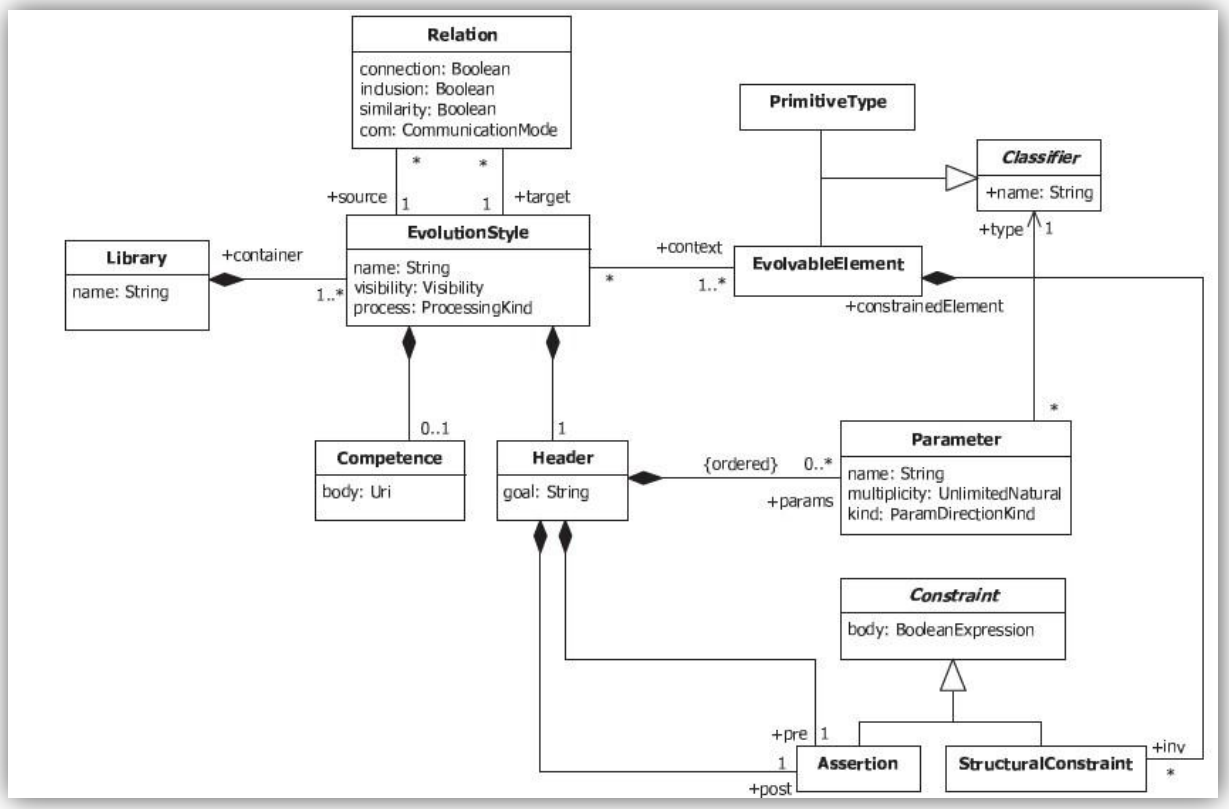


Figure 2.15: Meta modèle SAEM

- **Elément évolutif** : représente une abstraction forte qui peut recouvrir n'importe quel élément architectural dont les instances sont susceptibles d'évoluer, alors il devient possible de lui associer des styles d'évolution. Un style d'évolution encapsule ce qui permet de décrire et d'appliquer une évolution à un élément architectural.

- **Style d'évolution** : est une entité composée de deux parties complémentaires : un entête et une compétence :

- L'entête possède une description informelle du but poursuivi et publie une liste de paramètres et d'assertions. L'élément évolutif apparaît comme un paramètre implicite nommé contexte. Le type des paramètres est fourni par l'ensemble des éléments évolutifs, plus les types primitifs usuels (String, int, boolean, float, etc.)
- La compétence décrit une unité d'implémentation correspondant à l'entête. L'unité d'implémentation spécifie le flot des données et toute la logique de contrôle. L'implémentation est considérée comme une ressource externe, identifiable par son URI (Uniform Resource Identifier).

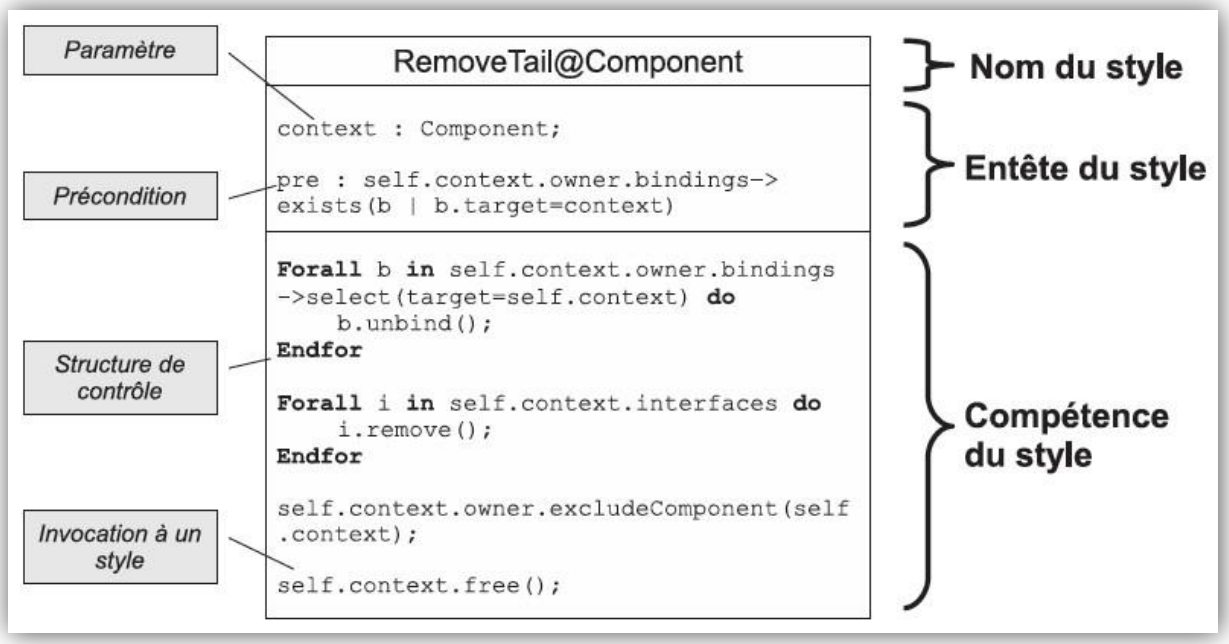


Figure 2.16: Nom, entête et compétence d'un style d'évolution.

La figure 2.16 présente un style *RemoveTail@Component* qui décrit la suppression d'un composant qui se trouve à l'extrémité de sa configuration d'appartenance (`self.context.owner`). Ceci est exprimé par la précondition `self.context.owner.bindings->exists(b|b.target=context)`, qui indique que le composant est la cible d'au moins un binding de la configuration. Le compartiment de la compétence de ce style indique que pour la suppression d'un tel composant, il faut, dans l'ordre :

- Supprimer les bindings auquel le composant participe.
- Supprimer toutes les interfaces du composant.
- Exclure le composant de la liste des composants de la configuration.
- Détruire le composant proprement dit, par dés-allocation de celui-ci.

- **Bibliothèque:** Une bibliothèque est une unité structurante destinée à contenir un ensemble de styles d'évolution. Elle expose une interface, autorisant la réutilisation de tout ou partie de son contenu. Une bibliothèque fournit un espace de nom spécifique et contrôle son interface vis-à-vis de l'extérieur en jouant sur la visibilité des styles d'évolution, qui peuvent être privés ou publiques. Ainsi, il est possible de spécifier des styles d'évolution uniquement visibles et

donc utilisables depuis l'intérieur de la bibliothèque. Par ailleurs, la bibliothèque constitue le point d'entrée unique pour l'accès aux styles d'évolution et est considéré comme un élément de modélisation racine.

Les mécanismes opérationnels fournis par SAEM sont :

L'instanciation, la spécialisation, la composition, et enfin l'utilisation.

L'instanciation :

D'une manière générale, l'instanciation est un mécanisme qui permet de passer d'un niveau de modélisation donné au niveau inférieur. Les styles d'évolution peuvent être instanciés plusieurs fois dans une architecture.

La spécialisation :

Les styles d'évolution peuvent être définis par extension d'autres styles. Le mécanisme d'héritage associé à la relation de spécialisation est inspiré du mécanisme d'héritage des classes dans le paradigme objet. Un sous-style peut ajouter et surcharger des éléments de l'entête de son super-style, et d'autre part un sous-style peut redéfinir la compétence de son super-style. Ce mécanisme peut être utilisé pour définir des styles d'évolution concrets en tant que sous-styles de styles abstraits en fournissant la compétence manquante.

La composition :

La composition de style se réfère à la structuration « tout-partie » entre deux styles. A chaque niveau de composition, chaque style peut être vu comme ayant pour parties ces sous-styles qui représentent des étapes entrant dans sa compétence. Le modèle SAEM utilise le mécanisme de composition pour définir des styles d'évolution composites, de plus en plus complexes, déléguant leurs fonctionnalités aux styles composants. La composition démarre d'un ensemble de styles d'évolution primitifs, dont la compétence est élémentaire. Enfin, le mode de communication attribué à la composition est nécessairement synchrone, car l'exécution des styles composants est subordonnée à celle du style composite.

L'utilisation :

L'utilisation est une relation permettant à un style de référencer un autre style pour en utiliser la fonctionnalité. Elle ne doit pas être confondue avec une relation de composition, car elle a un caractère plus momentané. Le mode de communication attribué à l'utilisation est asynchrone, car les exécutions des styles n'ont pas nécessairement besoin d'être concordantes.

2.5.2.4 ADL :

L'évolution n'est pas abordée comme étant une problématique à part entière dans les ADL [9]. Des études sont faites [9,26, 27] sur un ensemble d'ADL pour pouvoir identifier leurs

approches liées à l'évolution : Wright [31, 32],Darwin [33, 34], C2SADEL [35], ACME [36], SafArchie [37, 38], UML2.0 [39] , Mae [40, 41], xADL2.0 [42, 43,49]. La plupart des ADL considèrent l'évolution des *composants types* et des *configurations*. Pour les *connecteurs* et *interfaces types*, uniquement les ADL qui les considèrent comme entités de premières classes abordent leurs évolutions tels que Mae et xADL2.0. Peu d'ADL aborde l'évolution des *composants*, des *connecteurs* et des *interfaces*. La plupart des ADL considèrent uniquement l'évolution *statique* des architectures logicielles. Les ADL tels que Dynamic ACME [36], Dynamic Wright abordent l'évolution *dynamique anticipée* d'une architecture logicielle. Autrement dit, ils permettent d'exprimer des besoins d'évolutions prévus et préalablement connus. Aucun des ADL étudiés ne permet de refléter automatiquement les évolutions de l'architecture vers le code source. Seuls les outils ArchStudio (associé à C2SADEL) et *ArchEvol* ont tenté de répondre à cette préoccupation, qui constitue l'un des défis actuels de la communauté d'architectures logicielles.

Pour l'évolution *statique* des *composants* et des *connecteurs types*, la plupart des ADL se limitent au mécanisme de *composition hiérarchique*. D'autres ADL tels que C2SADEL et Mae proposent aussi le *sous-typage hétérogène*. xADL2.0 et Mae adoptent le *versionnement* pour faire évoluer une architecture logicielle. L'évolution *statique* considérée dans la majorité des ADL est avec *rupture* (la trace de la situation avant l'évolution, n'est pas sauvegardée) à l'exception d'*xADL2.0* et *Mae*.

Dans le travail sur la gestion des architecture évolutives dans ArchWare [28], La description d'architectures dynamiques est possible par la construction de couche composant-connecteur à partir du langage noyau complété par des actions dédiées au comportement dynamique (création dynamique d'éléments et reconfiguration). De plus, toute entité architecturale est potentiellement dynamique, sa définition servant à la création dynamique de plusieurs occurrences. Ainsi, la définition est celle d'une *méta-entité*, une matrice contenant à la fois la définition ainsi que des informations permettant la création, la suppression et la gestion de plusieurs occurrences. L'évolution d'un composite est prise en compte par un élément architectural dédié, le *chorégraphe*. Ce dernier est en charge de changer la topologie en cas de besoin : changer les attachements entre éléments architecturaux, créer dynamiquement de nouvelles instances, exclure des éléments de l'architecture, en inclure d'autres, etc. Les travaux menés au sein du projet ArchWare, incluent une machine virtuelle capable d'interpréter le code architectural. Dans sa version actuelle, la machine virtuelle interprète la

couche noyau du langage, ArchWare p-ADL. L'approche consistant à pouvoir modifier dynamiquement l'architecture en cours d'interprétation (par la machine virtuelle).

Dans le langage AADL[29] une architecture est décrite comme un ensemble de composants logiciels (processus, thread, thread group, data subprogram) qui s'exécute sur une plateforme d'exécution décrite par des composants matériels (processors, mémoires, devices, buses).Ce langage propose la notion de modes permettant de décrire un ensemble de configurations dynamiques, décrire un système comme un ensemble fini d'architectures qui représentent des configurations successives dans le temps. Des modifications dynamiques sont modélisées, mais prédéterminées, de l'architecture de l'application.
Dans [30]une approche intégrée d'architecture logicielle est définie, qui d'une part s'inspire des domaines complémentaires à l'architecture logicielle et d'autre part se distingue par des modèles homogènes pouvant s'adapter à divers domaines de l'informatique, un langage d'action SEAL qui est l'ADL de l'approche intégrée, où les actions sont les entités comportementales de base qui échangent des flots de contrôle et des flots de données à travers des points d'entrée de données et de sortie de données, et une méthodologie de conception qui consiste à automatiser une grande partie du processus d'élaboration d'une architecture afin de réduire les sources potentielles pouvant engendrer des incohérence lors de l'élaboration des architectures. L'automatisation porte essentiellement sur le contrôle des diverses décisions de l'architecte et sur la possibilité de mettre le processus dans des situations sûres à partir desquels, des actions de conception pourraient être entamées de manière plus saine. La démarche proposée est une démarche itérative, incrémentale et centrée sur la prise en charge des opérations exceptionnelles. Elle se base sur un certain nombre de concepts, les plus importants sont : le graphe de conception, l'arbre des instances, la notion d'exception dans le processus de conception, les situations saine de redémarrage du processus de conception et la notion d'architecture de déploiement.

2.6 Gestion de cohérence des modèles :

2.6.1 Gestion de cohérence des modèles orientés objet :

2.6.1.1 Praxis :

La solution proposée dans ce travail [12], traite la cohérence structurelle syntaxique de manière statique grâce aux règles d'incohérences. De plus la cohérence intra et inter modèle expliquées dans la section 2.4.1.3 sont également traitées. Un formalisme appelé formalisme

de Praxis, présenté dans la section 2.5.1.2, est utilisé afin de rendre la gestion de la cohérence indépendante de tout méta modèle. Une correction automatique des incohérences est présentée dans ce travail en utilisant un algorithme de recherche qui propose un plan de réparation pour toutes incohérences trouvées.

La solution proposée consiste à détecter des incohérences dans un modèle après évolution en utilisant des règles appelées *règles d'incohérence* (*Règle de détection d'incohérence, Règle de détection des causes d'incohérence*).Dans le but de rendre un modèle cohérent, cette solution automatise la modification d'un modèle afin de le rendre cohérent à nouveau en proposant un plan de réparation comme suit :

1. Détecter les incohérences par le biais de règle logique (*Règle de détection d'incohérence)* appliquée à la séquence d'action de construction du modèle.

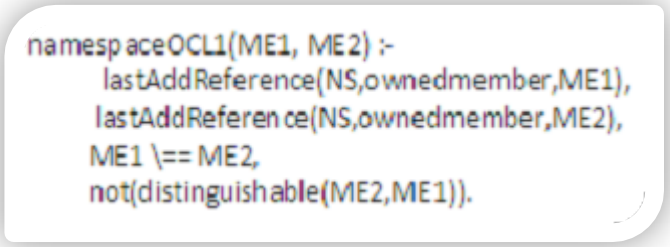


Figure 2.17: Règle de détection d'incohérence

Cette règle détecte les paires d'éléments de modèle *ME1* et *ME2* qui ont les mêmes noms *NS*, mais ne sont pas distinguables.

2. Identifier les causes possibles pour chaque incohérence trouvée en utilisant *les règles de détection de la cause de l'incohérence* qui sont générées à partir de la règle de détection d'incohérence.

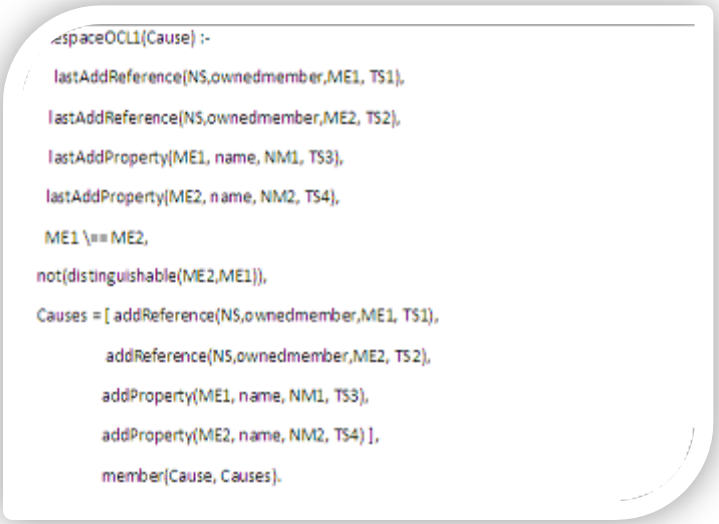


Figure 2.18: Règle de détection de la cause de l'incohérence

La règle de détection de la cause de l'incohérence présentée dans la figure 2.18 ci-dessus est générée à partir de la règle de détection d'incohérence. Cette règle identifie quatre causes possibles pour les incohérences détectées: qui peuvent être les actions qui ont ajouté les éléments *ME1 ME2* avec le même nom *NS*, ou bien les actions qui ont défini leurs noms.

3. Des générateurs de fonctions sont utilisés qui déterminent un ensemble de listes d'actions de correction pour les causes d'incohérences trouvées.

Figure 2.19: Exemple de générateur de fonction

La première règle annule les incohérences causées par l'action *addProperty*. Elle dit que chaque fois qu'une action *addProperty (E, nom, OldName, TS)* est une source d'incohérence, il existe un plan qui peut la corriger : supprimer l'ancienne valeur de la propriété par l'exécution de l'action *remProperty (E, nom, OldName)* et en définissant un nouveau nom par le biais de l'action *addProperty (E, nom, NewName)*.

4. Un plan de réparation est généré en utilisant un algorithme de recherche *IDFSS* en parcourant les règles qui ont causé l'incohérence ainsi que la liste d'action de correction avec une profondeur limitée.

Figure 2.20: Exemple plan de réparation

2.6.1.2 BDD :

- Le modèle de gestion et d'évolution de schéma présenté dans la section 5.1.3 [15] traite la cohérence dynamique des manipulation en proposant un mécanisme permettant aux utilisateurs d'assurer que les différentes sémantiques d'évolution n'engendrent pas des incohérences au niveau de la base en utilisant trois types de contrainte d'intégrité (*les contraintes locales à une version de type ou de schéma, les contraintes globales à toutes les versions d'un type ou d'un schéma, les contraintes globales au système (à tous les types et tous les schémas*). La cohérence structurelle des objets est assurée par des règles de visibilité pour la gestion de ces points de vues et des règles de transformation pour éviter certaines incohérences des objets vis à vis des différentes versions de leurs types. La cohérence structurelle des schémas quant à elle est assurée par des invariants de sous-typage est des contraintes.

-Dans l'approche R-D-W [21] traite l'évolution des schémas dans les entrepôts de données dans lequel des règles permettent d'intégrer les connaissances du domaine, exprimées par l'utilisateur. Ces règles définissent le lien d'agrégation entre deux niveaux de granularité dans une hiérarchie de dimension. Ces règles d'agrégation traduisent les connaissances du domaine exprimées par l'utilisateur lui-même ou extraites à l'aide d'un processus d'apprentissage. Ce sont des règles de type "*si-alors*". Dans la clause "*alors*" figure la définition du niveau de granularité supérieur, en fonction des conditions exprimées dans la clause "*si*" qui portent sur les niveaux de granularité inférieurs. Afin d'assurer la cohérence des analyses induit par les règles, des contraintes sur les règles sont définit : (1) validité du

domaine de définition (une règle doit se baser sur des données disponibles dans l'entrepôt) ; (2) validité temporelle (une règle a une durée de validité bien définie) ; et (3) cohérence des règles (deux règles ne doivent pas se contredire).

2.6.2 Gestion de cohérence des modèles orientés composant :

2.6.2.1 SAEV :

Le modèle SAEV [9] traite la cohérence sémantique et structurelle en utilisant des règles et stratégies d'évolutions afin d'automatiser autant que possible la détermination et la propagation des impacts d'une évolution. SAEV se base sur des invariants qui représentent des contraintes structurelle sur un élément architectural ou sur l'ensemble de l'architecture associés aux éléments architecturaux, et des propriétés sémantiques qui véhiculent des informations sur le degré de corrélation entre les éléments architecturaux définies au niveau connecteur pour les architecture à base de composant. Mais seuls les ADL [6] qui considèrent le concept de connecteur comme entité de première classe peuvent exploiter ces propriétés sémantiques.

Une *propriété sémantique* est définie comme une propriété associée à un connecteur pour exprimer le degré de corrélation ou de dépendance qui existe entre les deux interfaces composant fournie et requise qu'il relie :

• L'interface composant fournie attachée à un connecteur sous le terme d'*interface source* du connecteur;

• L'interface composant requise attachée à ce même connecteur sous le terme d'*interface cible* du connecteur.

Les propriétés sémantiques suivantes sont proposées :

• L'Exclusivité / Partage :

• **L'Exclusivité** : un connecteur définissant la propriété d'exclusivité (E), dit connecteur *exclusif*, indique que les deux interfaces source et cible de ce connecteur interagissent uniquement entre elles. Ceci implique que l'interface source ne peut fournir ses services que vers la seule interface cible de ce connecteur et l'interface cible ne peut recevoir ses services requis que de la seule interface source de ce connecteur. Ainsi, aucun autre connecteur ne peut être attaché aux interfaces source et cible d'un connecteur exclusif.

• **Le Partage** : à l'inverse, un connecteur définissant la propriété de partage (P), dit connecteur *partagé* indique que les interfaces source et cible de ce connecteur peuvent

interagir avec un nombre quelconque d'interfaces. Ainsi, plusieurs connecteurs peuvent être attachés aux interfaces source et cible de ce connecteur.

• La Dépendance / Indépendance :

• **La Dépendance** : la propriété de dépendance indique qu'il existe une dépendance existentielle de l'interface cible d'un connecteur vis-à-vis de son interface source. L'**Indépendance (I)** : à l'inverse, un connecteur *indépendant* (I) indique que l'existence de l'interface cible du connecteur est indépendante de l'existence de son interface source.

• La Prédominance / Non prédominance :

• **Prédominance** : cette propriété exprime une dépendance existentielle, le sens de cette dépendance existentielle est symétrique à celui de la propriété de dépendance. En effet, la prédominance met l'accent sur le fait que l'existence d'une interface source d'un connecteur est liée à l'existence de son interface cible, alors que la dépendance exprime le fait que l'existence de l'interface cible dépend de l'existence de l'interface source de ce connecteur.

• **Non prédominance** : la propriété de *non prédominance* (NPR) indique que la suppression d'une interface cible d'un connecteur n'entraîne pas la suppression de son interface source.

• La Cardinalité /Cardinalité inverse :

La *Cardinalité* d'un connecteur indique le nombre d'instances de l'interface cible de ce connecteur qui peuvent être attachées à une même instance de l'interface source de ce connecteur (autrement dit, le nombre d'instances de ce connecteur qui peuvent être attachées à une même instance d'interface source). La *Cardinalité inverse* d'un connecteur indique le nombre d'instances de l'interface source qui peuvent être attachées à une même instance de l'interface cible de ce connecteur (autrement dit, le nombre d'instances de ce connecteur qui peuvent être attachées à une même instance de l'interface cible).

La cardinalité et la cardinalité inverse peuvent être exprimées sous forme d'une valeur fixe ou sous forme d'un intervalle en indiquant la *valeur minimale* et la *valeur maximale* de cet intervalle.

Le modèle SAEV prévient les incohérences pendant l'évolution via les règles et stratégies d'évolution prédéfinies en prenant en compte la vérification des propriétés sémantiques en utilisant le formalisme ***ECA*** (évènement/ Condition/ Action) [45] voir exemple dans la figure 2.21**.** Ce dernier est composé de trois parties : une partie événement : l'événement est émis par l'environnement (concepteur ou par une autre règle), une partie condition : exprime ce qui doit être nécessairement vérifié pour exécuter une règle d'évolution (vérification des invariant et propriétés sémantiques), une partie action : décrit les changements, à proprement parlé, à

appliquer sur l'élément architectural auquel est associée la règle d'évolution. Une vérification de la cohérence est lancée après évolution en vérifiant les invariants. Dans le cas où des incohérences sont détectées une intervention manuelle est nécessaire pour corriger les incohérences.

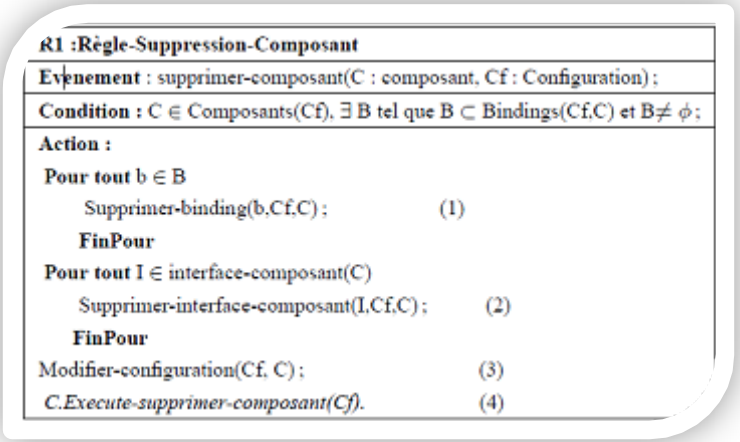


Figure 2.21: Exemple de règle d'évolution

L'inconvénient du formalisme ***ECA*** est qu'il peut annuler l'application d'une opération d'évolution suite à une condition non vérifiée, alors que cette dernière peut être revérifiée par l'application des autres opérations d'évolution par la suite. Un exemple serait l'ajout d'un lien entre le composant *Client* et le composant *Serveur2* via le connecteur *RPC* avec la propriété cardinalité égale à [1.10] sachant que ce *Client* est déjà connecté à un autre composant *Serveur1*. Cette opération d'ajout provoque une incohérence liée à la propriété cardinalité. Une autre opération de modification de cette propriété à la valeur 2 corrige cette incohérence comme montré dans l'exemple de la figure 2.22.

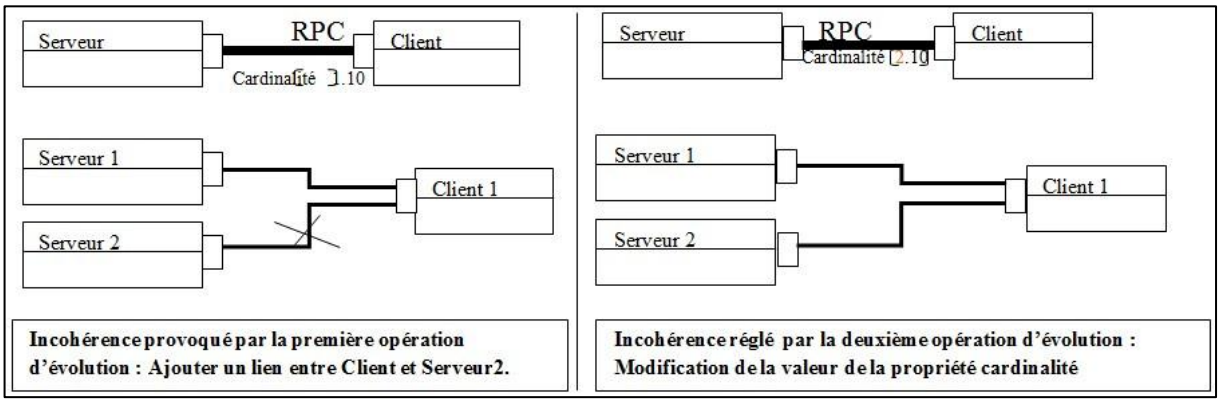


Figure 2.22: Opération d'évolution et cohérence

Cette cardinalité [1.10] indique qu'il est possible d'attacher au maximum 10 instances d'interfaces du composant *Client* à une même instance d'interface du composant *Serveur*, par contre une instance de l'interface composant *Client* doit être reliée uniquement à une seule instance de l'interface du composant *Serveur*.

2.6.2.2 SAEM :

SAEM [23] est un modèle d'évolution conçu pour formuler l'évolution sous forme de styles et également pour propager les impacts des évolutions à travers les relations sémantiques inter-styles pour garantir la cohérence globale de l'architecture. Les stratégies de propagation sont véhiculées par les relations sémantiques qui existent entre les styles d'évolution. SAEM introduit ce que l'on peut appeler un "langage de styles". Un langage de styles est une collection de styles formant un vocabulaire et une démarche pour comprendre et communiquer des idées portant sur l'évolution logicielle. Il décrit la manière avec laquelle les styles sont inter-reliés et la manière avec laquelle ils se complètent dans un tout cohésif.

La suite des concepts de base, présentés dans la section 2.5.2.3, pour la gestion de la cohérence de ce modèle sont définis comme suit :

- Relation : le concept de relation entre deux styles d'évolution est définit de façon générique par un triplé d'éléments relationnels, auquel est associé un mode de communication : (a) de connexion, (b) d'inclusion et (c) de similarité permettant de déterminer la nature de la sémantique portée par une relation entre deux concepts. Chaque relation est dirigée et supporte une communication par message entre le style « émetteur » et le style « receveur » dont le mode peut être synchrone ou asynchrone. :

– Synchrone : Le style d'évolution émetteur doit attendre la terminaison du style receveur pour pouvoir continuer son exécution.

– Asynchrone : Le style d'évolution émetteur peut continuer son exécution sans même attendre la terminaison du style receveur.

- contraintes : ont pour objectif principal de permettre de contrôler l'évolution des éléments d'une architecture. La classe abstraite *Constraint* est dérivée en deux sous-classes :

- Contrainte Structurelle : représente une contrainte sur la structure d'un élément évolutif. Elle doit être respectée tout au long du cycle de vie de l'élément.
- Assertion : représente des contraintes sur l'état de tout ou partie de l'architecture avant, pendant, et après son évolution.

2.6.2.3 Assistance à l'évolution :

Dans le travail [7], des contrats sont utilisés afin de documenter les droits et devoirs de deux parties : le développeur de la précédente version du logiciel qui s'engage à garantir les attributs qualité, et le développeur de la nouvelle version qui s'engage à respecter des contraintes architecturales que le premier avait établies.

Le vocabulaire suivant est associé aux contrats d'évolution :

NFP (Propriété Non Fonctionnelle) : clause dans le document de spécification non fonctionnelle du logiciel relative à un attribut qualité (par exemple la clause : "Le service de transfert devra s'exécuter en moins de 10ms relative à l'attribut qualité Performance).

AD (Décision Architecturale) : partie de l'architecture du logiciel qui cible une ou plusieurs **NFP** (par exemple le respect d'un style en Pipeline à un endroit particulier de l'architecture du logiciel). Pour sa description, une **AD** peut être construite, si nécessaire et dans un souci de factorisation, sur la base d'autres AD.

NFT (Tactique Non Fonctionnelle) : couple (**AD**, **NFP**) définissant un lien entre une décision architecturale **AD** et une propriété non fonctionnelle **NFP** que vise cette décision (par exemple l'**AD** spécifiant un style en pipeline avec une NFP relative à la performance d'un service particulier offert par le logiciel) ;

NFS (Stratégie Non Fonctionnelle) : ensemble de toutes les NFT définies pour un logiciel particulier.

Cette approche par contrats vise, à automatiser certaines vérifications de la qualité architecturale durant l'évolution. Ces vérifications peuvent, à la demande du développeur, assister l'activité d'évolution et lui notifier l'impact des changements architecturaux sur les décisions architecturales.

Un ensemble de règles sont introduites qui définissent les droits et devoirs d'un chargé de l'évolution. Un suivi strict de ces règles doit limiter le risque pour un logiciel d'atteindre un état incohérent :

Règle 1 : " « *Une version acceptable d'un logiciel est un système où chacune des NFP est impliquée dans au moins une NFT* ». Cette condition garantit, à la fin du processus d'évolution, qu'il n'existe aucune NFP pendante (sans AD associée). Le non-respect de cette

règle implique soit le refus de la création d'une nouvelle version, soit l'obligation (en toute connaissance de cause) de modifier la spécification pour lui retirer les NFP incriminées.

Règle 2 : « *L'abandon d'une AD n'est pas interdit lors d'un pas d'évolution. L'abandon est simplement notifié en précisant les NFP affectées (celles apparaissant avec l'AD dans une NFT)* ». Il revient au développeur, pleinement averti des conséquences, du maintien ou non des modifications. Si la modification est maintenue, les NFT correspondantes seront éliminées. Cette flexibilité est essentielle car, dans la suite, l'AD abandonnée pourrait être substitué par une autre à même de conserver les NFP ciblées. De plus, une décision peut être amené à être invalide temporairement pour effectuer une modification spécifique. Dans les deux cas, la **règle 1** imposera de documenter à nouveau les liens unissant la nouvelle AD et les NFP concernées.

Règle 3 : " « *De nouvelles NFT peuvent être ajoutées à la NFS* ». Durant une ´évolution, de nouvelles décisions architecturales peuvent compléter ou remplacer les anciennes.

L'approche d'assistance d'évolution oriente le développeur durant l'évolution vers une certaine direction de telle manière à ce que la déviation des spécifications de qualité initiales soit minimale. Elle ne permet pas de choisir un composant donné pour une évolution particulière. L'évolution prise en compte est d'ordre architectural. Elle peut être la conséquence d'une évolution des spécifications fonctionnelles ou des spécifications non-fonctionnelles. L'assistance permet de notifier l'impact de l'évolution sur les décisions architecturales et sur la qualité. Elle ne propose aucune solution existante. Par contre, elle permet d'alerter de l'impact sur la qualité qu'a une évolution des spécifications, aussi bien fonctionnelles que non-fonctionnelles.

2.6.2.4 ADL :

Tous les ADL proposent des mécanismes pour exprimer des *contraintes*, des *invariants* et des *contrats* sur les éléments d'une architecture. Ces mécanismes sont importants pour le maintien de la cohérence de l'architecture après évolution. Dans **Dedale[26]** un ADL pour l'évolution des architectures à base de composants permet la représentation explicite et distincte des spécifications, des configurations et des assemblages qui constituent les trois dimensions des architectures. Les décisions de conception des architectures peuvent être énoncées et suivi tout au long du cycle de développement. Les évolutions peuvent intervenir dans n'importe laquelle des trois dimensions. Leur cohérence est vérifiée par des contraintes d'assemblage qui définissent les conditions qui doivent être vérifiées par les attributs des composants. Les

contraintes sont exprimées sur les rôles et sont appliquées sur les composants associés à ces rôles. Les contraintes définissent ainsi de manière explicite des conditions génériques qui sont respectées dans toutes les instanciations d'une architecture. Les contraintes proposées en Dedal définissent les valeurs imposées à certains attributs, des conditions entre attributs et des contraintes de cardinalités pour les connexions sur les interfaces.

Dans [27] propose un méta modèle pour les architectures dynamique. Une architecture est décrite par un graphe enrichi permettant d'intégrer les différents paramètres des composants et leurs interdépendances. Le concept de styles architecturaux est utilisé pour décrire les architectures dynamiques et la spécification de leurs instances consistantes. Un style architectural est caractérisé par un modèle basé sur des grammaires de graphes étendues. L'idée est d'utiliser l'aspect génératif des grammaires pour caractériser rigoureusement les instances consistantes de l'architecture. Toute configuration correspondant à un graphe généré par la grammaire sera considérée comme consistante. D'autre part, toute configuration qui ne peut être générée par la grammaire est considérée comme inconsistante et ne faisant pas partie du style architectural de l'application. Des règles de reconfiguration sont définies, par une combinaison de règles de transformation de graphes, afin de maintenir l'application dans un état consistant. Contraindre la reconfiguration à suivre un protocole prédéfini, permettant de gérer l'évolution dynamique, la préserve de se retrouver dans un état inconsistant. Les règles de reconfiguration, exécutées à la suite de la réception d'un événement donné, doivent prendre en compte deux aspects principaux : Le premier aspect concerne la consistance de la reconfiguration elle-même. Ceci revient à l'autoriser ou non selon qu'elle implique ou non une inconsistance de l'architecture. Les règles de reconfiguration doivent, donc, définir le contexte correct de leur application. Le deuxième aspect concerne la définition du processus de reconfiguration en spécifiant la combinaison d'actions élémentaires à exécuter (i.e. les composants et liens à introduire ou supprimer).

2.7 Bilan :

A partir des travaux étudiés sur la gestion d'évolution des modèles et la gestion de cohérence des modèles, nous présentons dans cette section un bilan récapitulatif. L'objectif de ce bilan est d'évaluer et repérer les avantages des différentes approches proposées qui peuvent servir de supports pour définir notre solution. Ce bilan est composé de trois parties : la première partie représente deux comparaisons, une sur les différents travaux relatifs à la gestion d'évolution et une autre sur les différents travaux relatifs à la gestion de la cohérence. La

deuxième partie représente un ensemble de constatations sur les limites des différents travaux étudiés. La troisième partie et la dernière présente nos objectifs à atteindre.

2.7.1 Critères de comparaison:

Les critères utilisés dans la comparaison des différents travaux présentés dans l'état de l'art, sont expliqués est classés selon la gestion de l'évolution et la gestion de cohérence comme suit :

2.7.1.1 Gestion d'évolution :

Le tableau 2.1 ci-dessous présente une comparaison des différents travaux sur leurs gestions de l'évolution, selon des critères expliqués dans les sous sections suivantes:

2.7.1.1.1 Niveau d'abstraction : Une évolution peut être appliquée sur un ou plusieurs niveaux d'abstractions qui sont : M0 : niveau application. M1 : niveau modèle. M2 : niveau méta modèle. M3 : niveau méta méta modèle.

2.7.1.1.2 Mode d'évolution (Statique/ dynamique) : L'évolution peut être réalisée à l'étape de spécification ou de conception du système «évolution statique ». Elle peut être réalisée à l'exécution de ce système « évolution dynamique ».

2.7.1.1.3 Technique utilisée : Plusieurs techniques sont utilisées pour gérer une évolution et sont présentées dans le tableau de comparaison 2.1.

2.7.1.1.4 Outil : Ce critère précise si des outils sont développés pour les solutions proposées par les travaux, présentés dans l'état de l'art, sur la gestion de l'évolution.

2.7.1.1.5 Dépendance / Indépendance : La gestion d'évolution peut être dépendante d'un certain méta modèle ou d'un contexte bien défini. Sinon elle peut être généraliste et ne s'applique pas uniquement sur un contexte bien particulier dans ce cas la gestion de l'évolution est indépendante de tout méta modèle.

Gestion d'évolution							
Type de modèle	**Modèle**		**Niveau d'abstraction**	**Mode (Statique/ Dynamique)**	**Dépendance/Indépendance du méta modèle**	**Technique utilisée**	**Outil**
Orienté objet	Coévolution et gestion de version dans MDE		M1 & M2	Statique	Indépendant de tout méta modèle	Représentation de différence par le Delta modèle en utilisant des techniques de transformation et de versionning.	√
	Praxis		M1	Statique	Indépendant de tout méta modèle	-Transformation des modèles en formalisme de Praxis -Protocole de propagation des changements -Stratégie de résolution de conflit.	√
	BDD	Modèle de gestion et d'évolution de schéma	M0	Statique	Contexte des BDD	-L'existence simultanée de plusieurs *points de vue* d'un objet : *vues d'instances objets, vues de la base d'objets, vues de la méta base*	X
		DB-MAIN	M0	Statique	Contexte des BDD	-Propagation automatique des modifications - Analyse d'historique d'évolution - Analyse différentielle des conceptions logique et physique	√
Orienté composant	SAEV		M0, M1, M2, M3	Statique	Contexte des ADL	-Stratégies d'évolution -Formalisme ECA : Evénement, Condition, Action	X
	TranSat		M0	Statique	Contexte des ADL	-Dimension de préoccupation, -Tissage de plan -Règle de transformation	X
	SAEM		M0, M1, M2, M3	Statique	Contexte des ADL	-Langage de style d'architecture dotée de qualités descriptives et prescriptive de l'évolution	X
	ADL		M0	Dynamique + Statique	Contexte des ADL	Plusieurs techniques sont utilisées	√

Tableau 2.1: Tableau de comparaison des travaux sur la gestion d'évolution des modèles

2.7.1.2 Gestion de cohérence :
Le tableau 2.2 ci-dessous présente une comparaison des différents travaux sur leurs gestions de la cohérence, selon des critères expliqués dans les sous sections suivantes:

2.7.1.2.1 Cohérence traitée : plusieurs types de cohérences peuvent être traitées est sont expliquées dans la section 2.4.1.

2.7.1.2.2 Type de solution d'incohérence (manuelle, automatique) : les incohérences peuvent être corrigées par l'utilisateur «correction manuelle ». Sinon elles peuvent être corrigées sans intervention de l'utilisateur « correction automatique ».

2.7.1.2.3 Technique utilisée : Plusieurs techniques sont utilisées pour gérer une cohérence et sont présentées dans le tableau de comparaison 2.2.

2.7.1.2.4 Plan de réparation *:* pour la correction des incohérences détectées un plan de réparation peut être proposé.

2.7.1.2.5 Outil *:* Ce critère précise si des outils sont développés pour les solutions proposées par les travaux, présentés dans l'état de l'art, sur la gestion de cohérence.

2.7.1.2.6 Dépendance / Indépendance *:* La gestion de la cohérence peut être dépendante d'un certain méta modèle ou d'un contexte bien défini. Sinon elle peut être généraliste et ne s'applique pas uniquement sur un contexte bien particulier dans ce cas la gestion de la cohérence est indépendante de tout méta modèle.

Gestion de cohérence								
Type de modèle	**Modèle**		**Cohérence traitée**	**Type de solution d'incohérence (automatique/ manuelle)**	**Technique utilisée**	**dépendante /Indépendante**	**Plan De réparation**	**Outil**
Orienté objet	Praxis		Syntaxique Structurelle Inter-modèle Intra-modèle	automatique	-Transformation de graphe -Règle de cohérence -Plan de réparation	Indépendant de tout méta modèle	Sous forme d'action de correction en formalisme de Praxis.	√
	BDD	Modèle d'évolution de schéma	Structurelle comportementale	manuelle	-Contrainte d'intégrité, -règle de visibilité, -règle de transformation -invariant, -contrainte	Contexte des BDD	/	√
		R-D-W	Cohérence des analyses	Semi-automatique	-règle d'agrégation -contrainte sur les règles d'agrégations	Contexte des BDD	/	√
Orienté Compo--sant	SAEV		Sémantique	manuelle	-Propriété sémantique -règle d'évolution -invariant	Contexte des ADL	/	X
	SAE M		Structurelle	manuelle	-Contrainte -Relation	Contexte des ADL	/	X
	Assistance à l'évolution		Qualité logicielle	manuelle	Contrat d'évolution	Contexte ADL	/	X
	ADL		structurelle	manuelle	-Contrainte -invariant	Contexte des ADL	/	√

Tableau 2.2: Tableau de comparaison des travaux sur la gestion de cohérence des modèles

2.7.2 Limites des travaux étudiés:

Nous pouvons constater d'après l'état de l'art que la plus part des travaux étudiés :

1. Gèrent l'évolution statique ainsi que l'impact de l'évolution, l'évolution dynamique n'est abordée que par quelques ADL.
2. Se limitent à traiter différents types de cohérence mais pas la cohérence sémantique.
3. Dans le cas où la cohérence sémantique est traitée « par SAEV », la solution proposée est dépendante du contexte des ADL.
4. Proposent une correction manuelle des incohérences et ne proposent aucun plan de réparation à l'exception du modèle « Praxis ».

La question qu'on se pose est comment peut-on gérer la cohérence sémantique indépendamment de tout méta modèle et proposer une correction automatique des incohérences détectées après évolution.

2.7.3 Objectifs à atteindre :

En se basant sur notre étude et la comparaison faite sur les différents travaux effectués sur l'évolution et la vérification de cohérence des modèles, nous définissons les objectifs que nous voulons atteindre afin de proposer un modèle d'évolution qui doit assurer les points suivant :

1. Gérer l'évolution de manière statique ainsi que l'impact de l'évolution indépendamment de tout méta modèle.
2. Traiter la cohérence sémantique.
3. Proposer une solution pour la vérification de cohérence indépendante de tout méta modèle.
4. Définir les propriétés sémantiques pour la gestion de la cohérence sémantique.
5. Proposer une correction automatique pour les incohérences détectées après évolution.

2.8 Conclusion :

Ce chapitre a présenté les différents travaux effectués sur la gestion d'évolution et vérification de cohérence des modèles, ils sont classés selon deux types de modélisation : orientée objet et orientée composant. En se basant sur les différentes définitions et travaux trouvés dans la littérature, nous nous sommes positionnés par rapport à l'évolution et opération d'évolution. Nous considérons l'évolution comme un ensemble des changements et des transformations d'un système en utilisant les opérations d'évolution d'ajout, suppression, modification et les

activités de maintenance et d'adaptation comme étant des cas particuliers de l'évolution. La cohérence sémantique, une solution indépendante de tout méta modèle et une correction automatique des incohérences sont nos objectifs principaux à atteindre.

Partie 2 :
Conception du modèle évolution

Chapitre 3 :
Modèle d’évolution

3.1 Introduction :

Comme cité précédemment, notre principale problématique est liée à la gestion de l'évolution des modèles indépendamment de leurs méta modèle, en assurant une cohérence sémantique des modèles résultants. Pour cela, ce présent chapitre expose notre modèle nommé IMoSCM (***I**ndependent Model **S**émantique **C**onsistency **M**anagement)* pour une gestion automatique de la cohérence sémantique indépendamment de tout méta modèle lors de l'évolution des modèles.

Dans la section 3.2 nous présentons les principaux objectifs attendus et les différents concepts de notre modèle IMoSCM, la gestion d'évolution et la gestion de cohérence par le modèle IMoSCM sont présentées dans les sections 3.3 et 3.4 respectivement. Dans la section 3.5 nous terminons par un bilan qui positionne notre modèle par rapport aux limites des travaux étudiés dans le chapitre précédent, et les objectifs fixés dans la section 3.2.

3.2 Description d'IMoSCM :

Notre modèle IMoSCM (***I**ndependent Model **S**émantique **C**onsistency **M**anagement)* répond à la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs méta modèle, en assurant leurs cohérences sémantiques.

La gestion de l'évolution dans le modèle que nous proposons doit atteindre les objectifs suivants :

1. Gérer l'évolution ainsi que son impact indépendamment de tout méta modèle.
2. Traiter la cohérence sémantique indépendamment de tout méta modèle.
3. Proposer un plan de réparation et une correction automatique en cas d'incohérence.

3.2.1 Architecture :

Le diagramme de classes ci-dessous, voire la figure 3.1, présente les concepts proposés par IMoSCM, ainsi que les associations qui existent entre eux. Il présente principalement la manière dont sont organisés les concepts. La description détaillée de ces concepts est présentée dans les sections suivantes.

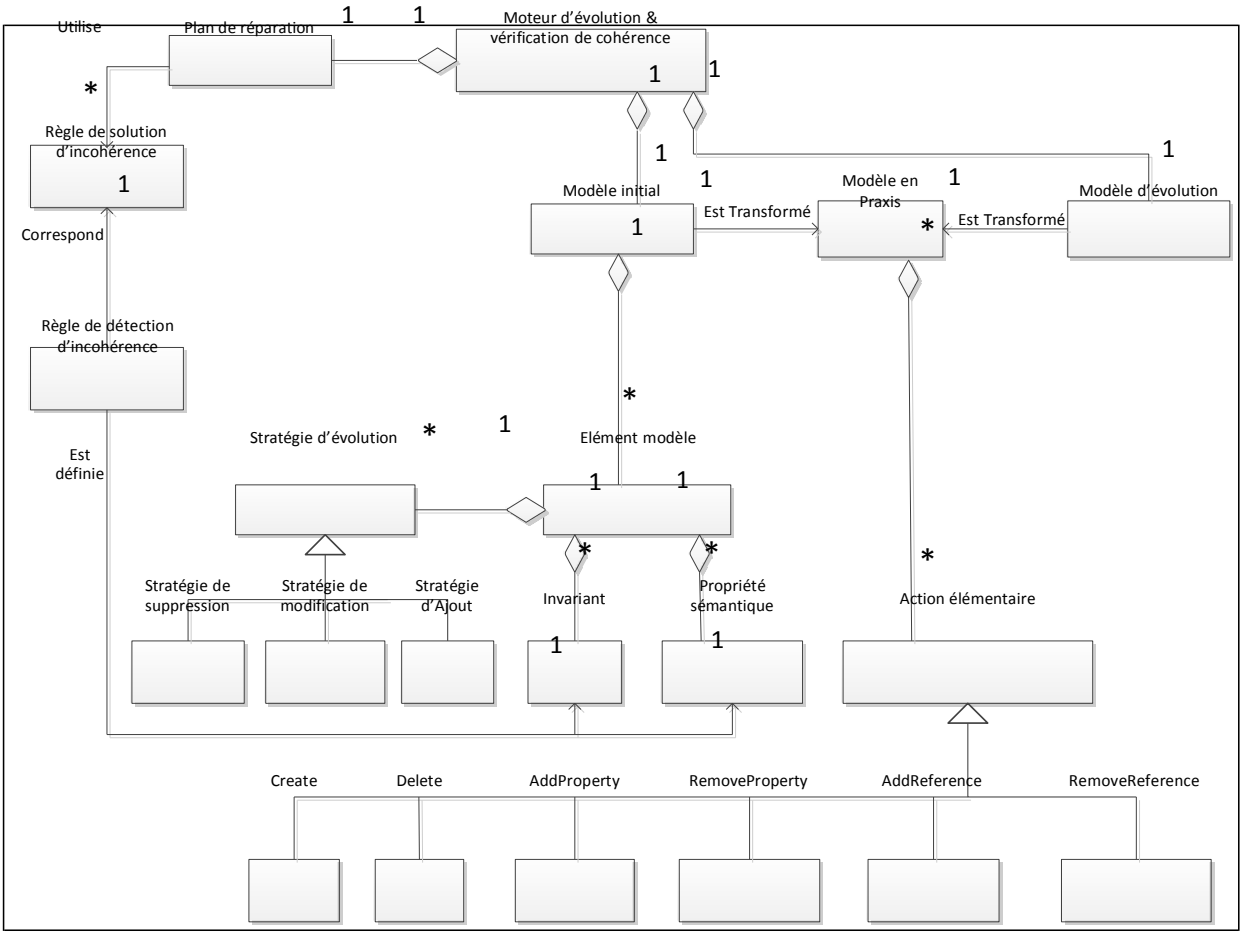


Figure 3.1: Architecture IMoSCM.

Le concept d'*Elément modèle* représente tous les éléments du modèle à faire évoluer. A chaque élément modèle correspond des *Invariants,* des *Stratégie d'évolution. Des propriétés sémantiques* sont également associées à certain de ces éléments. Une *stratégie* regroupe un ensemble de *Règles d'évolution,* décrivant l'application d'une opération d'évolution sur un élément modèle, qui sont composées *d'actions élémentaires.* Des *règles de détection d'incohérence* sont définies en utilisant les *invariants* et les *propriétés sémantiques* afin de détecter les incohérences après évolution. Le *plan de réparation* composé *d'actions élémentaires* est proposé par le *Moteur d'évolution et de vérification de cohérence* en utilisant les *règles de solution d'incohérence* définies pour chaque règle de détection d'incohérence. Nous décrivons en détail ces concepts dans les sections suivantes.

3.3 Gestion d'évolution par IMoSCM:

La gestion de l'évolution des modèles par IMoSCM est décrite par les sous sections ci-dessous :

3.3.1 Une évolution statique :

Le modèle IMoSCM traite l'évolution structurelle de manière statique. L'évolution structurelle se définie par les différents changements dans la structure des éléments du modèle à faire évoluer. Un changement appliqué sur un modèle est toujours généré par une ou plusieurs opérations appliquées à l'un de ses éléments, telles que la suppression, l'ajout, la modification appelées opérations d'évolution. L'évolution dynamique des modèles est laissée en perspective.

L'évolution est appliquée sur les éléments d'un modèle. Les éléments du modèle représentent tout élément susceptible d'évoluer constituant le modèle à faire évoluer. Des invariants, des propriétés sémantiques et des stratégies d'évolutions sont associés à chaque élément.

Exemples :

-Dans la modélisation UML: l'élément modèle pourrait être une classe, une opération…etc.

-Dans la modélisation à base de composant: l'élément modèle pourrait être une interface, un connecteur…etc.

Si on prend la modélisation à base de composant :

-Un exemple d'invariant associé à un connecteur : un connecteur doit relier au minimum deux composants.

-Un exemple de propriété sémantique associée à un connecteur : les deux interfaces source et cible de ce connecteur interagissent uniquement entre elles.

3.3.2 Une solution indépendante du méta modèle :

Le modèle en praxis [12] représente toute architecture logicielle (*orienté composant, orienté objet*) susceptible d'évoluer. Nous avons introduit ce concept pour pouvoir transformer et unifier tout modèle susceptible d'évoluer sous forme d'actions élémentaires en se basant sur ce formalisme.

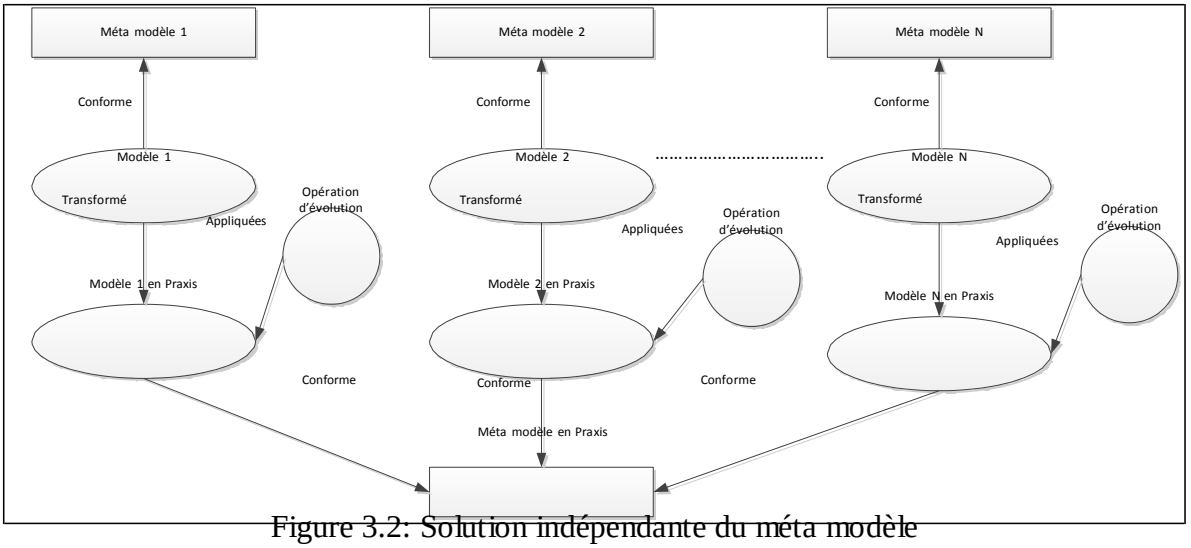


Figure 3.2: Solution indépendante du méta modèle

Avant d'appliquer les opérations d'évolution sur le modèle à faire évoluer, ce dernier est transformé sous forme Praxis. De ce fait l'application des opérations d'évolution se fait sur le modèle en Praxis, voire la figure 3.2. Ce qui permet une gestion d'évolution indépendamment du modèle à faire évoluer.

Dans le formalisme de Praxis [12], les modèles sont représentés par des séquences d'***actions élémentaires*** (*Create, Delete, AddReference, RemoveReference, AddProperty, RemoveProperty*) nécessaires à la construction de chaque élément de modèle. Chaque action est marquée avec un temps qui indique le moment où elle a été exécutée par le concepteur.

Exemples :

Exempe1 :

- La classe UML nommée *Server* composée d'une opération *Send()* est représentée en formalisme de Praxis comme suit :

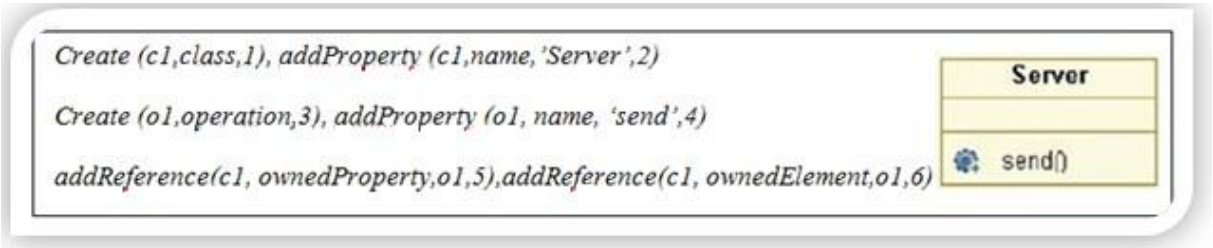


Figure 3.3: Classe UML en formalisme de Praxis

Exempe2 :

- Le composant *serveur* muni de deux interfaces cible *ICcp2* et source *IScp2* est représenté en formalisme de Praxis comme suit :

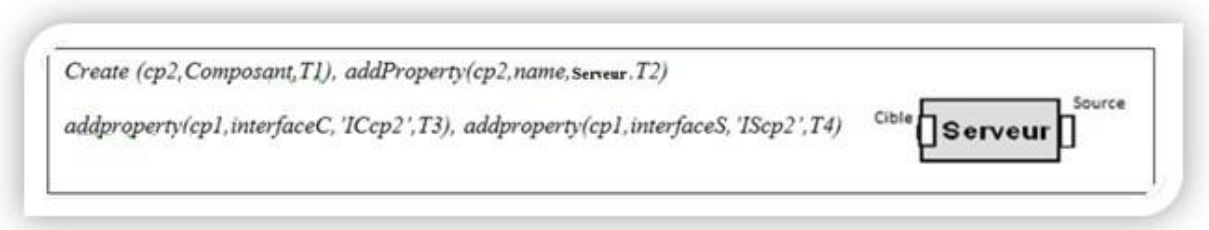


Figure 3.4: Composant *Serveur* en formalisme de Praxis

3.3.3 Technique utilisée :

On a vu précédemment que tout modèle à faire évoluer est transformé sous forme de Praxis. Donc les opérations d'évolution exprimées par le concepteur doivent être appliquées sur le modèle en Praxis. Pour cela nous proposons un modèle nommé *modèle d'évolution* construit à partir des opérations d'évolutions et les stratégies d'évolution exprimées par le concepteur. Ce modèle représente le delta modèle entre le modèle initial et le modèle après évolution, il est transformé sous forme d'action élémentaire en formalisme de Praxis pour être appliqué sur le modèle à faire évoluer qui est exprimé aussi en formalisme de Praxis, voire la figure 3.5. Cela permet une indépendance totale quant à l'application des évolutions de modèle.

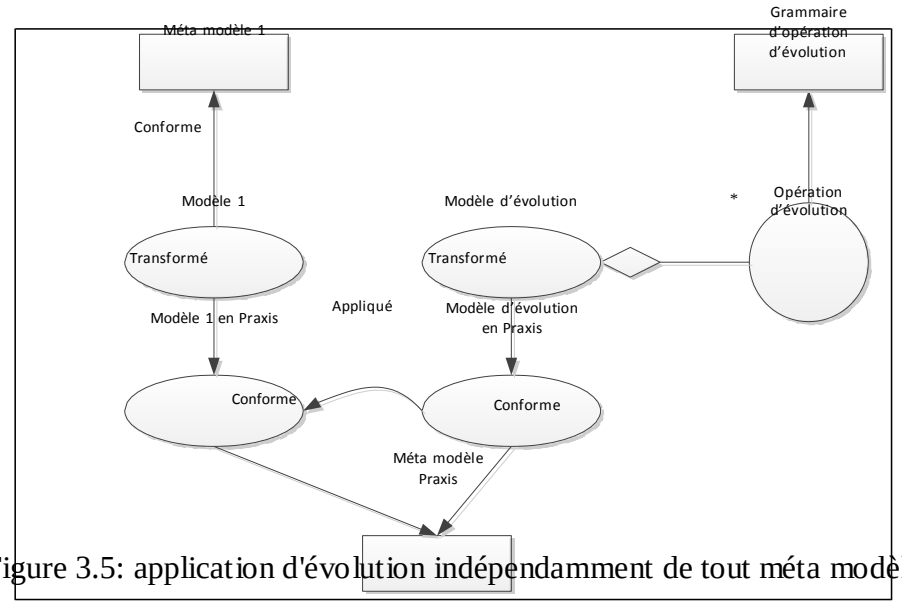


Figure 3.5: application d'évolution indépendamment de tout méta modèle

Des stratégies d'évolution sont utilisées afin d'appliquer les opérations d'évolution :

Une *stratégie d'évolution* permet de spécifier l'évolution d'un élément modèle. Elle est constituée d'un ensemble d'actions à exécuter pour chaque opération d'évolution appliquée sur un élément modèle, formant ainsi le *modèle d'évolution*. Les actions sont définies en se basant sur des invariants et des propriétés sémantiques à respecter afin de propager l'évolution en évitant d'avoir un état incohérent du modèle. Comme le modèle à faire évoluer ainsi que les actions à exécutées sont exprimées sous forme d'action élémentaire *creat, delete, remove, add*, la modification d'un élément se traduit par sa suppression suivi de sa création avec ses nouveaux paramètres. Donc pour chaque élément modèle, sont associées principalement les trois stratégies suivantes:

• **Stratégie d'ajout** : regroupe toutes les actions décrivant l'ajout d'un élément architectural ;
• **Stratégie de suppression** : regroupe toutes les actions décrivant la suppression d'un élément architectural.
Stratégie de modification : stratégie de suppression suivie de la stratégie d'ajout.

Exemple :
Afin de modifier le nom de l'opération de la classe *server*, les actions suivantes sont exécutées :

- Suppression de la propriété *Name* de l'opération avec son ancienne valeur.
- Ajouter une propriété *Name* à l'opération avec sa nouvelle valeur.

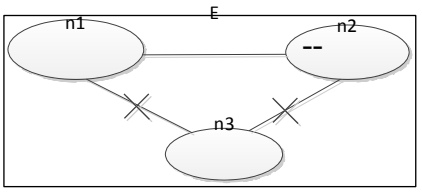
3.4 Gestion de la cohérence par IMoSCM :

La gestion de la cohérence sémantique des modèles par IMoSCM est décrite par les sous sections ci-dessous.

3.4.1 Une cohérence Sémantique :

La cohérence sémantique est assurée en utilisant des propriétés sémantiques. Une propriété sémantique exprime les contraintes sur les échanges de services entre deux éléments du modèle [9]. Notre modèle IMoSCM utilise les propriétés sémantiques proposées dans SAEV [9]. Nous avons implémenté que deux propriétés sémantiques, Exclusivité /Partage et Cardinalité/Cardinalité inverse. Les autres propriétés sémantiques, Dépendance/Indépendance et Prédominance/Non prédominance présentées dans [9] sont laissées en perspective.

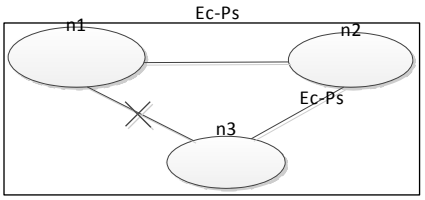
Exclusivité (E):



-Le nœud n1 ne peut être attaché qu'au nœud n2.

-Le nœud n2 ne peut être attaché qu'au nœud n1.

Exclusivité cible- Partage source (Ec-Ps):

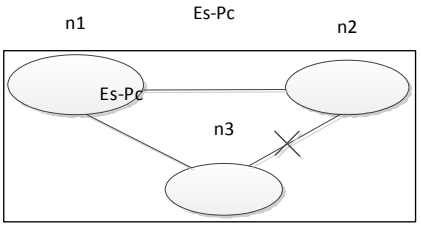


-Le nœud n1 peut être attaché à n'importe quel autre nœud.

- Les nœuds n2 et n3 ne peuvent être attachés qu'à un seul nœud.

Exclusivité source- Partage cible (Es-Pc) :

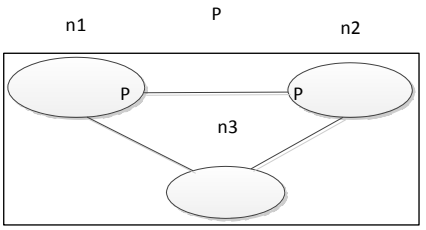
-Le nœud n2 peut être attaché à n'importe quel nœud.



-Les nœuds n1 et n3 ne peuvent être attachés qu'à un seul nœud.

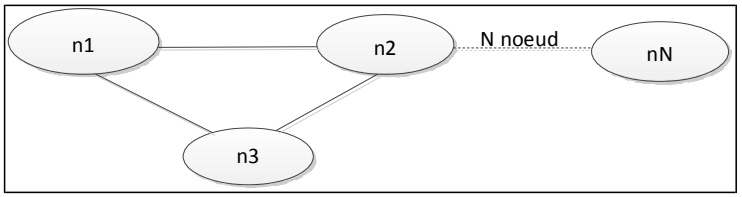
Partage (P):

Tous les nœuds peuvent être attachés à tous les autres nœuds.



1) Cardinalité et Cardinalité inverse :

La *Cardinalité et Cardinalité inverse* indique le nombre de nœud qui peuvent être attachés à un autre nœud. La cardinalité et la cardinalité inverse peuvent être exprimées sous forme d'une valeur fixe ou sous forme d'un intervalle en indiquant la *valeur minimale* et la *valeur maximale* de cet intervalle.



Le nœud n2 ayant la cardinalité [1.N] peut être attaché à N nœuds au maximum.

3.4.2 Technique utilisée:

3.4.2.1 Une vérification de cohérence prédéfinie :

La gestion de cohérence sémantique est assurée par des *règles de détection d'incohérence* prédéfinies par le concepteur. Ce dernier énumère toutes les possibilités d'incohérences possibles qui peuvent être causées par les actions constituant le modèle.

Le concept de règle de détection d'incohérence permet de détecter les incohérences après évolution. Elles permettent la vérification de la faisabilité d'une action d'évolution en se basant sur les propriétés sémantiques et les invariants. Elles retournent une liste des incohérences trouvées sous forme d'actions susceptibles de causer ces incohérences.

Exemple :

Règle de détection d'incohérence pour une propriété sémantique :

VerifePS (**Action d'évolution** *modification valeur*, **Actions du modèle** *Action*)
Si Propriété n'est pas respectée
alors Retourner un état incohérent + *modification valeur* cause de cette incohérence.

3.4.2.2 Une correction d'incohérence automatique :

La correction des incohérences est assurée par un *plan de réparation* qui est une suite d'action élémentaire, obtenue à partir des *règles de solution d'incohérence* prédéfinies par le concepteur. Ce dernier propose pour chaque règle de détection d'incohérence une règle de solution pour cette incohérence.

Exemple :

La règle ci-dessous représente une règle de vérification et de détection d'incohérence pour la propriété sémantique *cardinalité* du *nœud i* ayant une cardinalité égale à N. Suivi de la solution prédéfinie en cas d'incohérence.

Cardinalité (*nœud***i**, *cardinalité***N**)

Si (nombre de lien relié au nœud**i**) >**N** **Alors** {Retourner incohérence}

Solution *:*

Si (dernière action faite = modification de la valeur de la propriété)

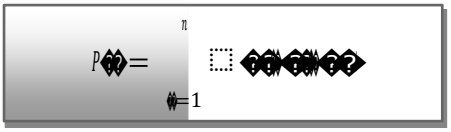
Alors {Modification de la valeur de la propriété avec une valeur (nombre de lien relié au nœud**i)**}
Sinon

{Supprimer les (nombre de lien relié au nœud**i – N)** derniers liens ajoutés}

La solution consiste à vérifier les actions dans le modèle d'évolution : si la dernière action concernant la propriété sémantique cardinalité est une modification de la valeur de cette dernière alors la solution est de modifier la valeur de la propriété avec une valeur égale au nombre de lien relié au *nœudi*. Sinon cela veut dire que la dernière action est un ajout de nouveau lien vers le *noeudi* alors la solution est de supprimer les liens en plus.

Le résultat de ces règles de solution est une suite d'action élémentaire constituant ainsi le plan de réparation.

=



En utilisant les actions du modèle d'évolution et les règles de solution d'incohérence pour chaque incohérence trouvée, une solution est proposée et rajouter au plan de réparation *PR*.

Reste à optimiser le plan de réparation en proposant d'autres solutions. L'optimisation du plan de réparation est laissée en perspective.

3.5 Mécanisme opératoire d'IMoSCM :

Nous présentons dans ce qui suit le processus général d'IMoSCM. Ce processus est composé de trois phases détaillées dans les sections suivantes : la spécification de l'évolution, l'application de l'évolution et la vérification de la cohérence. La figure 3.6 ci-dessous présente le processus général du modèle.

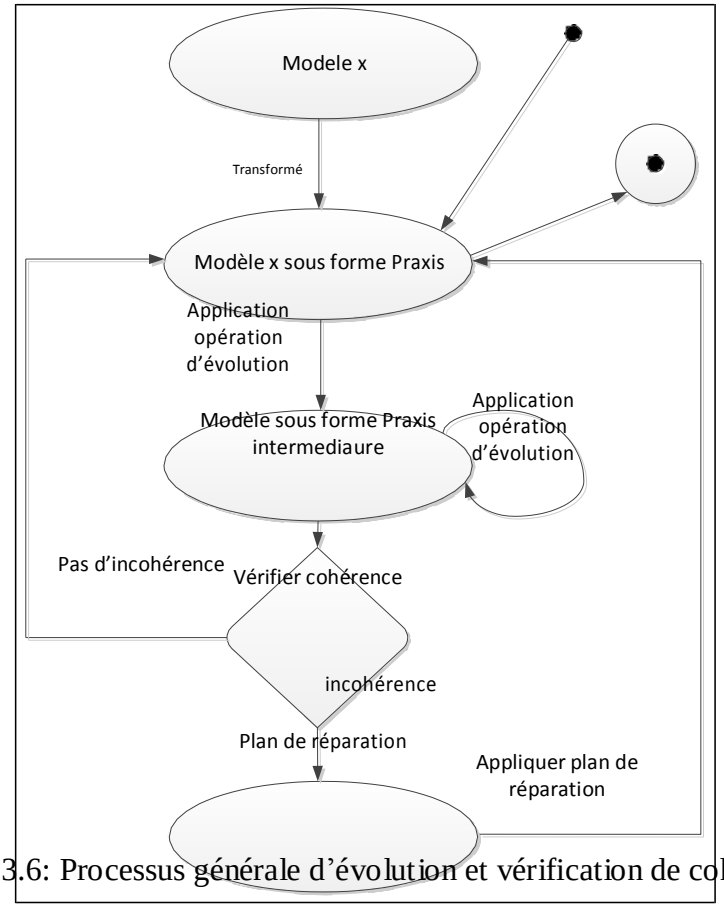


Figure 3.6: Processus générale d'évolution et vérification de cohérence

Le modèle à faire évoluer est transformé vers un modèle Praxis auquel des opérations d'évolutions sont appliquées en utilisant les stratégies d'évolutions. La cohérence sémantique du modèle est ensuite vérifiée en utilisant les règles de détection d'incohérence, dans le cas où la vérification retourne des incohérences un plan de réparation est proposé afin de corriger les incohérences trouvées et retourner à l'état cohérent.

3.5.1 Spécification de l'évolution :

Afin de pouvoir faire évoluer un modèle X, il faudrait spécifier le type du modèle, les éléments du modèle à faire évoluer. Pour chaque élément du modèle spécifier :

- Les invariants.
- Les stratégies d'évolutions.
- Les propriétés sémantiques.
- Pour chaque propriété sémantique spécifier :
    - Les règles de détections d'incohérence.
    - les solutions pour chaque incohérence trouvée.

Exemple :

Pour être indépendant de tout méta modèle nous choisissons de présenter un exemple de modèle par un graphe sous forme de nœud et d'arcs, voire la figure 3.7. Nous spécifions une évolution en donnant un exemple d'invariant, de propriété sémantique, stratégie d'évolution et règle et solution de cohérence.

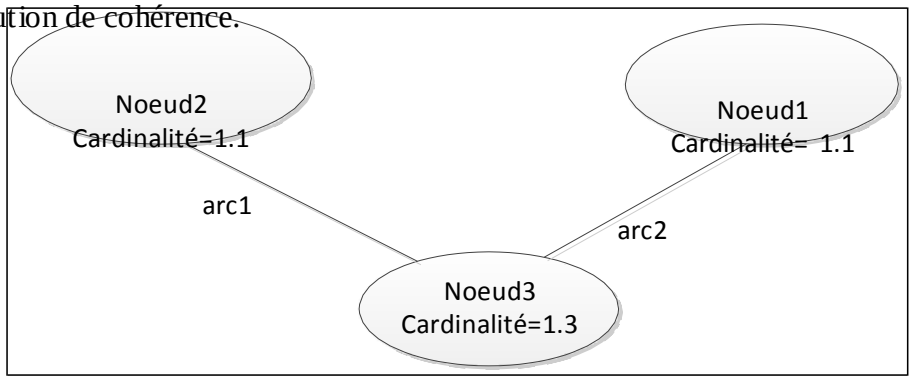


Figure 3.7: Modèle sous forme de nœud relié par des arcs

a)Invariant :

- un nœud doit être relié au moins à un autre nœud.

b) Propriétés sémantique :

Nous donnons comme exemple de propriété sémantique la cardinalité qui définit le nombre de lien entre deux nœuds.

- **cardinalité** : - $nœud_1$ peut être relié à un et un seul nœud. *Cardinalité 1.1*
  - $nœud_2$ peut être relié à un et un seul nœud. *Cardinalité 1.1*
  - $nœud_3$peut être relié au maximum à trois nœuds. *Cardinalité 1.3*

c) Stratégie d'évolution : Comme les stratégies sont définies pour chaque élément du modèle, nous prenons les deux éléments de l'exemple de la figure 3.7 pour les exemples ci-dessous.

Nœud :

Ajout : Aouter un nœud revient à ajouter le nœud ainsi que toutes ses propriétés.

Suppression : supprimer un nœud revient à :

- Supprimer tous les liens raccordés à ce nœud.

- Supprimer le nœud.

Arcs :

Ajout : Aouter un arc revient à :

- ajouter l'arc.
- Ajouter lien.

S*uppression :* Supprimer un arc revient à supprimer l'arc ainsi que toutes ses propriétés.

d) Règle de détection et solution de cohérence:

La règle ci-dessous représente une règle de vérification et de détection d'incohérence pour la propriété sémantique cardinalité du nœud *i* ayant une cardinalité égale à N. Suivi de la solution prédéfinie en cas d'incohérence.

Cardinalité (*nœud***i**, *cardinalité***N**)
Si (nombre de lien relié au nœud**i**) >**N**
Alors {Retourner incohérence}

Solution *:*

Si (dernière action faite = modification de la valeur de la propriété)
Alors {Modification de la valeur de la propriété avec une valeur (nombre de lien relié au nœud**i)**}
Sinon
{Supprimer les (nombre de lien relié au nœud**i – N)** derniers liens ajoutés}

La règle ci-dessous représente une règle de vérification et de détection d'incohérence pour l'invariant du nœud *i*. Suivi de la solution prédéfinie en cas d'incohérence.

Invariant (*nœud* **i)** *:*
Si nombre de lien relié au nœud **i** = 0
Alors {Retourner incohérence}
Solution *:*
Supprimer nœud i

3.5.2 Application de l'évolution:

Une fois le modèle d'évolution construit. L'évolution est appliquée comme suit :

- Transformé le modèle d'évolution en *Praxis [12]* en utilisant les *stratégies d'évolutions* spécifiées pour chaque *élément du modèle*.

- Le résultat de cette transformation, sous forme de liste d'action élémentaire, est appliqué sur le modèle *initiale* à fin de le faire évolué.
- Une copie du modèle initial est sauvegardée afin de pouvoir annuler l'évolution si le concepteur le souhaite.

Après évolution du modèle, un modèle modifié intermédiaire est obtenu. Ce dernier est utilisé dans la prochaine et dernière phase afin de vérifier la cohérence du modèle après évolution.

Exemple:

Ci-dessous un exemple d'opérations d'évolutions appliquées sur le modèle de la figure 3.7 comme suit :

1. Ajouter le Nœud **noeud$_4$** portant l'identificateur **Noe4**.
2. Ajouter le Nœud **noeud$_5$** portant l'identificateur **Noe5**.
3. Ajouter un lien via **arc$_3$** portant l'identificateur **ar3** entre **noeud$_4$** et **noued$_2$**.
4. Ajouter un lien via **arc$_4$** portant l'identificateur **ar4** entre **noeud$_3$** et **noeud$_5$**.
5. Modifier la propriété sémantique cardinalité du **noeud$_3$** *cardinalité* =1.2.

Ajouter noeud$_4$.	Create (Noe4, Noeud, 16)
	AddProprety (Noe4, Name, noeud$_4$, 17)
Ajouter noeud$_5$.	Create (Noe5, Noeud, 18)
	AddProprety (Noe5, Name, noeud$_5$, 19)
Ajouter arc3 entre noeud$_4$ et noued$_2$.	Create (ar3, Arc, 20)
	AddProprety (ar3, Name, arc$_3$, 21)
	AddProprety (ar3, lien, Noe4- Noe2,22)
Ajouter arc4 entre noeud$_3$ et noued$_5$.	Create (ar4, Noeud, 23)
	AddProprety (ar4, Name, arc$_4$, 24)
	AddProprety (ar4, lien, Noe3- Noe5, 25)
Modifier la propriété sémantique cardinalité du nœud$_3$ *cardinalité* =1.2.	RemoveProperty (Noe3, Cardinalité, 1.3, 24)
	AddProperty (Noe3, Cardinalité, 1.2, 25)

Figure 3.8: Modèle d'évolution en Praxis.

Ces opérations d'évolutions sont exprimées sous format de Praxis formant ainsi le modèle d'évolution en Praxis de la figure 3.8.

Le formalisme de Praxis [12] utilise uniquement les actions *Delete, Create, Add.* La modification d'une propriété sémantique est exprimée comme suit : suppression de la propriété avec l'ancienne valeur, suivie d'un ajout de la même propriété avec la nouvelle valeur. Après application du modèle d'évolution en Praxis, de la figure 3.8, sur le modèle initiale en Praxis, relatif au modèle de la figure 3.7, un modèle résultat intermédiaire est présenté dans la figure 3.9.

Create (Noe1, Noeud, 1)
AddProprety (Noe1, Name, $noeud_1$, 2)
AddProprety (Noe1, Cardinalité,1.1,3)
Create (Noe2, Noeud, 4)
AddProprety (Noe2, Name, $noeu_2$, 5)
AddProprety (Noe2, Cardinalité,1.1,6)
Create (Noe3, Noeud, 7)
AddProprety (Noe3, Name, $noeud_3$, 8)
AddProprety (Noe3, Cardinalité,1.2,9)
Create (ar1, Arc, 10)
AddProprety (ar1, Name, arc_1, 11)
AddProprety (ar1, lien, Noe3- Noe1, 12)
Create (ar2, Arc, 13)
AddProprety (ar2, Name, arc_2, 14)
AddProprety (ar1, lien, Noe3- Noe2, 15)
Create (Noe4, Noeud, 16)
AddProprety (Noe4, Name, $noeud_4$, 17)
Create (Noe5, Noeud, 18)
AddProprety (Noe5, Name, $noeud_5$, 19)
Create (ar3, Arc, 20)
AddProprety (ar3, Name, arc_3, 21)
AddProprety (ar3, lien, Noe4- Noe2,22)
Create (ar4, Arc, 23)
AddProprety (ar4, Name, arc_4, 24)
AddProprety (ar4, lien, Noe3- Noe5, 25)

Figure 3.9: Modèle intermédiaire

3.5.3 Vérification de cohérence :

Dans cette étape la vérification du modèle intermédiaire est lancée comme suit :

- Vérification des invariants.
- Vérification des propriétés sémantiques.
- Vérification des influences entre propriétés sémantiques.

Une fois que toutes ces vérifications effectuées en utilisant les règles de détections d'incohérences. Deux cas peuvent se présentés :

- Aucune incohérence n'est trouvée, alors l'évolution est validé et le modèle mis à jour.
- Des incohérences sont détectées. Une liste d'incohérence est alors présentée au concepteur.

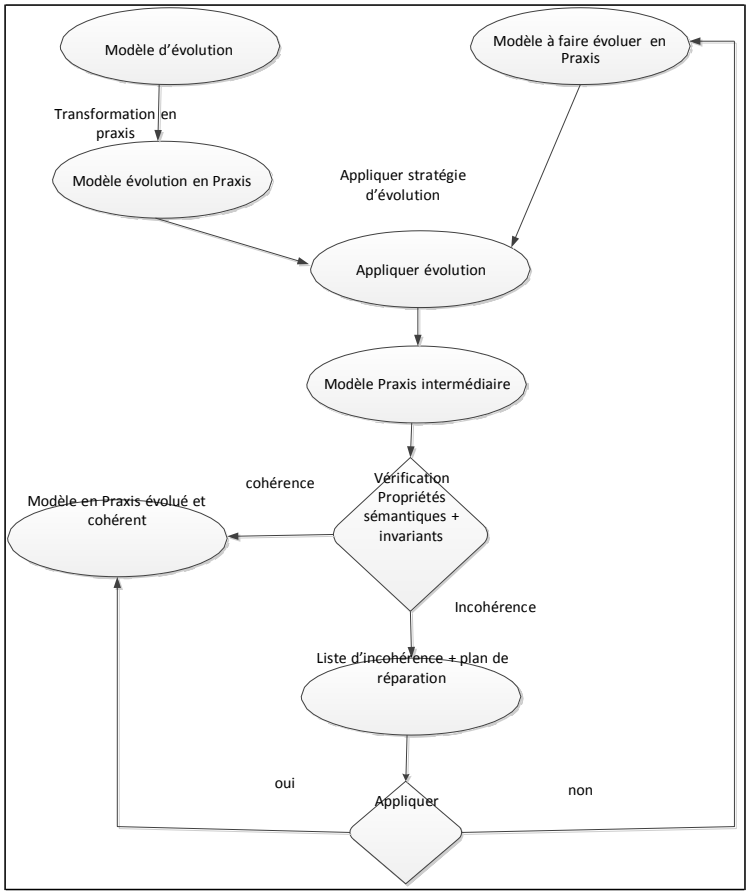


Figure 3.10: Vérification de la cohérence

Exemple :

En utilisant le modèle intermédiaire de la figure 3.9, ainsi que la règle de détection d'incohérence de la propriété cardinalité. La vérification détecte les incohérences suivantes :

- La propriété sémantique cardinalité du *nœud2* n'est pas respectée. Le *nœud2* est relié au *nœud3* et au *nœud4* avec une *cardinalité 1.1*.
- La propriété sémantique cardinalité du *nœud3* n'est pas respectée. Le *nœud3* est relié au *nœud1, nœud2* et au *nœud5* avec une *cardinalité 1.2*.

La liste d'action suivante source de la détection de ces incohérences est présentée au concepteur :

AddProprety (Noe2, Cardinalité,1.1, 6).
AddProprety (Noe3, Cardinalité,1.2, 9)

3.5.4 Proposition du plan de réparation :

En utilisant les incohérences trouvées dans l'étape précédente, ainsi que les règles de solution pour chaque incohérence spécifiée dans la première phase. Une liste d'action de correction est proposée au concepteur appelé plan de réparation. Ce dernier peut être appliqué ou ignoré par le concepteur.

Si le concepteur décide de ne pas appliquer le plan de réparation, alors le modèle initial sauvegardé dans la troisième phase est retourné. Sinon ce plan de réparation est appliqué automatiquement sur le modèle intermédiaire de la troisième phase afin d'obtenir un nouveau modèle évolué et cohérent.

Ce modèle évolué est cohérent sous forme de Praxis est transformé vers le format initial.

Exemple :

En utilisant le modèle d'évolution de la figure 3.8, ainsi que la règle de solution d'incohérence de la propriété cardinalité. Les actions du modèle d'évolution sont parcourues du plus ancienne au plus récente.

- Pour la première incohérence, la propriété n'étant pas modifiée, la solution est la suppression du dernier lien ajouté au *nœud2* ***AddProprety*** *(ar3, lien, Noe4- Noe2 ,22)* exprimé par les actions suivantes :

RemoveProperty (*ar3, lien,Noe4-Noe2, 27).*
RemoveProperty (*ar3, Name,arc3 , 28).*
Delete (ar3,Arc, 29).

- Pour la deuxième incohérence, la propriété étant modifiée, la solution est la modification de la propriété avec la valeur 3.exprimé par les actions suivantes :

RemoveProperty (Noe3, Cardinalité, 1.2, 30)
AddProperty (Noe3, Cardinalité, 1.3, 31)

- Une vérification de l'invariant est lancé suite à ces actions de correction. Après suppression du lien entre le *noeud2* et le *nœud4*, ce dernier n'étant relié à aucun autre nœud doit être supprimé, en utilisant la règle de l'invariant de la section 3.5.1 exprimée par les actions suivantes :

RemoveProperty (*Noe4, Name,noeud4 ,32).*
Delete (Noe4,Noeud, 33).

Le plan de réparation suivant est proposé au concepteur :

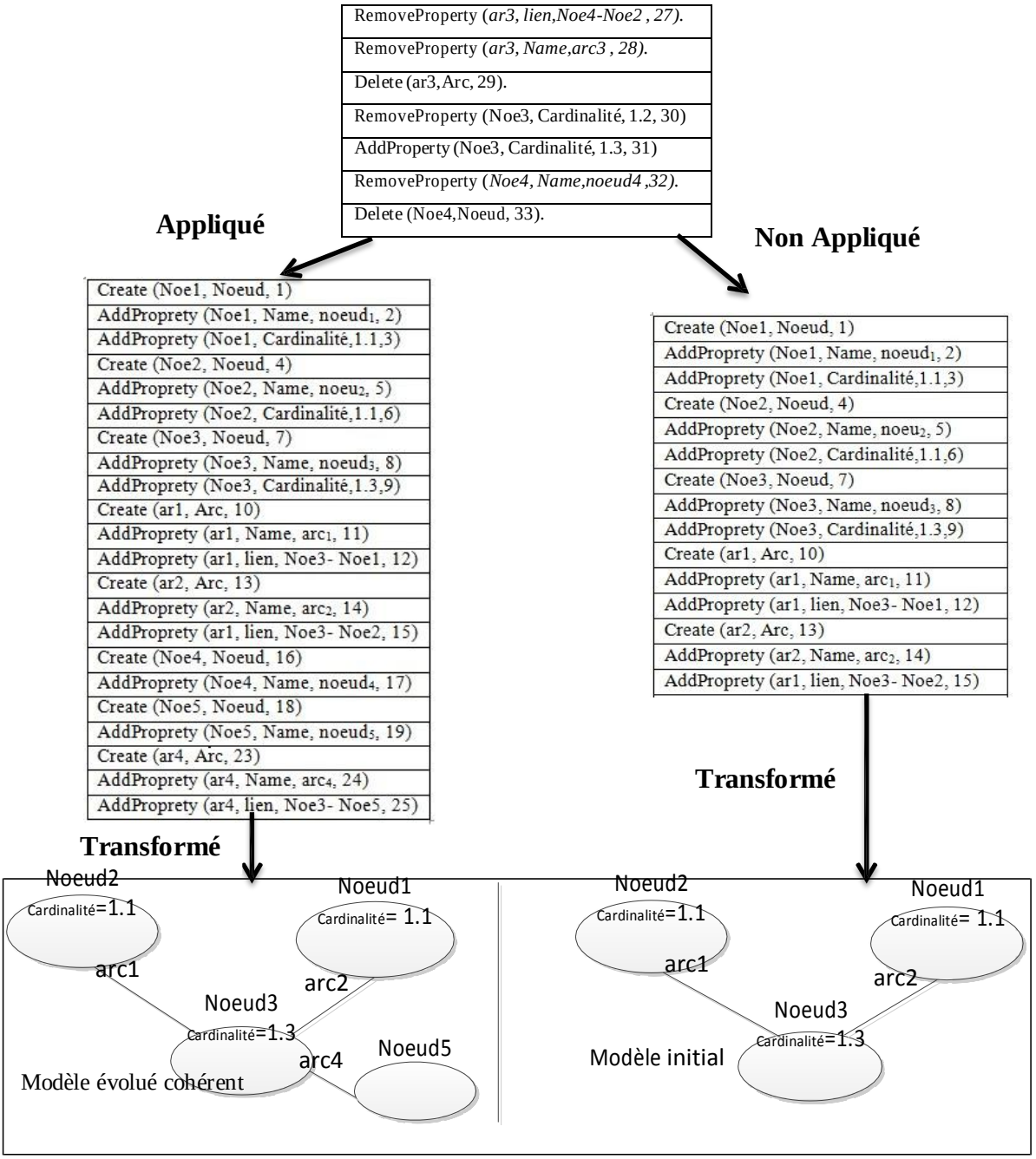


Figure 3.11: Exemple de plan de réparation

3.6 Bilan :

Après avoir présenté le modèle IMoSCM, nous avons constaté que le modèle IMoSCM répond à la majorité des limites des travaux étudiés dans le chapitre précèdent:

1. IMoSCM traite la cohérence sémantique indépendamment de tout méta modèle. Contrairement aux différents travaux étudiés qui peuvent traiter différents types de cohérences indépendamment de tout méta modèle mais pas la cohérence sémantique, dans le cas où la cohérence sémantique est traitée, la solution proposée est dépendante d'un certain méta modèle.
2. IMoSCM propose une correction automatique des incohérences via le plan de réparation. Contrairement aux différents travaux étudiés qui proposent une correction manuelle des incohérences et ne proposent aucun plan de réparation à l'exception du modèle « Praxis [12]».
3. Dans les différents travaux étudiés l'évolution dynamique n'est abordée que par quelques ADL. IMoSCM n'aborde pas l'évolution dynamique. Ce point est laissé en perspective.

Le modèle IMoSCM répond aux objectifs fixés dans la section 2.7.3 du chapitre précèdent:

1. Transformer le modèle à faire évoluer en formalisme de praxis [12]. En utilisant ce formalisme le modèle IMoSCM peut être utilisé sur n'importe quel modèle en donnant la possibilité au concepteur d'introduire les nouveaux paramètres (propriétés sémantiques, invariants…etc.) propres au modèle qu'il souhaite faire évoluer. Ce qui le rend **indépendant de tout méta modèle.**
2. Détecter les incohérences sémantiques après évolution en utilisant les propriétés sémantiques proposées dans SAEV [9], les définir en utilisant les règles de détection d'incohérence sémantique. Ce qui permet à notre modèle de **traiter la cohérence sémantique.**
3. **Proposer un plan de réparation** en utilisant les règles de solution de cohérence.
4. Gérer **l'impact de l'évolution** en utilisant les règles de détection d'incohérence sémantique.
5. **Correction automatique** de l'incohérence en utilisant le plan de réparation.

3.7 Conclusion :

Nous avons présenté dans ce chapitre notre modèle IMoSCM, comme solution à la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs métas modèles, en assurant une cohérence sémantique des modèles. Dans IMoSCM le modèle sous format de Praxis [12] représente toute architecture logicielle (*orientée composant, orientée objet*) susceptible d'évoluer. Nous avons introduit ce concept pour pouvoir modéliser toutes architectures logicielles, sous forme d'action élémentaires en se basant sur le formalisme de Praxis [12]. Ce qui permet une gestion d'évolution indépendamment de tout méta modèle. Dans IMoSCM des règles de détection d'incohérence sémantique sont prédéfinies, afin de permettre la détection des incohérences sémantiques après évolution en utilisant les propriétés sémantiques proposées dans SAEV [9]. Dans le but de corriger les incohérences trouvées, IMoSCM utilise des règles de solution d'incohérence prédéfinies et propose une suite d'action élémentaire sous forme de Praxis [12], comme un plan de réparation, à partir de la liste des incohérences trouvées, ainsi qu'une correction automatique des incohérences.

Chapitre 4 :
Modélisation UML du modèle d'évolution

4.1 Introduction :

Dans le chapitre précédent nous avons présenté, IMoSCM, un modèle de gestion de la cohérence sémantique lors de l'évolution des modèles appliquée aux ADL. Nous avons décrit ces concepts ainsi que son mécanisme opératoire. Dans ce chapitre nous nous intéressons à la conception fonctionnelle afin d'implémenter le modèle IMoSCM.

4.2 Besoin fonctionnel:

Les diagrammes de cas d'utilisations représentent les cas d'utilisation, les acteurs et les relations entre les cas d'utilisations et les acteurs. Les cas d'utilisations permettent de structurer et d'articuler les besoins en fonctionnalités et de définir la manière dont les utilisateurs voudraient interagir avec le système.

4.2.1 Les acteurs :

Utilisateur : il introduit le modèle à faire évoluer ainsi qu'une suite d'opération d'évolution. Applique les opérations d'évolutions et lance la vérification de la cohérence.

Spécialiste modélisation : définit les stratégies d'évolution, les propriétés sémantiques ainsi que les invariants. Elabore le langage d'évolution du modèle à faire évoluer et assure la transformation du modèle à faire évoluer vers praxis [12] et inversement.

4.2.2 Détermination des cas d'utilisation :

Nous présentons dans ce qui suit le diagramme de cas d'utilisation général. Ce diagramme sera détaillé pour chaque étape du processus du modèle IMoSCM : Construction du modèle d'évolution, l'application de l'évolution et la vérification de la cohérence & application du plan de réparation. La figure 4.1 présente le diagramme de cas d'utilisation général du modèle.

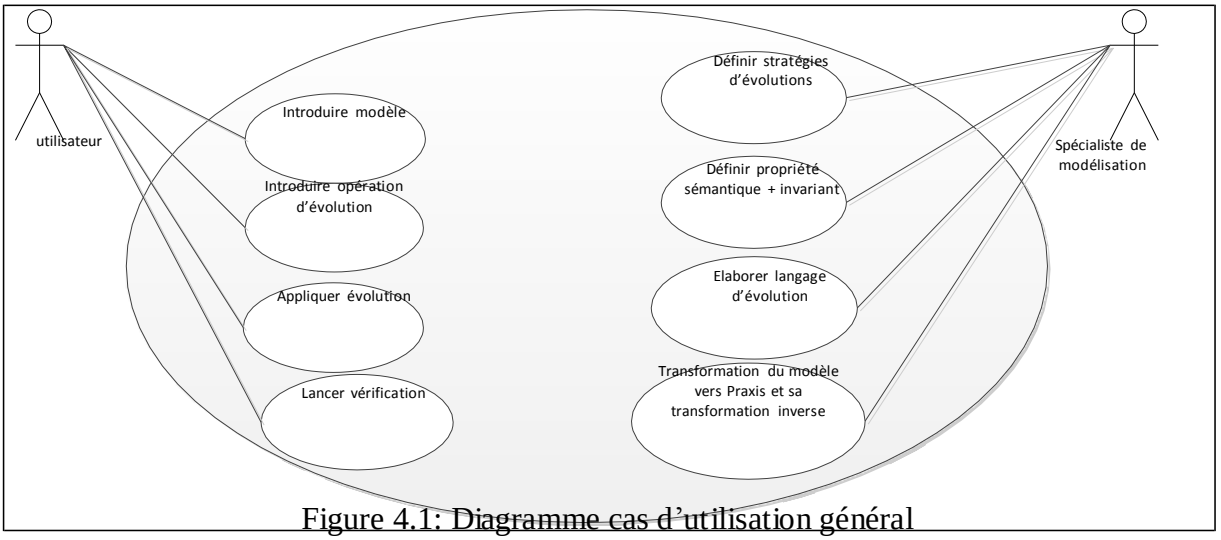


Figure 4.1: Diagramme cas d'utilisation général

4.2.2.1 Construction et application du modèle d'évolution :

Construction du modèle d'évolution :

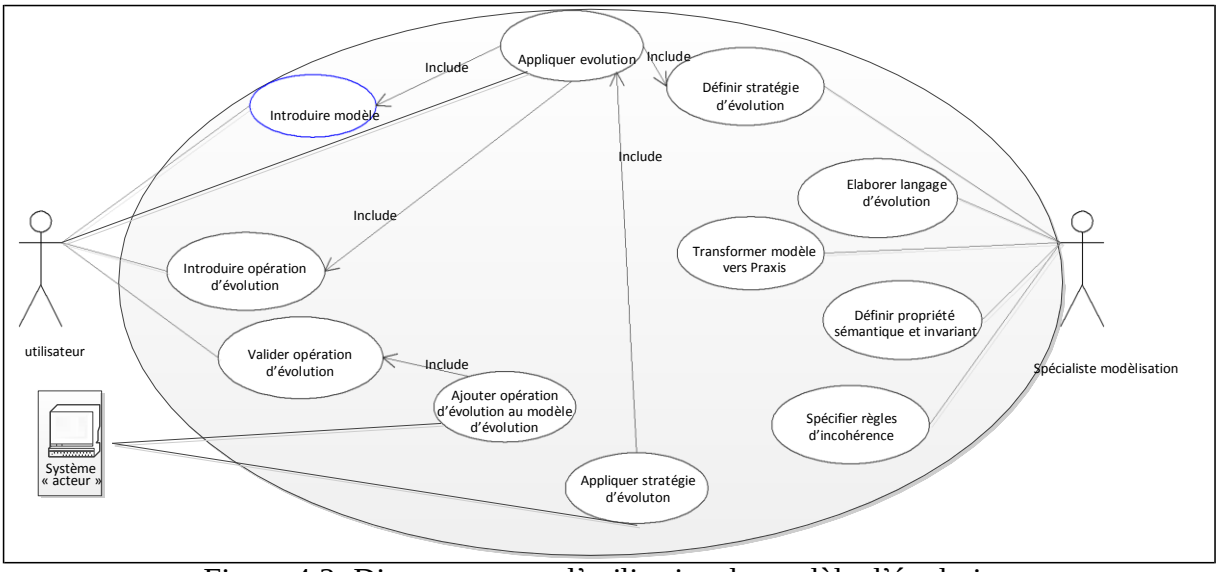


Figure 4.2: Diagramme cas d'utilisation du modèle d'évolution

L'utilisateur doit introduire le modèle à faire évoluer afin de lancer la vérification de cohérence. Le spécialiste modélisation élabore le modèle dévolution correspondant, définit les stratégies d'évolution ainsi que les propriétés sémantiques et les invariants et spécifie les règles d'incohérences.

Diagramme de séquence :

Le spécialiste de modélisation commence par élaborer le langage d'évolution, Définit les stratégies d'évolution ainsi que les propriétés sémantiques et les invariants et spécifie les règles d'incohérences. L'utilisateur introduit un modèle x à faire évoluer ainsi que les

opérations d'évolutions. Chaque opération choisie, peut être annulée ou validée par l'utilisateur. Dans le cas de validation l'opération est ajoutée au modèle d'évolution. Sinon elle sera ignorée.

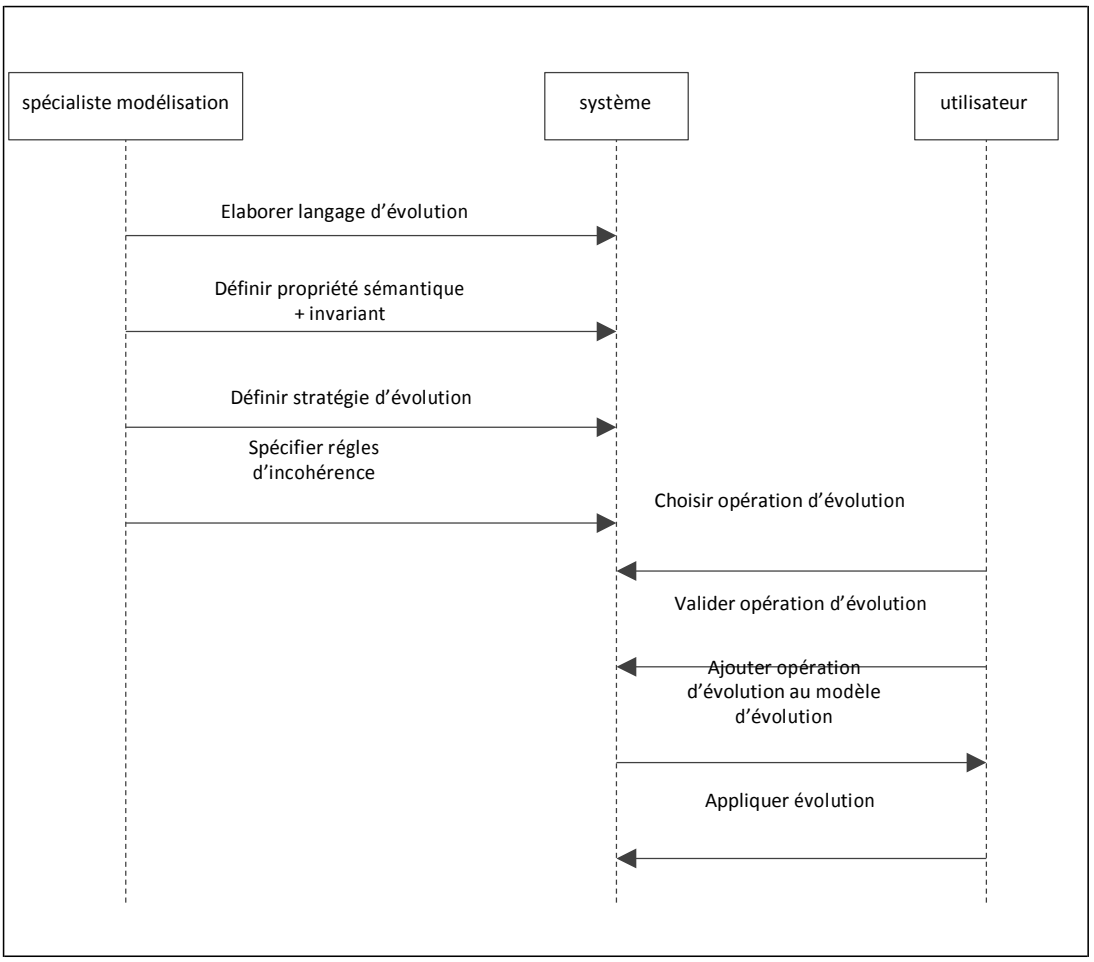


Figure 4.3 Diagramme de séquence construction et application du modèle d'évolution

La figure 4.4 présente le diagramme de classe des propriétés sémantiques où figurent uniquement les deux propriétés Exclusivité-Partage et Cardinalité-Cardinalité inverse. Le spécialiste de modélisation définit les méthodes pour les règles de cohérences. D'autres propriétés sémantiques peuvent être prises en compte en ajoutant les sous classes correspondantes et en implémentant les méthodes de vérification et de détection de cohérence pour chaque propriété sémantique.

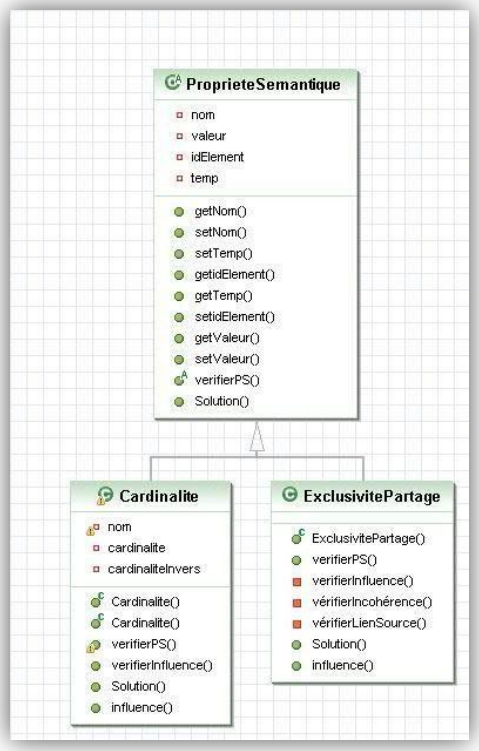


Figure 4.4: Diagramme de classe des propriétés sémantiques

Appliquer évolution :

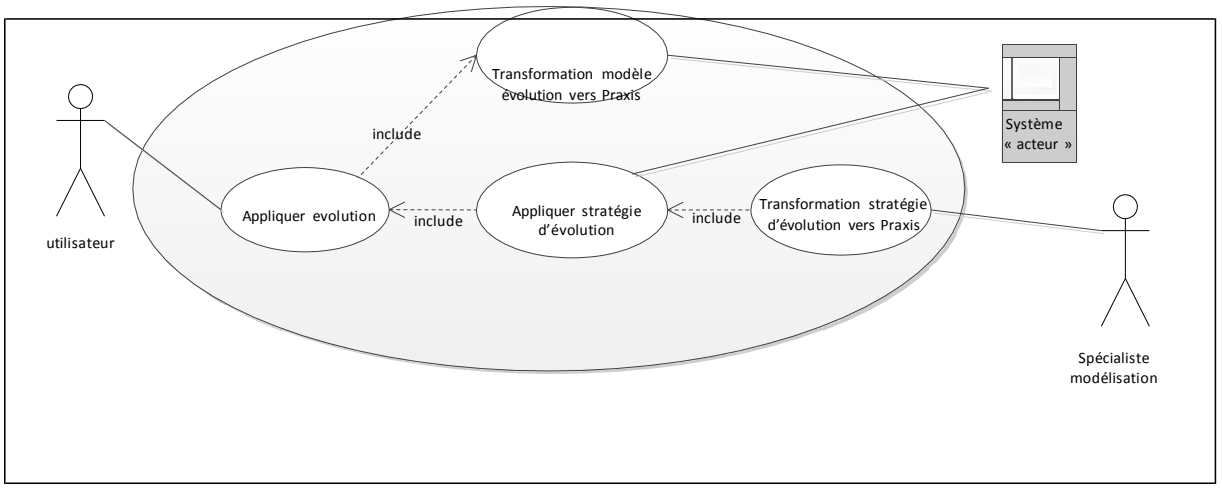


Figure 4.5: Diagramme de cas d'utilisation d'application de l'évolution

Une fois que le spécialiste modélisation élabore la transformation des stratégies d'évolution ainsi que le modèle d'évolution en Praxis [12], l'utilisateur peut appliquer le modèle d'évolution c.-à-d. les opérations validées auparavant. Ainsi le système applique les stratégies dévolutions correspondantes.

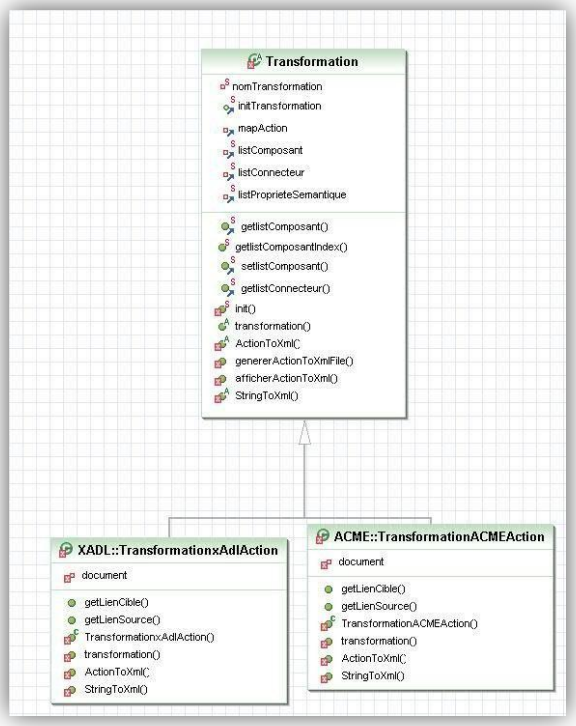


Figure 4.6: Diagramme de classe de transformation vers Praxis et la transformation inverse.

Afin d'appliquer le modèle d'évolution sur le modèle à faire évoluer, ces deux modèles doivent être transformés en Praxis. La figure 4.7 présente le diagramme de classe pour la transformation vers Praxis et la transformation inverse pour deux métas modèles différents xADL et ACME. Un spécialiste de modélisation peut ajouter la sous classe correspondante à la transformation de son méta modèle.

La figure 4.7 présente le diagramme de classe des actions élémentaires de Praxis utilisées dans la transformation des modèles en formalisme de Praxis.

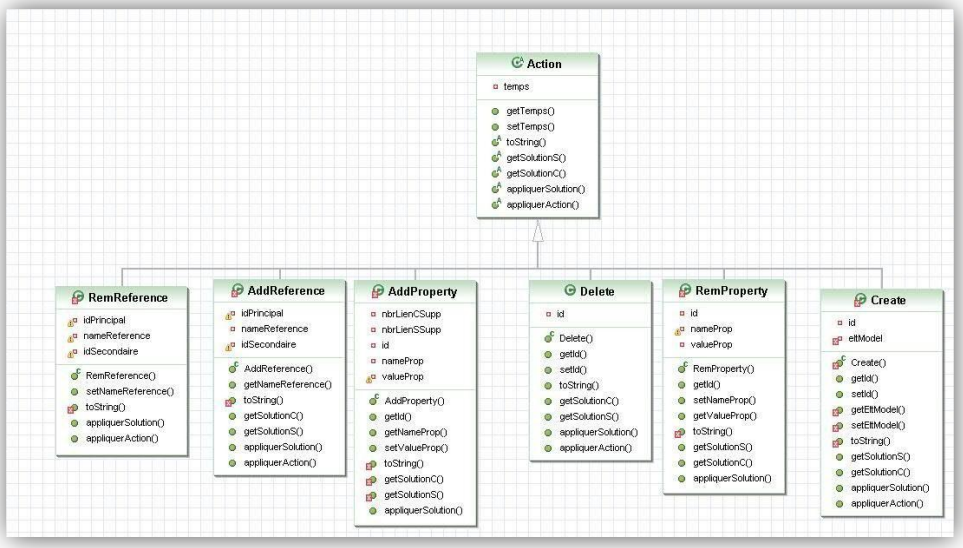


Figure 4.7: Diagramme de classe d'action de Praxis

Diagramme de séquence :

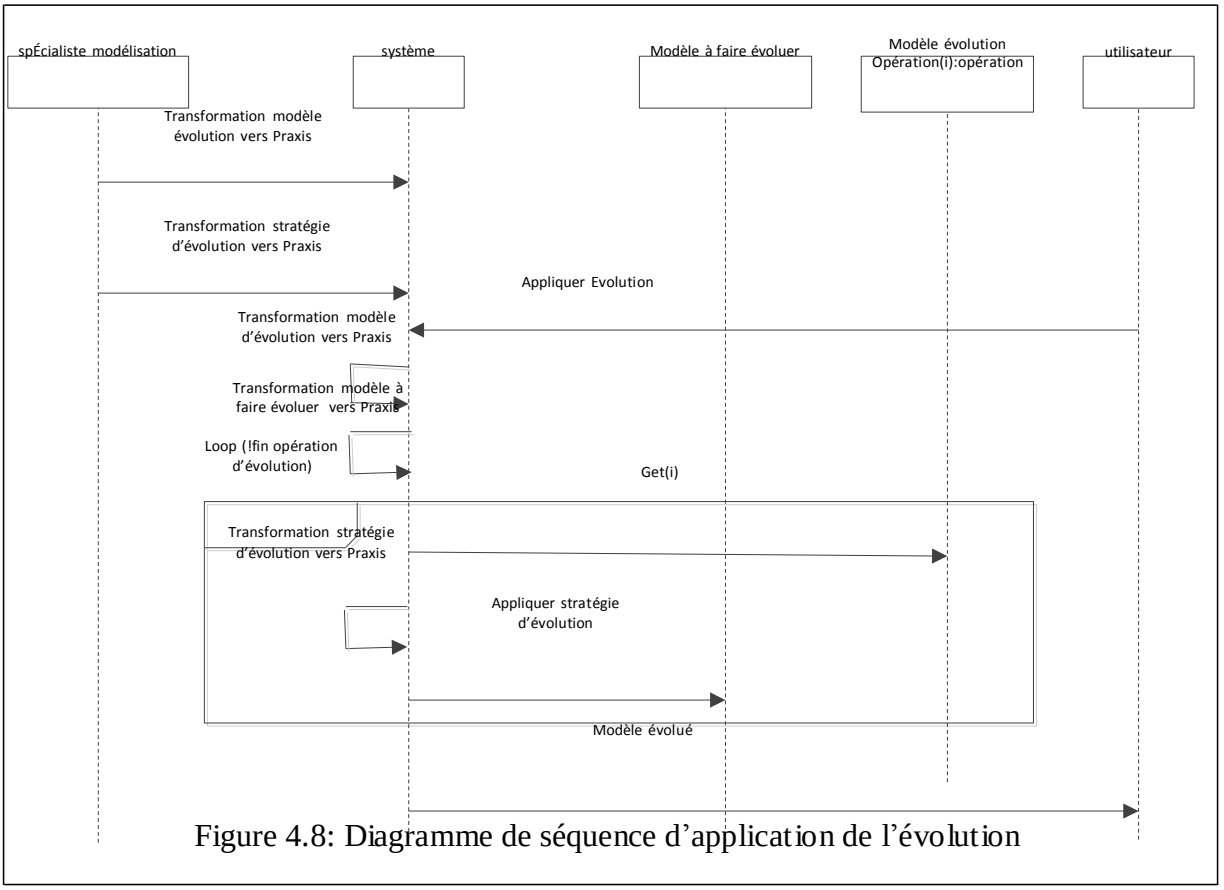


Figure 4.8: Diagramme de séquence d'application de l'évolution

L'utilisateur choisit d'appliquer son modèle d'évolution, ce dernier est à son tour transformé en Praxis [12] afin qu'il puisse être appliqué au modèle à faire évolué. Ce modèle d'évolution est le *Delta modèle* obtenu à partir des modifications opérées sur le modèle initial. Les stratégies d'évolutions correspondantes aux opérations d'évolutions présentées dans le modèle d'évolution seront transformées vers Praxis pour être appliquées sur le modèle à faire évoluer. Un modèle évolué est à la fin présenté à l'utilisateur.

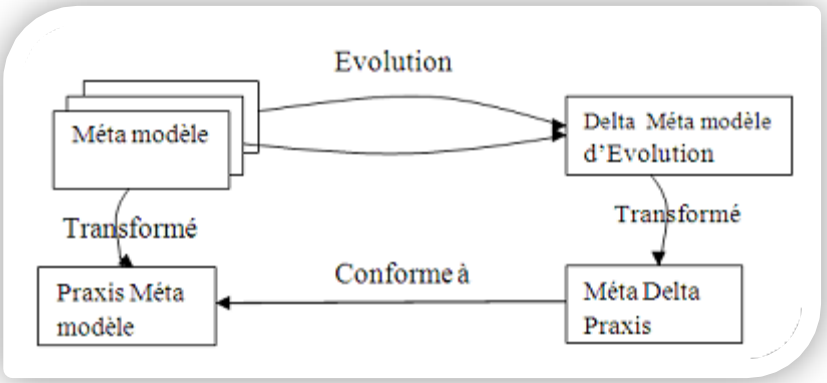


Figure 4.9: Transformation du modèle d'évolution vers Praxis

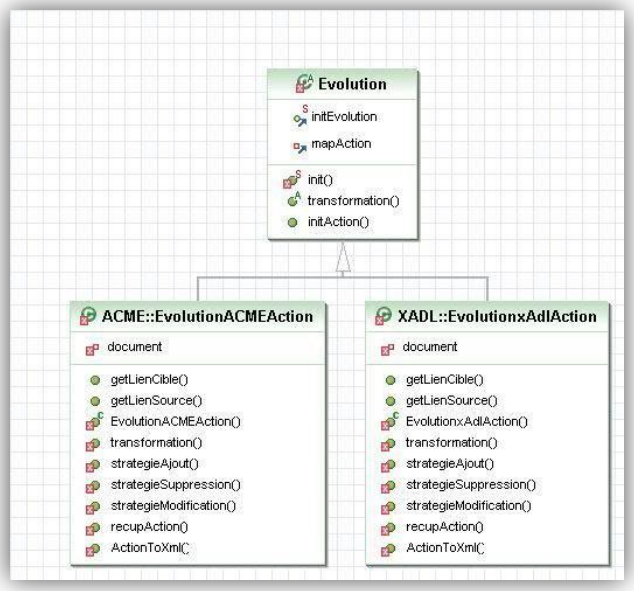


Figure 4.10: Diagramme de classe d'évolution

La figure 4.10 présente le diagramme de classe d'évolution pour les deux métas modèles ACME et xADL.

Un spécialiste de modélisation peut ajouter la sous classe correspondante à l'évolution de son méta modèle.

4.2.2.2 Vérification de la cohérence et application du plan de réparation :

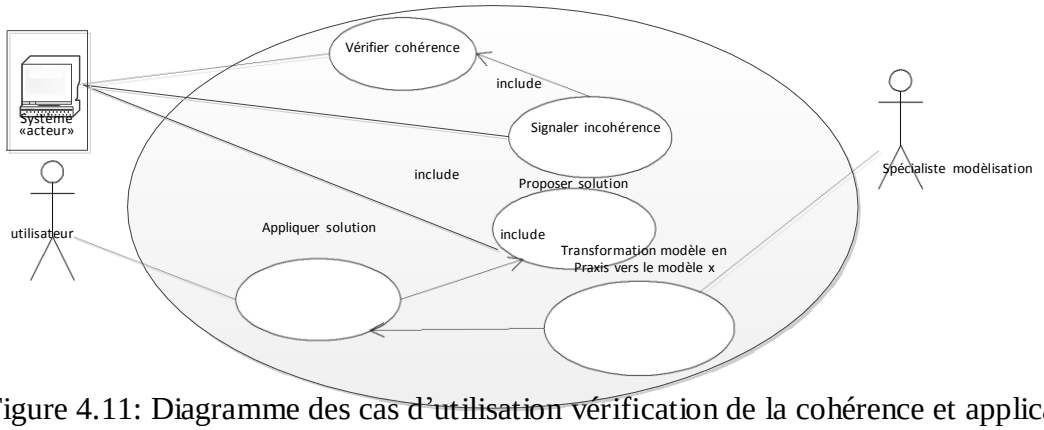


Figure 4.11: Diagramme des cas d'utilisation vérification de la cohérence et application du plan de réparation

Une fois le modèle d'évolution appliqué, le système lance une vérification de la cohérence du nouveau modèle intermédiaire. Les incohérences trouvées sont signalées à l'utilisateur. Un plan de réparation est proposé pour les incohérences trouvées. L'utilisateur peut appliquer la solution afin d'avoir un modèle Praxis évolué et cohérent.

Diagramme de séquence :

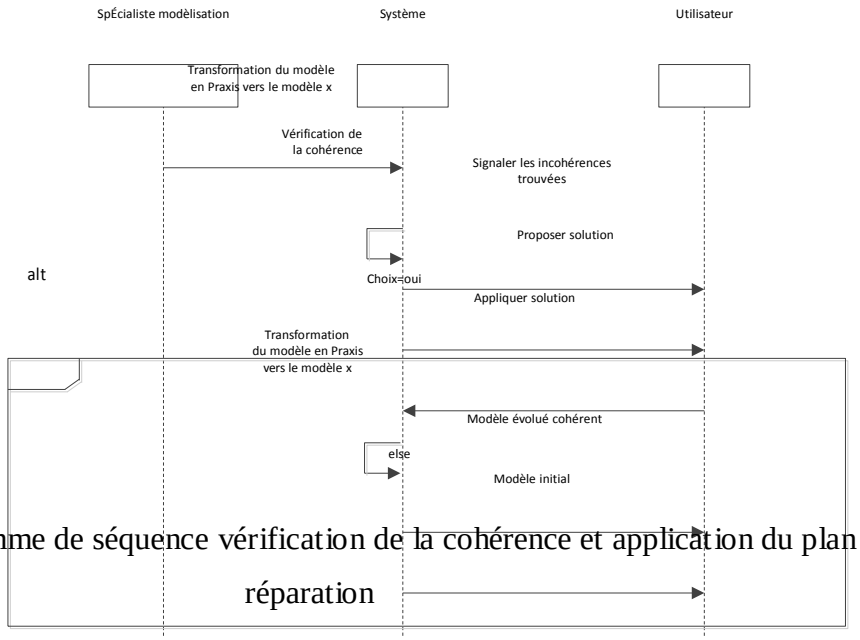


Figure 4.12: Diagramme de séquence vérification de la cohérence et application du plan de réparation

Le système vérifie la cohérence du modèle intermédiaire puis propose à l'utilisateur la liste des incohérences trouvées ainsi qu'une proposition de solution à chacune d'elle. L'utilisateur peut choisir d'appliquer la solution afin d'avoir un modèle Praxis évolué et cohérent, ou bien ignorer la solution et retourner au modèle Praxis initial.

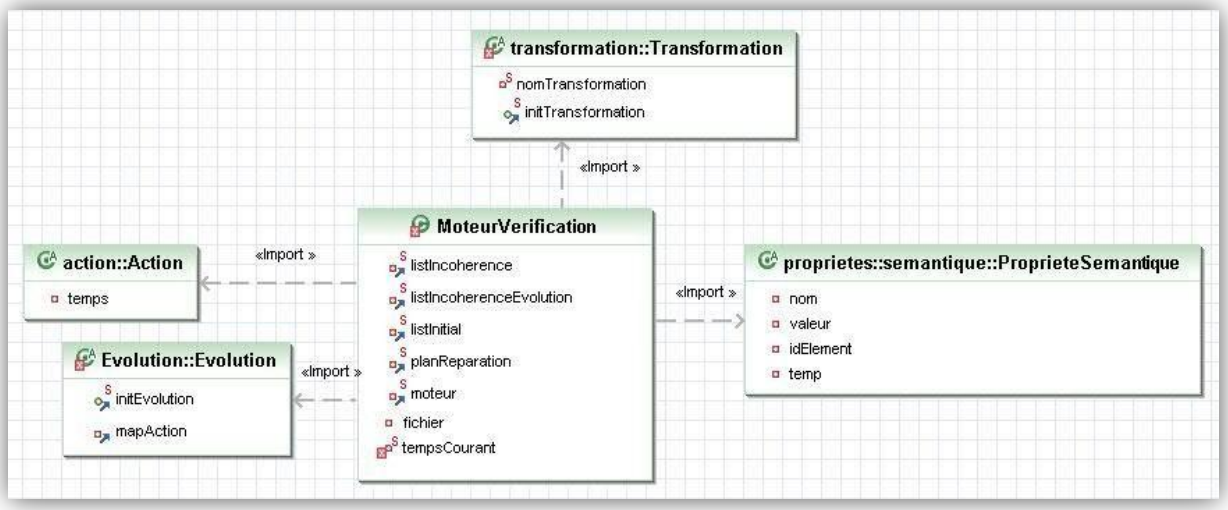


Figure 4.13: Diagramme de classe du moteur de vérification de cohérence et ses dépendances

De même après évolution le modèle résultant en Praxis est transformé sous la forme de départ via la transformation inverse T^{-1} comme montrer dans la figure 4.14.

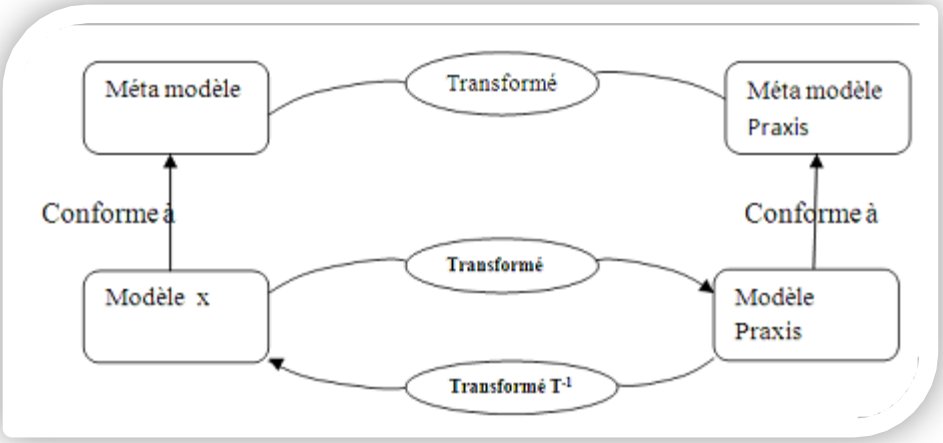


Figure 4.14: Transformation du modèle à faire évoluer et sa transformation inverse

4.3 Diagramme de cas d'utilisation globale :

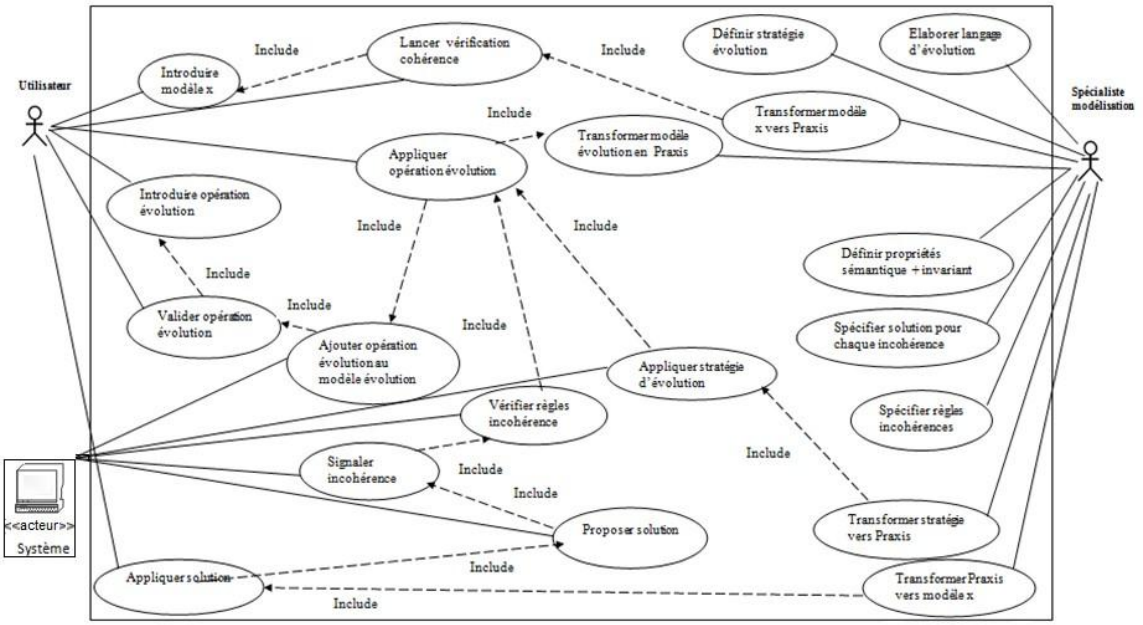


Figure 4.15: Diagramme cas d'utilisation globale

L'utilisateur doit introduire le modèle à faire évoluer. Le spécialiste modélisation élabore le modèle dévolution correspondant, définit les stratégies d'évolution ainsi que les propriétés sémantiques et les invariants et spécifie les règles de détection d'incohérences.

Chaque opération choisie, peut être annulée ou validée par l'utilisateur. Dans le cas de validation l'opération est ajoutée au modèle d'évolution. Sinon elle sera ignorée. Une fois que le spécialiste modélisation élabore la transformation des stratégies d'évolution ainsi que le modèle d'évolution en Praxis [12], l'utilisateur peut appliquer le modèle d'évolution c.-à-d. les opérations validées auparavant. Ainsi les stratégies dévolutions correspondantes sont appliquées. Ces stratégies d'évolution présentées dans le modèle d'évolution seront transformées vers Praxis pour être appliquées sur le modèle à faire évoluer. Une fois le modèle d'évolution appliquée, la vérification de la cohérence du nouveau modèle intermédiaire est lancée. Les incohérences trouvées sont signalées à l'utilisateur. Un plan de réparation est proposé pour les incohérences trouvées. L'utilisateur peut appliquer la solution afin d'avoir un modèle Praxis évolué et cohérent.

4.4 Diagramme de Package :

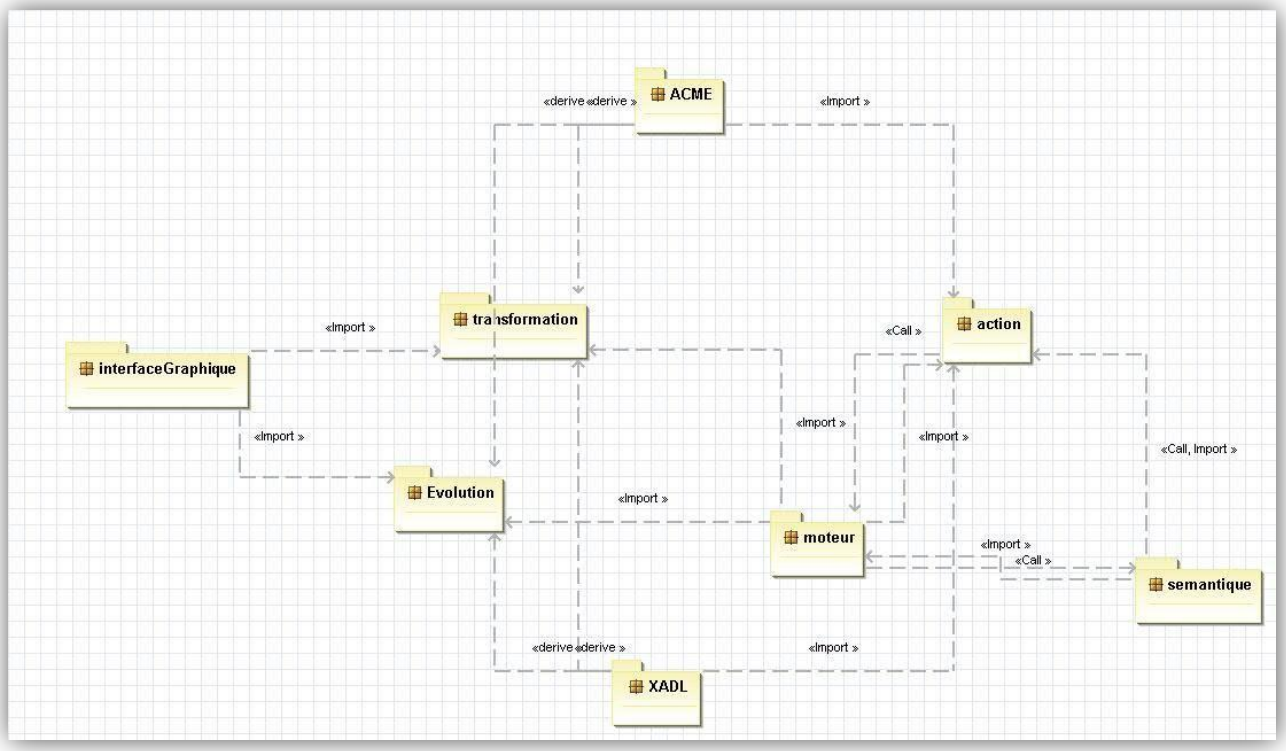


Figure 4.16: Diagramme de Package

4.5 Conclusion :

Dans ce chapitre nous avons présenté une conception fonctionnelle du modèle IMoSCM. Nous avons identifié deux acteurs : **l'utilisateur** qui introduit le modèle à faire évoluer ainsi qu'une suite d'opération d'évolution, applique les opérations d'évolutions et lance la vérification de la cohérence. Le s**pécialiste modélisation** qui définit les stratégies d'évolutions, les propriétés sémantiques ainsi que les invariants, élabore le langage d'évolution du modèle à faire évoluer et assure la transformation du modèle à faire évoluer vers praxis et inversement. Nous avons présenté le diagramme de classe ainsi que le diagramme de cas d'utilisation pour chaque étape du processus du modèle IMoSCM : l'application de l'évolution, la vérification de la cohérence & application du plan de réparation. Les détails d'implémentation ainsi que la validation du modèle IMoSCM sont présentés dans le chapitre suivant.

Chapitre 5 :
Validation

5 .1 Présentation généralé dé la maniéré dé validér :

5.1.1 Introduction :

Ce chapitre présente notre travail de validation, qui consiste à valider notre solution pour deux métas modèles différents, afin de prouver que notre solution est indépendante de tout méta modèle. Pour cela, l'implémentation est décomposée en deux parties: une partie indépendante du méta modèle et une partie dépendante du méta modèle.

Pour la partie dépendante, deux implémentations ont été effectuées pour les méta modèle xADL[49 ,51] et ACME[36] en utilisant le langage JAVA[50]. Cette partie doit être réalisée par un spécialiste de modélisation qui doit élaborer un *langage d'évolution* du modèle initiale à faire évoluer ainsi que les stratégies d'évolutions, assurer la transformation du modèle d'évolution vers praxis [12] et le modèle initial vers praxis et inversement. Une fois réalisés un utilisateur introduit le modèle à faire évoluer ainsi qu'une suite d'opération d'évolution, applique les opérations d'évolution et lance la vérification de la cohérence. L'application des opérations d'évolutions et la vérification de la cohérence représentent la partie indépendante du méta modèle.

5.1.2 Démarche à suivre:

Dans ce qui suit une démarche à suivre pour l'implémentation des deux parties, dépendante et indépendante du méta modèle, est présentée et détaillée dans les sections suivantes.

5.1.2.1 Construction modèle évolution :

Dans le but de faire évoluer un modèle, une interface graphique est implémentée pour l'utilisateur. En se basant sur les travaux sur la gestion des versions et la représentation de différences entre deux modèles [44] dans la section 2.5.1.1 du chapitre II, trois types d'opération par élément sont possibles via cette interface: *l'ajout, la suppression et la modification* voir la figure 5.1.

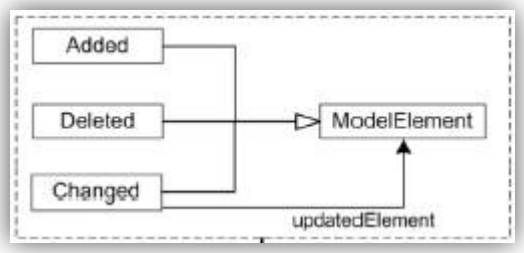


Figure 5.1: représentation de différence [44]

Le spécialiste du méta modèle définit une grammaire en DTD à fin de représenter les opérations d'évolutions par élément, présentées dans la figure 5.1, introduites et validées par l'utilisateur à travers l'interface graphique. Une fois les différents types d'opérations validées, un modèle se construit au fur et à mesure appelé *modèle d'évolution* comme montrer dans la figure 5.2.

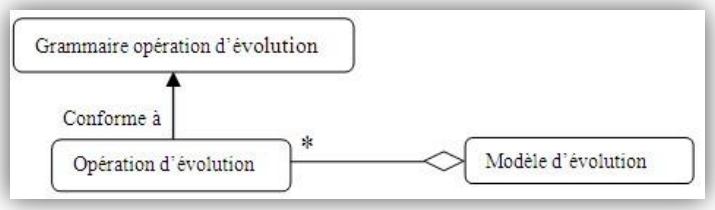


Figure 5.2: Construction modèle d'évolution

Ce modèle d'évolution est donc constitué d'un ensemble d'opération d'évolution conformément à la grammaire définie par le spécialiste de modélisation.

5.1.2.2 Élaboration des stratégies d'évolution :

Une stratégie d'évolution permet de spécifier l'évolution d'un élément du modèle. Pour chaque opération d'évolution appliquée sur un élément du modèle, correspond une stratégie d'évolution constituée d'un ensemble d'actions à exécuter, voir figure 5.3. Comme exemple, la suppression d'un composant revient à: premièrement la suppression des liens vers les interfaces de ce composant, deuxièmement la suppression des interfaces de ce composant et troisièmement la suppression du composant lui-même. Ces trois actions constituent la **stratégie de suppression** d'un composant.

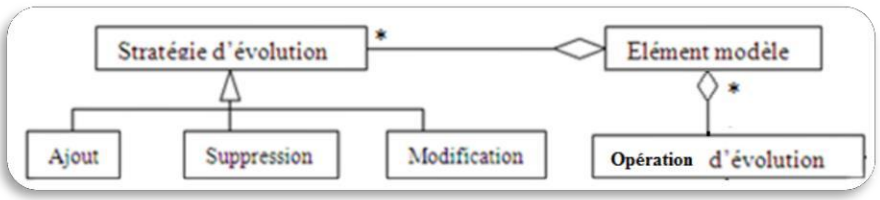


Figure 5.3: Stratégie d'évolution

5.1.2.3 Transformation modèle vers Praxis et sa transformation inverse:

Cette transformation, **T** sur la figure 5.4, assure l'obtention d'une séquence d'action en Praxis [12] du modèle correspondant à fin d'assurer l'indépendance par rapport au méta modèle. Il peut s'agir d'une séquence d'action correspondant à l'état actuel du modèle, ou bien une séquence d'action correspondant aux dernières modifications apportées par l'utilisateur c.-à-d. le modèle d'évolution. La transformation inverse, $\mathbf{T^{-1}}$ sur la figure 5.4, assure l'obtention du modèle à faire évoluer, avant ou après son évolution, sous son format d'origine à partir de la séquence d'action de Praxis [12] correspondante.

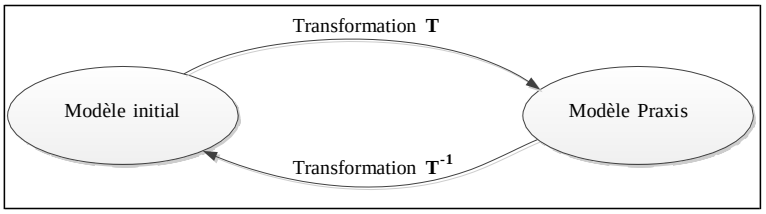


Figure 5.4: Transformation vers Praxis et la transformation inverse.

5.1.2.4 Moteur d'évolution et de vérification de cohérence :

Ce moteur assure l'application des opérations d'évolutions, vérifie la cohérence du modèle résultat et propose un plan de réparation en cas d'incohérence comme montrer dans la figure 5.5

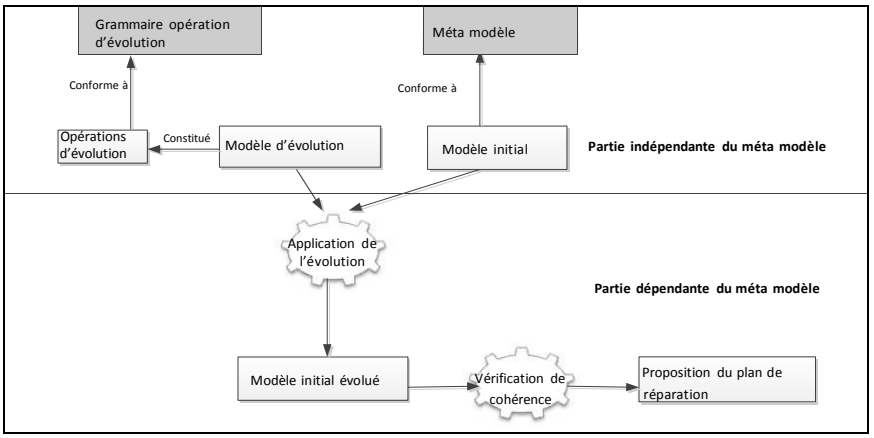


Figure 5.5: Moteur d'évolution et de vérification de cohérence.

5.2 Partié indépéndanté du méta modélé:

5.2.1 Moteur d'évolution et de vérification de cohérence :

A cette étape le modèle d'évolution ainsi que le modèle initial sont sous format Praxis [12]. De ce fait le moteur d'évolution et de vérification de cohérence est indépendant de tout méta modèle.

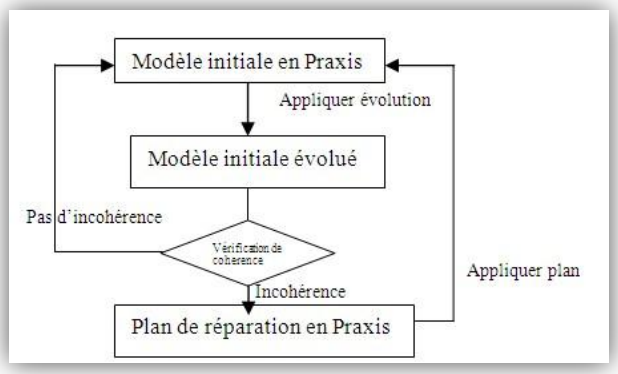


Figure 5.6: Schéma générale

Des actions d'évolutions sont appliquées sur le modèle initial en utilisant les stratégies d'évolutions. Après évolution la cohérence du modèle est vérifiée en respectant les invariants et propriétés sémantiques, dans le cas où les opérations provoquent des incohérences, un plan de réparation est proposé afin de corriger les incohérences et retourner à l'état cohérent. Cette partie se déroule en deux étapes : Application de l'évolution, vérification de cohérence et proposition du plan de réparation.

5.2.1.1 Application de l'évolution :

Après avoir appliqué les stratégies d'évolution et transformé les opérations d'évolutions présentées dans le modèle d'évolution en une suite d'action du formalisme de Praxis, cette dernière sera appliquée sur la séquence d'action en *PRAXIS* [12] initiale, afin de faire évoluer le modèle, en obtenant une nouvelle séquence d'action représentant le modèle après évolution.

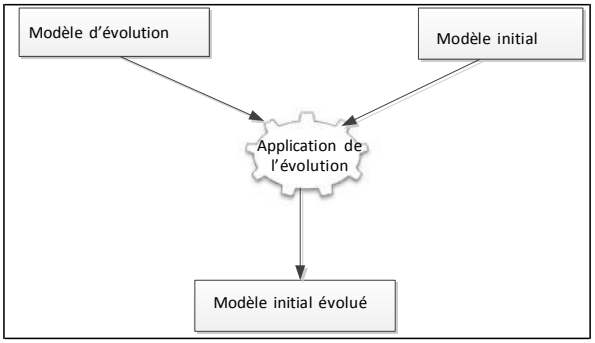


Figure 5.7: Application de l'évolution

Action d'évolution	**Actions exécutées**
AddProperty (idElement,nameProperty,valeurP,temps)	Ajouter à la liste initiale en incrémentant le temps d'exécution l'action : **AddProperty** (idelement,nameProperty,valeurP,temps)
RemProperty (idélémentr,nameProperty,valeurP,temps)	Supprimer de la liste initiale l'action : **AddProperty** (idelément,nameProperty,valeurP,temps)
Create(idelement,Typeelement,temps)	Ajouter à la liste initiale en incrémentant le temps d'exécution l'action : **Create**(idelement,Typeelement,temps)
Delete(idelement,Typeelement,temps)	Supprimer de la liste initiale l'action : **Create**(idelement,Typeelement,temps)

Tableau 5.1: Règles d'application des actions d'évolution

Exemple :

Dans l'exemple présenté dans le listing 5.3, l'action numéro 1) de la liste d'action d'évolution en praxis revient à supprimer l'action numéro 11) de la liste d'action initiale du listing 5.4: `AddProperty(co1,LinkCi,ISClient1.int1,11)`. L'action numéro 2) revient à la suppression de l'action numéro 3) de la liste d'action initiale : `AddProperty(cp1,InterfaceSource,ISClient1.int1,3)`. L'action numéro 3) revient à supprimer l'action numéro 2) de la liste d'action initiale `AddProperty(cp1,Name,Client1,2)`. L'action numéro 4) revient à supprimer

l'action numéro 1) de la liste d'action initiale : `Creat(cp1,Composant,1)`. Le résultat de cette évolution est la liste d'actions présentées ci-dessous.

```
1)Create(cp2,Composant,4)
2)ddProperty(cp2,Name,Serveur1,5)
3)AddProperty(cp2,InterfaceCible,ISServeur.int2,6)
4)Create(co1,Composant,7)
5)AddProperty(co1,Name,RPC,8)
6)AddProperty(co1,InterfaceSource,ISRPC.int5,9)
7)AddProperty(co1,InterfaceCible,ICRPC.int4,10)
8)AddProperty(co1,LinkSi,ICServeur.int2,11)
```

Listing 5.1: Liste d'action résultat

5.2.1.2 Vérification de cohérence et proposition du plan de réparation :

À cette étape une vérification de la cohérence sémantique est lancée sur le modèle évolué représenté par la séquence d'action en Praxis [12]. La vérification se fait via des règles appelées règles d'incohérences. Pour chaque propriété sémantique, définie dans la section 3.4.1 du chapitre 3, correspond une règle de vérification de cohérence exprimée en *JAVA [50]*. Les incohérences trouvées sont présentées par une liste d'action en Praxis comme source de l'incohérence.

Figure 5.8: Vérification de cohérence

En parcourant la liste des incohérences trouvées suite à la vérification de la cohérence, et en utilisant les règles de solution pour chaque incohérence spécifiée dans la phase spécification de l'évolution. Une liste d'action de correction est proposée au concepteur appelée plan de réparation en utilisant l'algorithme présenté dans la figure 5.9. Le plan de réparation peut être appliqué ou ignoré par le concepteur.

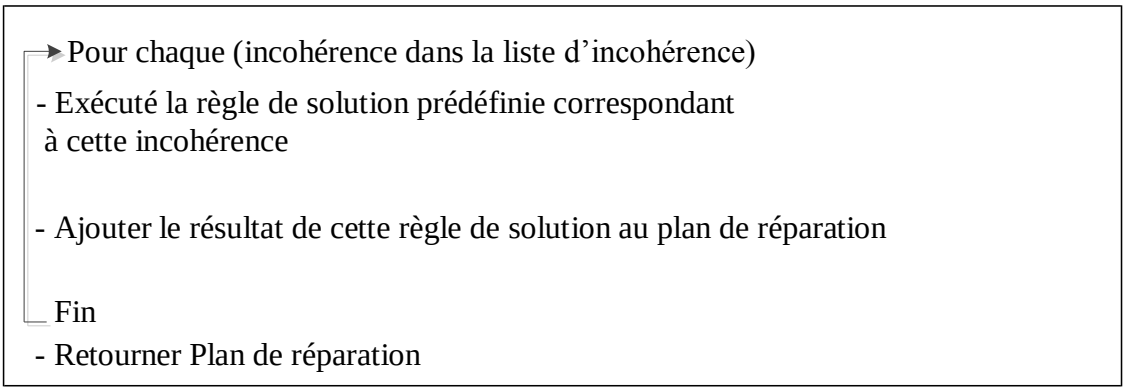


Figure 5.9: Algorithme du plan de réparation

5.2.1.2.1 Règles de cohérences :

En se basant sur les travaux [9] fait sur les propriétés sémantiques définies au niveau connecteur, nous présentons uniquement deux règles de vérification de cohérence avec leurs solutions correspondantes aux deux propriétés Exclusivité et Card-Cardinalité Inverse.

Exclusivité :

Règle de vérification de cohérence Propriété exclusivité (exclusive) pour le connecteur (con)	**Solution de l'incohérence**
Si valeur (exclusive)=Ps-Ec **alors** récupérer le lien **cible** **Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liencible et A.valeur != cible alors signaler incohérence **Finpour** **Si** valeur (exclusive)=Es-Pc **alors** récupérer le lien **source** **Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liensource et A.valeur != source alors **signaler incohérence** **Finpour** **Si** valeur (exclusive)=E **alors** récupérer les liens **cible&source.** **Pour** tout A € liste d'action tel que (A.IdElement=con et A.NomProp=Liencible et A.valeur != cible) ou(A.IdElement=con et A.NomProp=Liensource et A.valeur != source) alors **signaler incohérence.** **Finpour**	**Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liencible alors *NombreLienCible*++ ; **Finpour** **Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liensource alors *NombreLiensource*++ ; **Finpour** **Si** (*NombreLienCible*=1) et (*NombreLiensource*=1) **alors** Valeur (exclusive)=E. **Si** (*NombreLienCible*>1) et (*NombreLiensource*=1) **alors** Valeur (exclusive)=Ps-Ec. **Si** (*NombreLienCible*=1) et (*NombreLiensource*>1) **alors** Valeur (exclusive)=Es-Pc. **Si** (*NombreLienCible*>1) et (*NombreLiensource*>1) **alors** Valeur (exclusive)=P.

Tableau 5.2: Règle de vérification de cohérence de la propriété sémantique Exclusivité

Card-Cardinalité Inverse :

Règle de vérification de cohérence Propriété Card-Cardinalité Inverse (card-cardInv) pour le connecteur (con)	**Solution de l'incohérence**
Pour tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liencible alors *NombreLienCible*++ ; **Finpour** **Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liensource alors *NombreLiensource*++ ; **Finpour** **Si** (*NombreLienCible*>**cardInverse) ou** (*NombreLiensource*> **card) alors** **Signaler incohérence.**	**Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liencible alors *NombreLienCible*++ ; **Finpour** **Pour** tout A € liste d'action tel que A.IdElement=con et A.NomProp=Liensource alors *NombreLiensource*++ ; **Finpour** **Si** (*NombreLienCible*=1) et (*NombreLiensource*=1) **alors** Valeur **(card-cardInv)**=(1,1). **Si** (*NombreLienCible*>1) et (*NombreLiensource*=1) **alors** Valeur **(card-cardInv)**=(1, *NombreLienCible*). **Si** (*NombreLienCible*=1) et (*NombreLiensource*>1) **alors** Valeur **(card-cardInv)**=(*NombreLiensource*,1). **Si** (*NombreLienCible*>1) et (*NombreLiensource*>1) **alors** Valeur **(card-cardInv)**=(*NombreLienCible*, *NombreLiensource*)

Tableau 5.3: Règle de vérification de cohérence de la propriété sémantique Cardinalité-Cardinalité inverse

Exemple :

Pour la propriété sémantique Exclusivité Partage correspond une règle de vérification de cohérence exprimée en JAVA [50] par la méthode *vérifierIncohérence (String, String)* de la figure 5.10 où le premier attribut représente la valeur de la propriété sémantique et le deuxième attribut représente le type de lien à vérifier.

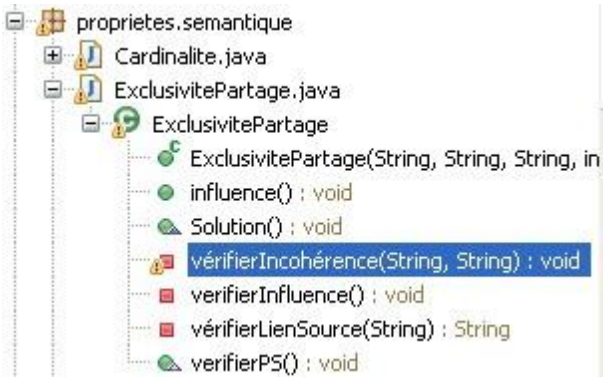


Figure 5.10: Méthode de vérification de cohérence implémentée pour la propriété Exclusivité/Partage

5.3 Partié dépéndanté du méta modélé

Dans cette section deux implémentations sont présentées, une implémentation pour xADL [49 ,51] et une autre pour ACME [36]. Un exemple d'évolution est présenté, pour chacun des deux ADL ACME et xADL, afin d'illustrer une application du modèle d'évolution IMoSCM.

5.3.1 Validation pour xADL:

Dans ce qui suit une présentation de l'ADL xADL est donnée, suivi d'une présentation détaillée des trois étapes de la partie dépendante du méta modèle : grammaire des opérations d'évolution, stratégies d'évolution, transformation vers Praxis & et la transformation inverse. En dernier un exemple d'évolution est présenté afin d'illustrer une application du modèle d'évolution IMoSCM.

5.3.1.1 Présentation de xADL :

L'ADL xADL [49 ,51] est un langage de description d'architecture logicielle (ADL [6]) développé par l'Université de Californie Irvine [49 ,51]. Contrairement à de nombreux autres ADL, xADL est défini comme un ensemble de schémas XML. xADL permet la modélisation de la majorité des concepts utilisés dans une modélisation structurelle d'architectures logicielles (*Composant*, *Connecteur* et *interface*), permettant la manipulation de ces concepts et par conséquent pouvoir les faire évoluer. Ainsi xADL attribue aux connecteurs une importance semblable à celui des composants. De ce fait, il permet de définir une extension du concept connecteur de xADL par les propriétés sémantiques. La force principale de xADL est son extensibilité. En fait, là où les méthodologies divergent, xADL tend à adopter une approche neutre en identifiant les écarts entre ce que xADL peut modéliser et la nouvelle modélisation voulue, choisit l'élément à étendre pour répondre au besoin de la nouvelle modélisation, propose une syntaxe pour coder ces extensions ainsi que les éléments nouveaux et génère une bibliothèque de nouvelles données pour ces extensions.

Du point de vue structurel, xADL est construit autour des éléments de base suivants :

- **Composants**: Les composants sont les unités de calcul de l'architecture. Avec un identificateur unique et une description textuelle, et un ensemble d'interfaces.

- **Connecteurs**: Connecteurs sont les lieux de la communication dans la conception de l'architecture, ils sont semblables aux composants, ils ont un identifiant unique, une description textuelle, et un ensemble d'interfaces.

- **Interfaces**: les interfaces sont considérées comme des portes pour les composants et les connecteurs afin de communiquer avec l'extérieur. Une «interface» dans xADL est appelée "port" pour les composants et «rôles» pour les connecteurs. Les interfaces ont un identifiant unique, une description textuelle, et une direction. La direction indique si l'interface est fournie, requise, ou les deux.

- **Liens**: Les liens sont les connexions entre les interfaces, définissant la topologie de l'architecture.

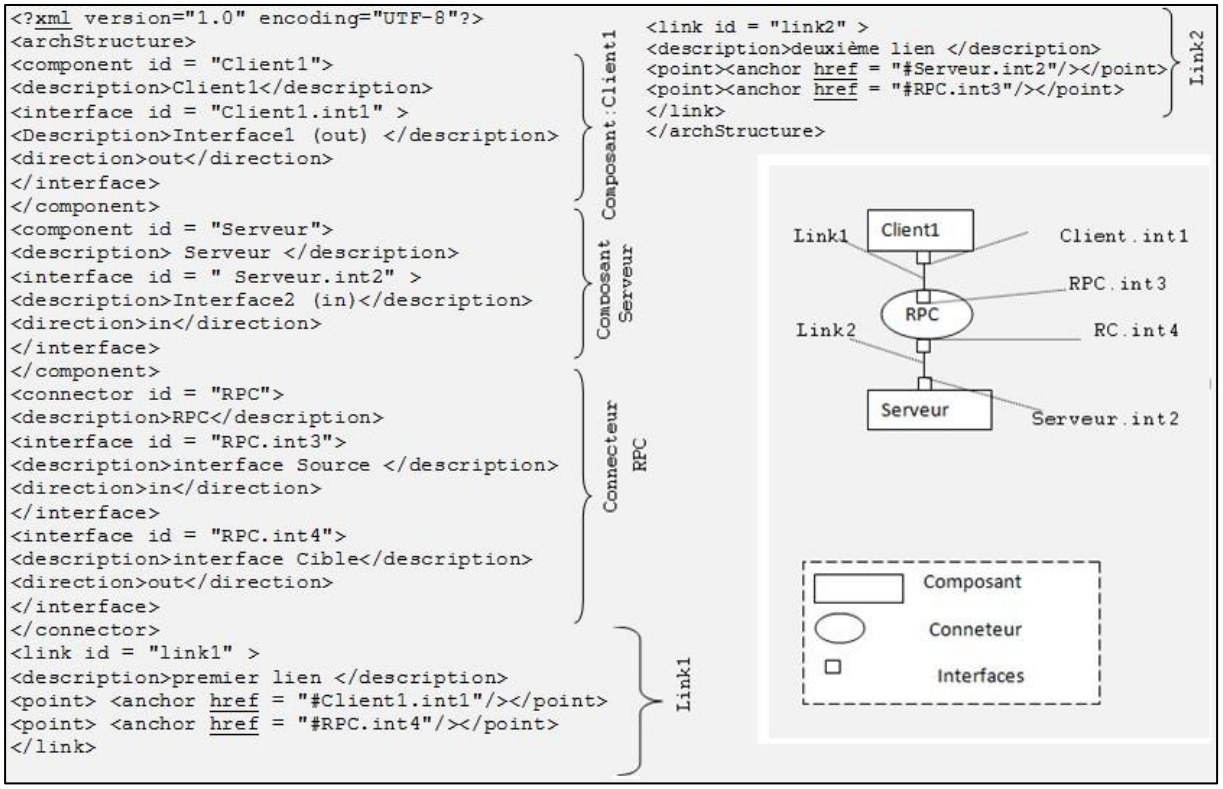


Figure 5.11: Exemple Modèle xADL en XML.

La figure 5.11 ci-dessus montre un exemple de description d'architecture contenant deux composants (*Client1 et Serveur*), un connecteur *RPC* et des liens pour connecter le connecteur *RPC* aux composants, l'identifiant d'une interface peut être n'importe quelle chaîne

unique. Dans cet exemple l'identifiant est construit, en suivant le modèle «élément-nom-dot-Numéro-interface", comme *Client.int1*. Les références sont représentées par la propriété *href*, contenant un identificateur unique.

xADL [49 ,51] est supporté par un ensemble d'outils intégrés dans la plate-forme Eclipse[50], appelé ArchStudio. ArchStudio[49 ,51] contient à la fois un éditeur visuel, qui permet de manipuler graphiquement l'architecture et un éditeur de la spécification à des fins spéciales.

5.3.1.2 Grammaire des opérations d'évolution en DTD :

Le spécialiste de modélisation de xADL [49 ,51] construit une grammaire en DTD à fin de représenter le modèle d'évolution comme suit:

```
< ! ELEMENT operation-Evolution (add*, remove*, update*)>
<! ELEMENT add
((nom,type,parent?)| (nom,type,cible,source)| (nom,type,parent)|
(nom,type,parent,valeur))>
< !ELEMENT nom(#PCDATA)>
< !ELEMENT type(#PCDATA)>
< !ELEMENT parent(#PCDATA)>
< !ELEMENT cible(#PCDATA)>
< !ELEMENT source(#PCDATA)>
< !ELEMENT valeur(#PCDATA)>
<! ELEMENT remove
((nom,type,parent?)|(nom,type,cible,source)|(nom,type,parent))>
< !ELEMENT nom(#PCDATA)>
< !ELEMENT type(#PCDATA)>
< !ELEMENT parent(#PCDATA)>
< !ELEMENT cible(#PCDATA)>
< !ELEMENT source(#PCDATA)>
<! ELEMENT update
(nom,type,parent,valeur)>
< !ELEMENT nom(#PCDATA)>
< !ELEMENT type(#PCDATA)>
< !ELEMENT parent(#PCDATA)>
< !ELEMENT valeur(#PCDATA)>
```

Listing 5.2: Grammaire d'opération d'évolution pour xADL

Une liste d'opération d'évolution peut contenir plusieurs, zéro ou un des opérations suivantes : `add`, `remove`, `update` marquées par le quantifieur *.

L'opération est toujours spécifiée par un nom et un type. Le type peut être un composant, un connecteur, une interface, un lien ou une propriété sémantique.

L'élément `add` doit contenir :

1) Soit le `nom` de l'élément à ajouter, son `type` et le `parent` auquel il appartient, le champ `parent` est facultatif, marqué par le quantifieur ? et utilisé dans le cas d'un composant composite.
2) Soit le `nom` de l'élément à ajouter, son `type` et le parent auquel il appartient.
3) Soit le `nom` de l'élément à ajouter, son `type`, le `parent` auquel il appartient et la `valeur` pour le cas d'ajout de propriété sémantique.

4) Soit le `nom` de l'élément à ajouter, son `type`, la `cible` et la `source` d'une connexion entre deux composants.

L'élément `remove` doit contenir :

1) Soit le `nom` de l'élément à supprimer, son `type` et le `parent` auquel il appartient, le champ `parent` est facultatif, marqué par le quantifieur ? et utilisé dans le cas d'un composant composite.
2) Soit le `nom` de l'élément à supprimer, son `type` et le parent auquel il appartient.
3) Soit le `nom` de l'élément à supprimer, son `type`, la `cible` et la `source` d'une connexion entre deux composants.

L'élément `update` doit contenir :

1) Le `nom` de l'élément à mettre à jour, son `type`, le `parent` auquel il appartient et la `valeur` pour le cas de modification de propriété sémantique.

Exemples :

1) Ajouter un composant nommé Client3 :

<add Name="Client3" type="Composant"/>

2) Ajouter un composant nommé Serveur1 :

<remove Name="Serveur1" type="Composant"/>

3) Mettre à jour la propriété sémantique ***exPar*** du connecteur ***RPC1*** avec la nouvelle valeur ***E*** *:*

<update Name="exlPar" type="PropSem" parent="RPC1" valeur="E"/>

4) Ajouter une interface cible pour le connecteur *RPC1* :

<add Name="InerfaceCible" type="Interface" parent="RPC1"/>

5) Ajouter un lien entre l'interface cible du composant *serveur1* et l'interface source du connecteur *RPC1* :

<add Name="LinkCi" type="Link" cible="Serveur1" source="RPC1"/>

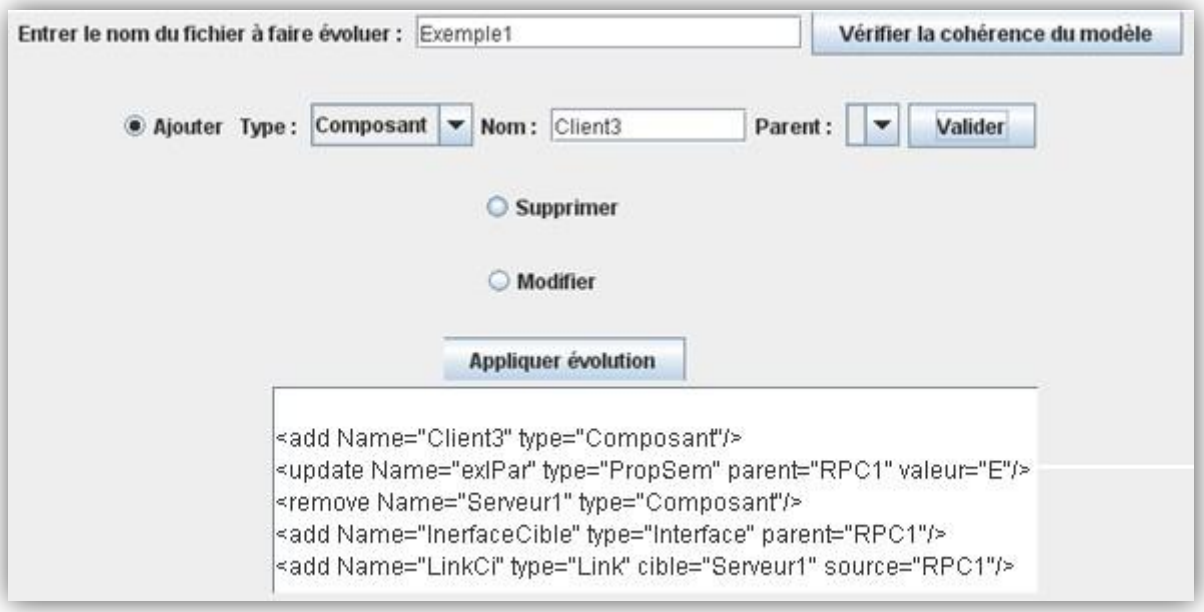


Figure 5.12: Capture d'image de l'interface graphique montrant le choix des opérations d'évolutions et un exemple du modèle d'évolution

La figure 5.12 illustre l'interface graphique du choix d'opération d'évolution: ajouter, supprimer et modifier. Ainsi qu'un exemple d'un modèle d'évolution en utilisant la grammaire définie par le spécialiste de modélisation représentant trois ajouts, une mise à jour et une suppression.

5.3.1.3 Stratégies d'évolution:

Le tableau 5.4 présente les stratégies d'évolutions associées aux éléments architecturaux de xADL [49 ,51]. Pour chaque stratégie nous avons apporté des exemples d'action d'évolution qui nous servirons dans l'exemple d'illustration.

Stratégie	Composant	Connecteur	Interface
Stratégie d'ajout : Actions	**-Ajouter Composant**	**-Ajouter connecteur** **-Ajouter propriété sémantiques.**	**-Ajouter interface.**
Stratégie de suppression : Actions	**-supprimer ces liens** **-supprimer ces interfaces** **-supprimer le composant**	**-supprimer ces liens** **-supprimer ces interfaces** **-supprimer le connecteur**	**-supprimer ces liens** **-supprimer l'interface**
Stratégie de modification : Actions	/	**-supprimer ancienne Propriété sémantique** **-ajouter une nouvelle propriété sémantique avec la nouvelle valeur**	/

Tableau 5.4: Stratégie d'évolution

Exemple :

Une opération d'évolution de suppression d'un composant nommé « **`Client1`** » de l'exemple de la figure 5.11 est générée à partir de l'application comme suit : **`<remove Name=`** « **`Client1`** » **`type=`** «**`Composant`** »**`/>`**. La stratégie de suppression est alors déclenchée comme suit :

1. Supprimer les liens vers ces interfaces.
2. Supprimer les interfaces de ce composant.
3. Supprimer le composant lui-même.

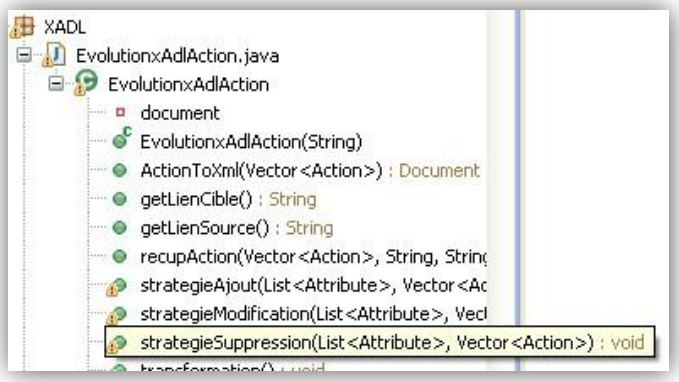


Figure 5.13: Méthode stratégie Suppression implémentée pour l' ADL xADL

Cette stratégie est implémentée par la méthode ***strategieSuppression(List<Attribute>, Vector<Action>)*** de la figure 5.13 où ***List <Attribute>*** représente l'opération en DTD et ***Vector<Action>*** représente la séquence d'action en *PRAXIS* [12] du modèle initiale. Le résultat de cette méthode est la liste d'action d'évolution en *PRAXIS* suivantes :

```
1)RemProperty(Co1,LinkCi,ISClient1.int1,12)
2)RemProperty(cp1,InterfaceSource,ISClient1.int1,13)
3)RemProperty(cp1,Name,Client1,14)
4)Delete(cp1,15)
```

Listing 5.3: Opération d'évolution en Praxis

5.3.1.4 Transformation modèle xADL vers Praxis & transformation inverse :

Après avoir présenté le modèle xADL [49 ,51] en XML « voir figure 5.11 » et le modèle d'évolution « voir figure 5.12 », cette section s'intéresse à la transformation vers praxis [12] de ces deux modèles.

Opération en xADL	**Opérations Praxis équivalentes**	**Explication**
Si el.getName= component	**Create**(idComposant,Composant,temps)	-idComposant: est l'identificateur du composant à créer. -Composant: est le mot clé pour la création. -Temps: est le temps d'exécution.
Si el.getName= connecteur	**Create**(idConnecteur,Connecteur,temps)	idConnecteur: est l'identificateur du connecteur à créer. -Connecteur: est le mot clé pour la création. -Temps: est le temps d'exécution.

Si el.getName= Link	**AddProperty** (idConnecteur,Link,valeurP,temps)	idConnecteur: est l'identificateur du connecteur à relier. -Link: est le type de lien, source ou cible. -valeur P : est la valeur de la propriété correspond à l'interface d'un composant à relier avec l'interface du connecteur idConnecteur. -Temps: est le temps d'exécution.
Si el.getName= Interface	**AddProperty** (idElement,Interface,idInterface,temps)	-idElemen: est l'identificateur de l'élément pour lequel l'interface est ajoutée. -Interface : est le type d'interface, source ou cible à créer. -idInterface: est l'identificateur de l'interface crée. -Temps: est le temps d'exécution.
Si el.getName= Propriete	**AddProperty** (idElement,,nameProprerty,valeur,temps)	-idElemen: est l'identificateur de l'élément pour lequel la propriété est ajoutée. -nameProprety : est le nom de la propriété. -valeur: est la valeur de la propriété. -Temps: est le temps d'exécution.

Tableau 5.5: Règles de transformation de xADL vers Praxis

La transformation du modèle xADL vers Praxis est spécifiée en parcourant le modèle XML et en utilisant les règles de transformations de xADL vers praxis données dans le tableau 5.5. La transformation inverse est spécifiée en parcourant le chemin inverse de la transformation du Modèle xADL vers praxis.

Les opérations d'évolutions présentées dans le modèle d'évolution sont transformées en une suite d'action de PRAXIS. Comme chaque opération d'évolution correspond à une stratégie d'évolution voir section 3.4.1.3 du chapitre III. Cette dernière est exécutée en *JAVA* [50] et exprimée par une liste d'action d'évolution en *PRAXIS* ce qui va enrichir le modèle d'évolution en utilisant les règles de transformations présentées dans le tableau 5.6.

Opération d'évolution	**Opérations Praxis équivalentes**	**Explication**
Si el.getName=Add	**AddProperty**(idElement,nameProperty, valeurP,temps) **Create**(idelement,Typeelement,temps)	-*idElement* : est l'identificateur de l'élément pour lequel la propriété est ajoutée. -*nameProperty :* est le nom de la propriété. (interface, propriété, lien,nom) - *valeurP*: est la valeur de la propriété. -*Temps*: est le temps d'exécution -*TypeElement* : est le type de l'élément à créer. (Composant, connecteur).
Si el.getName=remove	**RemProperty**(idélémentr,nameProperty, valeurP,temps) **Delete**(idelement,Typeelement,temps)	-*idElement* : est l'identificateur de l'élément pour lequel la propriété est ajoutée. -*nameProperty :* est le nom de la propriété. (interface, propriété,

		lien,nom) - *valeurP*: est la valeur de la propriété. -*Temps*: est le temps d'exécution -*TypeElement* : est le type de l'élément à créer. (Composant, connecteur).
Si el.getName=update	**RemProperty**(idélémentr,nameProperty, valeurP,temps) **AddProperty**(idélémentr,nameProperty, valeurP,temps)	*idElement* : est l'identificateur de l'élément pour lequel la propriété est ajoutée. -*nameProperty :* est le nom de la propriété. (interface, propriété, lien,nom) - *valeurP*: est la valeur de la propriété. -*Temps*: est le temps d'exécution

Tableau 5.6: Règles de transformations du modèle d'évolution vers Praxis

```
1. Create(cp1,Composant,1)
2. AddProperty(cp1,Name,Client1,2)
3. AddProperty(cp1,InterfaceSource,ISClient1.int1,3)
4. Create(cp2,Composant,4)
5. AddProperty(cp2,Name,Serveur1,5)
6. AddProperty(cp2,InterfaceCible,ISServeur.int2,6)
7. Create(co1,Composant,7)
8. AddProperty(cco1,Name,RPC,8)
9. AddProperty(co1,InterfaceSource,ISRPC.int5,9)
10.AddProperty(co1,InterfaceCible,ICRPC.int4,10)
11.AddProperty(co1,LinkCi,ISClient1.int1,11)
12.AddProperty(co1,LinkSi,ICServeur.int2,11)
```

Listing 5.4: Exemple de transformation vers Praxis du modèle en XML présenté dans la figure 5.11

Le modèle en Praxis est implémenté par un vecteur *d'actions*. Les *actions* représentent les actions élémentaires de Praxis *(Create, Delete, AddProprety, RemoveProprety, AddReference, RemoveReference)*

5.3.2 Validation pour ACME:

Dans ce qui suit une présentation de l'ADL ACME est donnée, suivi d'une présentation détaillée des trois étapes de la partie dépendante du méta modèle : grammaire des opérations d'évolution, stratégies d'évolution, transformation vers Praxis & et la transformation inverse. En dernier un exemple d'évolution est présenté afin d'illustrer une application du modèle d'évolution IMoSCM.

5.3.2.1 Présentation d'ACME :

L'ADL ACME [36] est un langage de description d'architecture établi par la communauté scientifique dans le domaine des architectures logicielles. La particularité d'Acme [36] s'appuie sur les caractéristiques communes de l'ensemble des ADLs [6], en fournissant un langage permettant d'intégrer facilement de nouveaux ADLs. Le langage fournit un ensemble de sept concepts permettant de spécifier la structure d'une architecture : composant, connecteur, système, port, rôle, la représentation et la carte de représentation (*rep-map*).

- Le **composant** représente l'unité de traitement ou de donnée d'une application. Par exemple, un client ou un serveur est un composant. Il est spécifié par une interface composée de plusieurs types de ports. Chaque type de port identifie un point d'interaction entre le composant et son environnement.

- Le **connecteur** représente l'interaction entre composants. Il s'agit d'un médiateur de communication qui coordonne les connexions entre composants. Il est spécifié par une interface composée d'un ensemble de type de rôles. Chaque type de rôle d'un connecteur définit un participant à une interaction.

- Le **système** représente la configuration d'une application, c'est-à-dire l'assemblage structurel entre les composants et les connecteurs.

- La **représentation et la carte de représentation** permettent à ACME [36] de supporter la description hiérarchique d'une architecture. Ainsi, un composant ou un connecteur peut être décrit d'un niveau général à un niveau plus détaillé et peut donc être raffiné. Chaque nouvelle description (sous élément) d'un élément est appelée une représentation. La correspondance entre l'élément et ses représentations est spécifiée grâce à la carte de représentation. Ainsi, la carte de représentation permet d'établir la correspondance entre les ports de l'interface d'un composant et ceux définis dans les interfaces de ses sous composants.

L'exemple suivant décrit l'architecture client-serveur en ACME [36] :

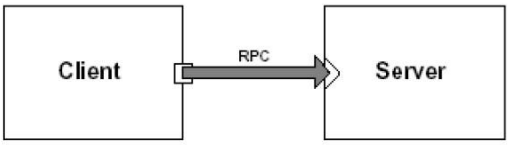


Figure 5.14 Architecture client-serveur en ACME

```
System simple_cs = {
Component client = { Portsend-request }
Component server = { Portreceive-request }
Connector rpc = { Roles{caller,callee}}
Attachments : {
client.send-request to rpc.caller ;
server.receive-request to rpc.callee}
```

ACME [36] propose un moyen d'intégrer des caractéristiques à travers la notion de propriété. Une propriété a un nom qui l'identifie, un type optionnel et une valeur. Chaque élément du design décrit ci-dessus (composant, connecteur) peut avoir une ou plusieurs propriétés.

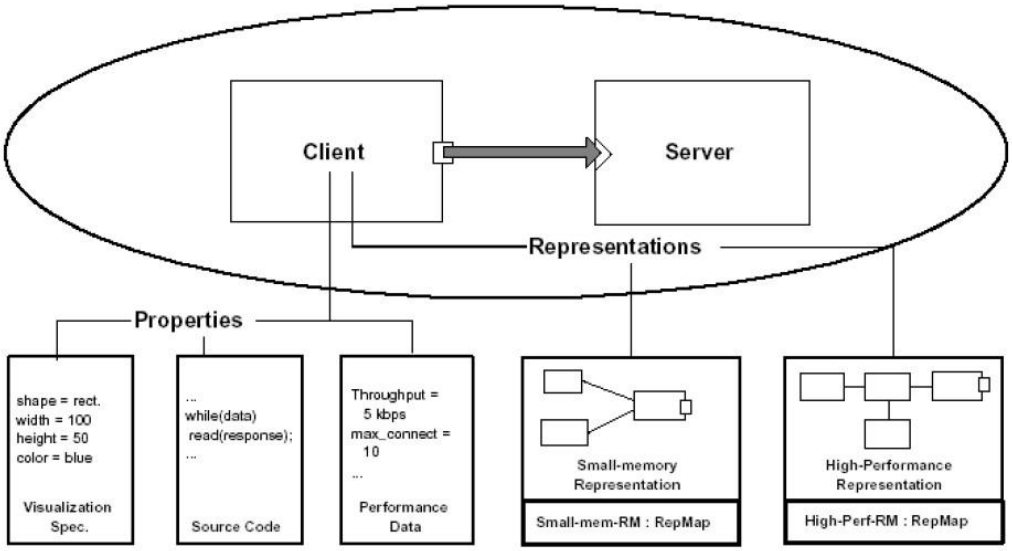


Figure 5.15 Intégration des caractéristiques à travers la notion de propriété.

ACME [36] fournit un moyen de décrire des gabarits de conception (templates). Cette notion est équivalente à la notion de style d'architecture que l'on peut trouver dans la plupart des ADLs. Le langage permet de créer des gabarits de conception paramétrables et réutilisables permettant de spécifier des patrons de conception. Le langage ACME [36] a été conçu pour spécifier une architecture de manière syntaxique et ne se focalise pas sur la sémantique. Néanmoins, il propose un cadre (OpenSemantic Framework) fournissant une base pour décrire la sémantique d'un système de manière formelle en donnant ainsi une manière d'imiter certains ADLs qui permettent de spécifier le comportement des architectures.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<AcmeModel>
<System name= "ACME">
<userdata key="connection-patterns">
      <data>
        <portConnectionRule connector="Connector" name="RPC">
          <association componentType="Component" portName="ISClient1" portType="Port" roleName="ICRPC"> Link1
roleType="Role"/>
          <association componentType="Component" portName="ICServeur" portType="Port" roleName="ISRPC"> Link2
roleType="Role"/>
        </portConnectionRule>
     </data>
    </userdata>
      <component name="Client1">
      <port name="ICClient1">
      </port>
      <port name="ISClient1">
      </port>
    </component>
    <component name="Serveur">
      <port name="ICServeur">
      </port>
      <port name="ISServeur">
      </port>
    </component>
    <connector name="RPC">
      <role name="ISRPC">
      </role>
      <role name="ICRPC">
      </role>
      <property name="exlPar" value="E"></property>
      <property name="Cardinalite" value="1,1"></property>
    </connector>
</System>
</AcmeModel>
```

Composant Client1

Composant Serveur

Connecteur RPC

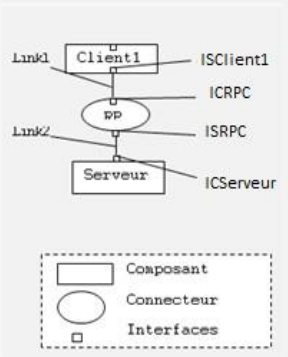


Figure 5.16: Exemple Modèle ACME en XML.

La figure 5.16 montre un exemple de description d'architecture contenant deux composants (*Client1* et *Serveur*), un connecteur *RPC* et des liens pour connecter le connecteur aux composants, l'identifiant d'une interface peut être n'importe quelle chaîne unique. Dans cet exemple l'identifiant est construit-en suivant le modèle «codeInteface-élément-nom », comme *ICClien1* avec le code Interface égale à IC qui signifie l'interface cible du composant *Client1*. ACME [36] est supporté par un ensemble d'outils intégrés dans la plate-forme Eclipse [50], appelé AcmeStudio. AcmeStudio est un environnement d'édition personnalisable et outil de visualisation pour la conception de logiciels d'architecture basé sur le langage de description d'architecture ACME.

5.3.2.2 Grammaire des opérations d'évolution en DTD :

Le spécialiste de modélisation d'ACME construit une grammaire en DTD à fin de représenter le *modèle d'évolution* comme suit:

```
< ! ELEMENT operation-Evolution (add*, remove*, update*)>
<! ELEMENT add
((nom,type,parent?)| (nom,type,port,role)| (nom,type,parent)|
(nom,type,parent,valeur))>
< !ELEMENT nom(#PCDATA)>
< !ELEMENT type(#PCDATA)>
< !ELEMENT parent(#PCDATA)>
< !ELEMENT port(#PCDATA)>
< !ELEMENT role(#PCDATA)>
< !ELEMENT valeur(#PCDATA)>
<! ELEMENT remove
((nom,type,parent?)|(nom,type,port,role)|(nom,type,parent))>
< !ELEMENT nom(#PCDATA)>
< !ELEMENT type(#PCDATA)>
< !ELEMENT parent(#PCDATA)>
< !ELEMENT port(#PCDATA)>
< !ELEMENT role(#PCDATA)>
<! ELEMENT update
(nom,type,parent,valeur)>
< !ELEMENT nom(#PCDATA)>
< !ELEMENT type(#PCDATA)>
< !ELEMENT parent(#PCDATA)>
< !ELEMENT valeur(#PCDATA)>
```

Listing 5.5: Grammaire d'opération d'évolution pour ACME

Une liste d'opération d'évolution peut contenir plusieurs, zéro ou un des opérations suivantes : `add`, `remove`, `update` marquées par le quantifieur *. L'opération est toujours spécifiée par un nom et un type. Le type peut être un composant, un connecteur, une interface, un lien ou une propriété sémantique.

L'élément `add` doit contenir :

1) Soit le `nom` de l'élément à ajouter, son `type` et le `parent` auquel il appartient, le champ `parent` est facultatif, marqué par le quantifieur `?` et utilisé dans le cas d'un composant composite.
2) Soit le `nom` de l'élément à ajouter, son `type` et le `parent` auquel il appartient.
3) Soit le `nom` de l'élément à ajouter, son `type`, le `parent` auquel il appartient et la `valeur` pour le cas d'ajout de propriété sémantique.

4) Soit le `nom` de l'élément à ajouter, son `type`, le `port` et le `role` d'une connexion entre deux composants.

L'élément `remove` doit contenir :

1) Soit le `nom` de l'élément à supprimer, son `type` et le `parent` auquel il appartient, le champ `parent` est facultatif, marqué par le quantifieur `?` et utilisé dans le cas d'un composant composite.
2) Soit le `nom` de l'élément à supprimer, son `type` et le `parent` auquel il appartient.
3) Soit le `nom` de l'élément à supprimer, son `type`, le `port` et le `role` d'une connexion entre deux composants.

L'élément `update` doit contenir :

1) Le `nom` de l'élément à mettre à jour, son `type`, le `parent` auquel il appartient et la `valeur` pour le cas de modification de propriété sémantique.

Exemples :

1) Ajouter un composant nommé Client3 : ***<add Name="Client3" type="Composant"/>***

2) Ajouter un composant nommé Serveur1 : ***<remove Name="Serveur1" type="Composant"/>***

3) Mettre à jour la propriété sémantique ***exPar*** du connecteur ***RPC1*** avec la nouvelle valeur ***E*** *:*

<update Name="exlPar" type="PropSem" parent="RPC1" valeur="E"/>

4) Ajouter une interface cible pour le connecteur RPC1 :

<add Name="InerfaceCible" type="Interface" parent="RPC1"/>

5) Ajouter un lien entre l'interface cible du composant *serveur1* et l'interface source du connecteur RPC1 :

<add Name="LinkCi" type="Link" port="Serveur1" role="RPC1"/>

5.3.2.3 Stratégies d'évolution:

Le tableau 5.7 présente les stratégies d'évolutions associées aux éléments architecturaux d'ACME [36]. Pour chaque stratégie nous avons apporté des exemples d'action d'évolution qui nous servirons dans l'exemple d'illustration.

Stratégie	Composant	Connecteur	Interface
Stratégie d'ajout : Actions	**-Ajouter Composant**	**-Ajouter connecteur** **-Ajouter propriété sémantique.**	**-Ajouter interface.**
Stratégie de suppression : Actions	**-supprimer ces liens** **-supprimer ces interfaces** **-supprimer le composant**	**-supprimer ces liens** **-supprimer ces interfaces** **-supprimer le connecteur**	**-supprimer ces liens** **-supprimer l'interface**
Stratégie de modification : Actions	/	**-supprimer liens** **-supprimer ancienne Propriété sémantique** **-ajouter une nouvelle propriété sémantique avec la nouvelle valeur**	/

Tableau 5.7: Stratégie d'évolution

Exemple :

Une opération d'évolution de suppression du lien entre le composant ***Client1*** et le connecteur ***RPC*** de l'exemple de la figure 5.16 est générée à partir de l'application comme suit : **<remove Name= « LinkCi » type= «lien » port= « RPC » rol= «Client1 »/>.** La stratégie de suppression est alors déclenchée comme suit :

1. Supprimer le lien.
2. Supprimer les interfaces **port** et **rol** du connecteur **RPC** et du composant **Client1** respectives.

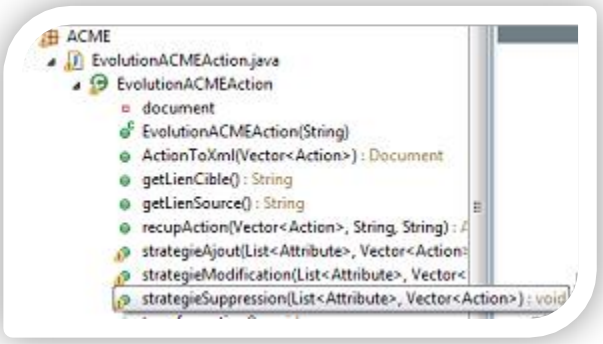


Figure 5.17: Méthode stratégie Suppression
implémentée pour l' ADL ACME

Cette stratégie est implémentée par la méthode ***strategieSuppression(List<Attribute>, Vector<Action>)*** de la figure 5.17 où ***List <Attribute>*** représente l'opération en DTD et ***Vector<Action>*** représente la séquence d'action en *PRAXIS* du modèle initiale. Le résultat de cette méthode est la liste d'action d'évolution en *PRAXIS* suivantes :

```
1)RemProperty(co1,LinkCi,ISClient1.int1,12)
2)RemProperty(cp1,InterfaceSource,ISClient1,13)
3)RemProperty(co1,InterfaceCible,ICRPC.int1,13)
```

Listing 5.6: Opération d'évolution en Praxis

5.3.2.4 Transformation modèle ACME vers Praxis & transformation inverse :

Après avoir présenté le modèle ACME [36] en XML « voir figure 5.16 » et le modèle d'évolution, cette section s'intéresse à la transformation vers praxis de ces deux modèles.

Opération en ACME	**Opérations Praxis équivalentes**	**Explication**
Si el.getName= component	**Create**(idComposant,Composant,temps)	-idComposant: est l'identificateur du composant à créer. -Composant: est le mot clé pour la création. -Temps: est le temps d'exécution.
Si el.getName= connecteur	**Create**(idConnecteur,Connecteur,temps)	idConnecteur: est l'identificateur du connecteur à créer. -Connecteur: est le mot clé pour la création. -Temps: est le temps d'exécution.
Si el.getName= portConnecti--onRule	**AddProperty** (idConnecteur,Link,valeurP,temps)	idConnecteur: est l'identificateur du connecteur à relier. -Link: est le type de lien, source ou cible. -valeur P : est la valeur de la propriété correspond à l'interface d'un composant à relier avec l'interface du connecteur idConnecteur. -Temps: est le temps d'exécution.
Si el.getName= role	**AddProperty** (idElement,Interface,idInterface,temps)	-idElemen: est l'identificateur du connecteur pour lequel le role est ajouté. -Interface : est le type du role, source ou cible à créer. -idInterface: est l'identificateur de

		l'interface crée. -Temps: est le temps d'exécution.
Si el.getName= port	**AddProperty** (idElement,Interface,idInterface,temps)	-idElemen: est l'identificateur du composant pour lequel le port est ajouté. -Interface : est le type du port, source ou cible à créer. -idInterface: est l'identificateur de l'interface crée. -Temps: est le temps d'exécution.
Si el.getName= Propriete	**AddProperty** (idElement,,nameProprerty,valeur,temps)	-idElemen: est l'identificateur de l'élément pour lequel la propriété est ajoutée. -nameProprety : est le nom de la propriété. -valeur: est la valeur de la propriété. -Temps: est le temps d'exécution.

Tableau 5.8: Règles de transformation d'ACME vers Praxis

La transformation du modèle ACME [36] vers Praxis [12] est spécifiée en parcourant le modèle XML et en utilisant les règles de transformations d'ACME vers praxis données dans le tableau 5.8. La transformation inverse est spécifiée en parcourant le chemin inverse de la transformation du Modèle ACME vers praxis.

Les opérations d'évolutions présentées dans le modèle d'évolution sont transformées en une suite d'action de PRAXIS. Comme chaque opération d'évolution correspond à une stratégie d'évolution voir section 3.4.1.3 du chapitre III. Cette dernière est exécutée en JAVA [50] et exprimée par une liste d'action d'évolution en PRAXIS ce qui va enrichir le modèle d'évolution en utilisant les règles de transformations présentées dans le tableau 5.9.

Opération d'évolution	**Opérations Praxis équivalentes**	**Explication**
Si el.getName=Add	**AddProperty**(idElement,nameProperty, valeurP,temps) **Create**(idelement,Typeelement,temps)	-*idElement* : est l'identificateur de l'élément pour lequel la propriété est ajoutée. -*nameProperty :* est le nom de la propriété. (interface, propriété, lien,nom) - *valeurP*: est la valeur de la propriété. -*Temps*: est le temps d'exécution -*TypeElement* : est le type de l'élément à créer. (Composant, connecteur).
Si el.getName=remove	**RemProperty**(idelement,nameProperty, valeurP,temps) **Delete**(idelement,Typeelement,temps)	-*idElement* : est l'identificateur de l'élément pour lequel la propriété est ajoutée. -*nameProperty :* est le nom de la propriété. (interface, propriété, lien,nom) - *valeurP*: est la valeur de la propriété. -*Temps*: est le temps d'exécution -*TypeElement* : est le type de l'élément à créer. (Composant, connecteur).
Si	**RemProperty**(idelement,nameProperty, valeurP,temps)	*idElement* : est l'identificateur de l'élément pour lequel la propriété

el.getName=update	**AddProperty**(idelemente,nameProperty ,valeurP,temps)	est ajoutée. -*nameProperty :* est le nom de la propriété. (interface, propriété, lien,nom) - *valeurP*: est la valeur de la propriété. -*Temps*: est le temps d'exécution

Tableau 5.9: Règles de transformations du modèle d'évolution vers Praxis

```
1. Create(cpClient1,Composant,1)
2. AddProperty(cpClient1,Name,Client1,2)
3. AddProperty(cpClient1,InterfaceCible,ICClient1,3)
4. AddProperty(cpClient1,InterfaceSource,ISClient1,4)
5. Create(cpServeur,Composant,5)
6. AddProperty(cpServeur,Name,Serveur,6)
7. AddProperty(cpServeur,InterfaceCible,ICServeur,7)
8. AddProperty(cpServeur,InterfaceSource,ISServeur,8)
9. Create(coRPC1,Connecteur,9)
10.AddProperty(coRPC1,Name,RPC1,10)
11.AddProperty(coRPC1,InterfaceSource,ISRPC1,11)
12.AddProperty(coRPC1,InterfaceCible,ICRPC1,12)
13.AddProperty(coRPC1,exlPar,E,13)
14.AddProperty(coRPC1,Cardinalite,2,2,14)
15.AddProperty(coRPC1,LinkCi,ISClient1,15)
16.AddProperty(coRPC1,LinkSi,ICServeur,16)
```

Listing 5.7: Exemple de transformation vers Praxis du modèle en XML présenté dans la figure 5.16

5.4 Exemples appliqués :

Nous utilisons l'exemple Client/serveur tiré de [9], voire la figure 5.18 afin de présenter l'utilisation des propriétés sémantiques et une application du modèle d'évolution IMoSCM. Cet exemple est souvent présenté et traité précisément par la plupart des ADL [6].

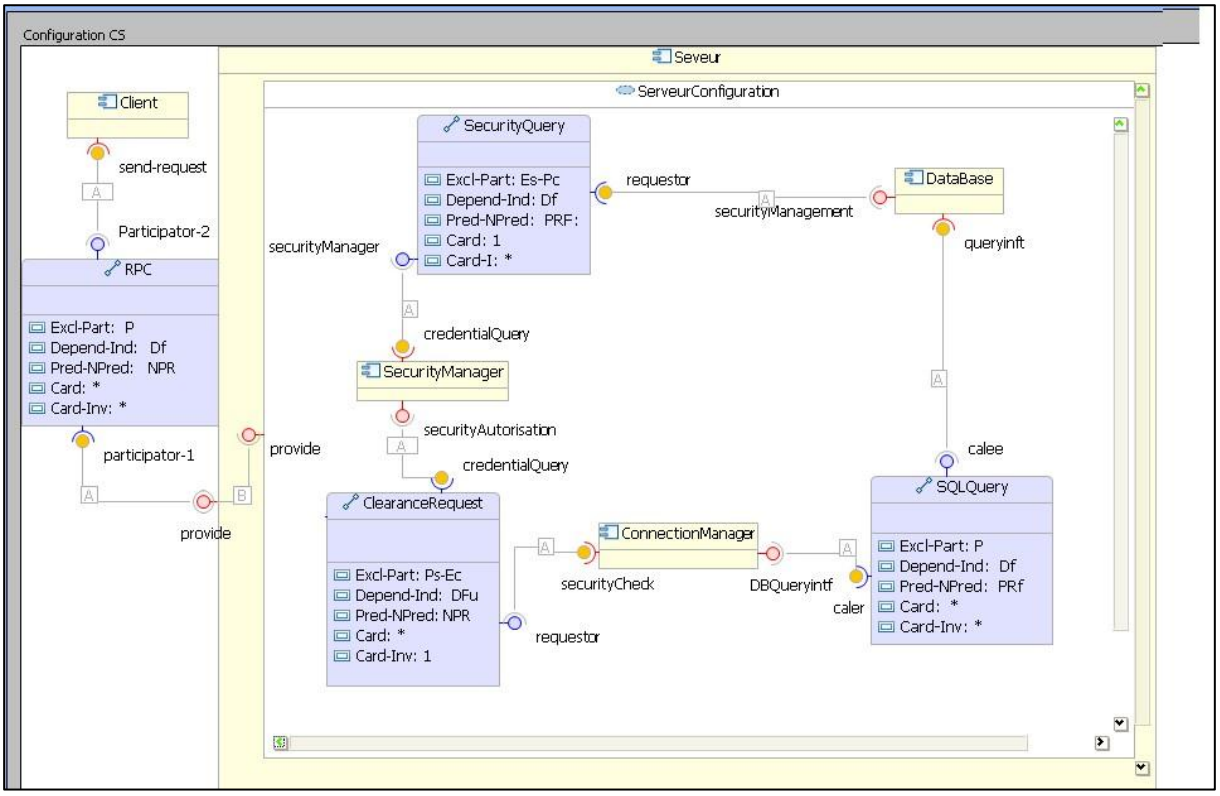


Figure 5.18: Exemple Architecture Client Serveur [9]

Exemple d'évolution de l'ADL ACME :

Dans l'exemple de la figure 5.18 ci-dessus, nous voulons rendre le connecteur **RPC** exclusif. Ceci implique que les deux interfaces source et cible du connecteur **RPC** interagissent uniquement entre elles. Cette évolution est exprimée par l'opération d'évolution « Modification de la valeur de la propriété sémantique Exclusivité-Partage du connecteur **RPC**». Dans ce qui suit nous détaillons cette évolution en précisant les différentes spécifications et en l'exécutant via le modèle IMoSCM.

Spécification de l'évolution :

1. **Les propriétés sémantiques ainsi que les invariants associés aux éléments architecturaux :** Le tableau 5.10 présente les éléments architecturaux avec leurs propriétés sémantiques et les invariants :

Elément architectural	**Propriétés sémantiques**	**Invariants**
Composant	/	-Un composant est constitué au moins d'une interface fournie. -Un composant doit être relié au moins à un autre composant.
Connecteur	***Exclusivité*** ***Cardinalité*** - ***Cardinalité Inverse***	-Un connecteur est constitué au moins d'une interface fournie et une interface requise. -un connecteur doit relier au moins deux composants.
Interface	/	-Une interface ne doit pas être libre.

Tableau 5.10: éléments architecturaux avec leurs propriétés sémantiques et invariants

2. Les stratégies d'évolutions associées aux éléments architecturaux à faire évoluer sont présentées dans la section 5.3.2.3.
3. Les règles de cohérences à vérifier ainsi que les solutions correspondantes aux incohérences sont présentées dans la section 5.2.1.2.1.
4. Règles de transformation du modèle, à faire évoluer, vers praxis et inversement sont présentés dans la section 5.3.2.4.

Exécution de l'évolution et vérification de cohérence :

Après spécification de l'évolution, le concepteur sélectionne l'opération d'évolution à appliquer sur l'architecture. Cette dernière est interprétée par le langage ***<update Name="exlPar" type="PropSem" parent="RPC" valeur="E"/>*** comme montrer dans la figure 5.19 :

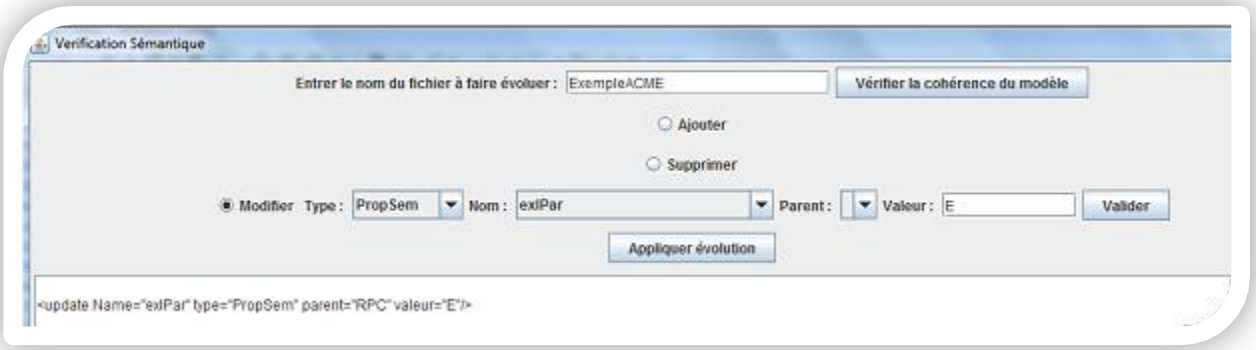


Figure 5.19: Opération d'évolution en DTD

En cliquant sur le bouton appliquer évolution :

1. L'architecture *Client/Serveur* en *XML* est transformée en *PRAXIS* en utilisant les règles de transformations. Le tableau 5.11 ci-dessous présente un exemple de transformation du composant *Client*, du connecteur *RPC* ainsi que la connexion entre eux comme suit :

Elément modèle	XML	Praxis
Composant *Client*	<component name="Client"> <port name="ICClient"> </port> <port name="ISClient"> </port> </component>	Create(cpClient,Composant,9) AddProperty(cpClient,Name,Client,10) AddProperty(cpClient,InterfaceCible,ICClient,11) AddProperty(cpClient,InterfaceSource,ISClient,12)
Connecteur *RPC*	<connector name="RPC"> <role name="ISRPC"> <userdata key="vis-information"> <data category="role" rotation="0.0" x="420" y="47"/> </userdata> </role> <role name="ICRPC"> <userdata key="vis-information"> <data category="role" rotation="0.0" x="614" y="50"/> </userdata> </role> <property name="exlPar" value="E"> </property> <property name="Cardinalite" value="2,2"> </property> </connector>	Create(coRPC,Connecteur,17) AddProperty(coRPC,Name,RPC,18) AddProperty(coRPC,InterfaceSource,ISRPC,19) AddProperty(coRPC,InformationRolISRPC,vis-information,20) AddProperty(coRPC,InterfaceCible,ICRPC,21) AddProperty(coRPC,InformationRolICRPC,vis-information,22) AddProperty(coRPC,exlPar,E,23) AddProperty(coRPC,Cardinalite,2,2,24)
Connecxion	<portConnectionRule connector="Connector" name="RPC"> <association componentType="Component" portName="ISClient" portType="Port" roleName="ICRPC" roleType="Role"/> </portConnectionRule>	AddProperty (coRPC,LinkCi,ISClient.40)

Tableau 5.11: Transformation de XML vers PRAXIS de l'ADL ACME

Figure 5.20: Modèle ACME sous format PRAXIS

2. Le modèle d'évolution constitué de l'opération **Update** est interprété en utilisant la stratégie d'évolution de **Modification** :

   a) Supprimer les liens de ce connecteur.

   b) Supprimer ancienne Propriété sémantique.

c) Ajouter une nouvelle propriété sémantique avec la nouvelle valeur.

Cette stratégie est transformée en PRAXIS, en utilisant les règles de transformation, comme montrer dans la figure 5.21 ci-dessous :

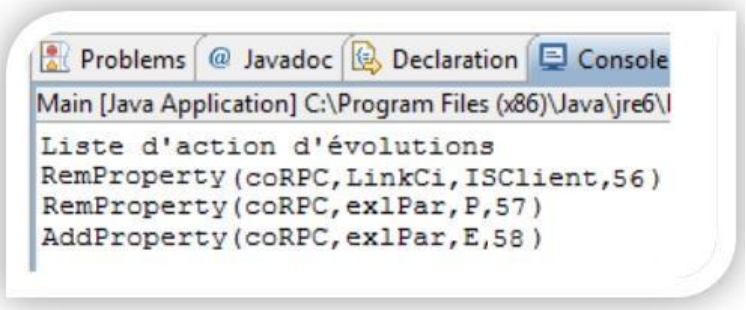


Figure 5.21: Exemple de liste d'action d'évolution sous format Praxis

La liste d'action de la figure 5.21 est appliquée sur la séquence d'action du modèle en PRAXIS à fin de le faire évoluer :

a) En supprimant l'action `AddProperty(coRPC,LinkCi,ISClient,49)` du modèle Praxis en utilisant l'action d'évolution `RemProperty(coRPC,LinkCi,ISClient,56)`.

b) En supprimant l'action `AddProperty(coRPC,exlpar,p,23)` du modèle Praxis en utilisant l'action d'évolution `RemProperty(coRPC,exlpar,p,57)`.

c) En ajoutant l'action `AddProperty(coRPC,exlpar,E,58)` au modèle Praxis en utilisant l'action d'évolution `AddProperty(coRPC,exlpar,E,58)`.

3. En utilisant la règle de vérification de cohérence pour la propriété exclusivité du connecteur (**RPC**) ainsi que les invariants, la vérification de la cohérence génère la liste d'incohérence suivante :

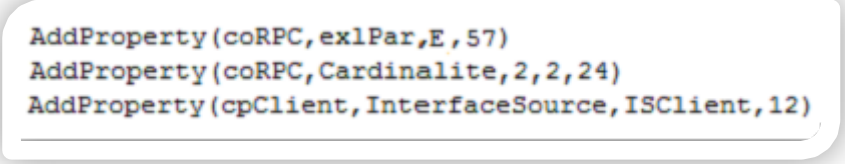


Figure 5.22: liste d'action source d'incohérences

La nouvelle valeur de la propriété *exlPar* est exclusive *E* présentée par la première action dans la figure 5.22 c.-à-d. que le connecteur **RPC** ne peut avoir qu'un seul lien source et un seul lien cible connecté à ces interfaces source et cible correspondante. Alors que la valeur de la propriété Cardinalité est *(2,2)*. Ce qui est contradictoire avec la définition de la propriété exclusivité. La dernière action correspond à l'incohérence provoquée par la vérification de l'invariant relatif à l'élément architectural *interface*, qui dit qu'une *interface* ne doit pas être libre, alors que l'interface cible du composant *Client* n'est connectée à aucune autre interface.

4. Proposition du plan de réparation :

En utilisant les règles de solutions correspondantes aux incohérences trouvées, le plan de réparation suivant est proposé :

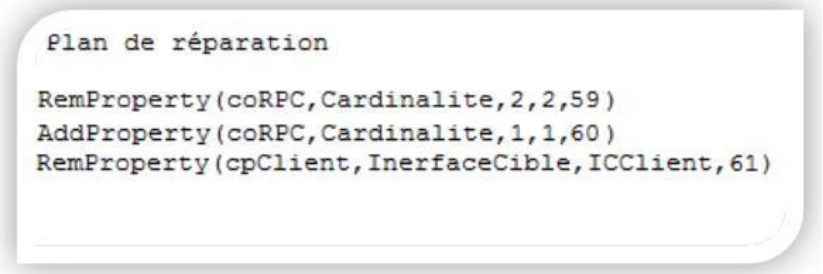


Figure 5.23: Plan de réparation

Le plan de réparation présenté dans la figure 5.23 propose de modifier la valeur de la propriété sémantique cardinalité par la nouvelle valeur *(1,1)* pour être en cohérence avec la valeur E de la propriété Exclusivité. La dernière action correspond à la suppression de l'interface cible du composant *Client* à fin de respecter l'invariant relatif à l'élément architectural *interface*.

Exemple d'évolution pour xADL :

Dans l'exemple ci-dessus nous voulons rendre le connecteur **SecurityManager** *exclusif*. Ceci implique que les deux interfaces source et cible du connecteur **SecurityManager** interagissent uniquement entre elles. Cette évolution est exprimée par l'opération d'évolution « **Modification de la valeur de la propriété sémantique Exclusivité-Partage du connecteur SecurityManager**». Dans ce qui suit nous détaillons cette évolution en précisant les différentes spécifications et en l'exécutant via le modèle IMoSCM.

Spécification de l'évolution :

1. Les propriétés sémantiques ainsi que les invariants associés aux éléments architecturaux, nous prenons ceux présentés dans le tableau 5.10.
2. Pour les stratégies d'évolutions nous prenons les stratégies associées aux éléments architecturaux à faire évoluer présentées dans la section 5.3.1.3.
3. Pour les règles de cohérences à vérifier ainsi que les solutions correspondantes aux incohérences trouvées nous prenons ceux présentées dans la section 5.2.1.2.1.
4. Les règles de transformation du modèle, à faire évoluer vers praxis et inversement sont présentés dans la section 5.3.1.4.

Exécution de l'évolution et vérification de cohérence :

Après spécification de l'évolution, le concepteur sélectionne l'opération d'évolution à appliquer sur l'architecture. Cette dernière est interprétée par le langage DTD ***<update Name= »exlPar » type= »PropSem » parent= »SecurityQuery » valeur= »E »/>*** comme montrer dans la figure 5.19 :

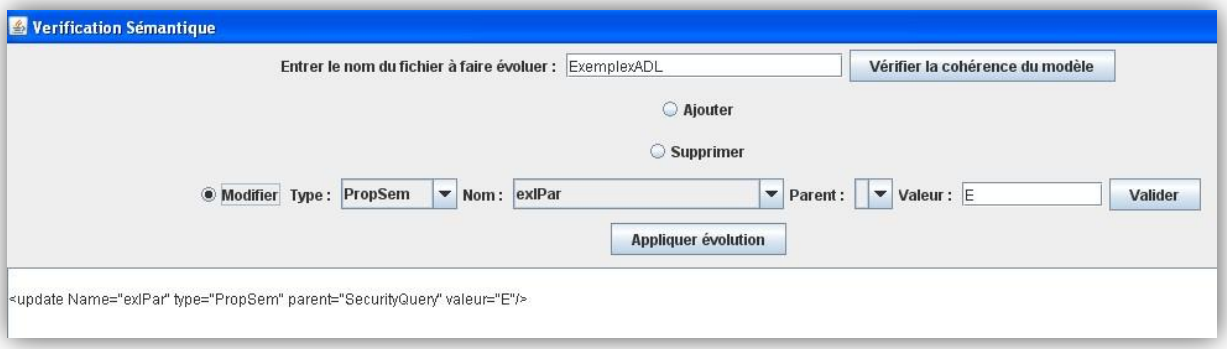


Figure 5.24: Opération d'évolution en DTD

En cliquant sur le bouton **appliquer évolution :**

1. L'architecture *Client/Serveur* en *XML* est transformée en *PRAXIS* en utilisant les règles de transformations. Le tableau 5.12 ci-dessous présente un exemple de transformation du composant *Client*, du connecteur *RPC* ainsi que la connexion entre eux comme suit :

Elément modèle	XML	Praxis
Composant *Client*	<component id = "Client"> <description>Client</description> <interface id = "Client:int1" > <description>Interface (in)</description> <direction>in</direction> </interface> </component>	Create(cpClient,Composant,1) AddProperty(cpClient,Name,Client,2) AddProperty(cpClient,InterfaceSource,ISClient:int1,3)
Connecteur *RPC*	<connector id = "RPC"> <description>RPC</description> <interface id = "RPC:in"> <description>interface RPC in</description> <direction>in</direction> </interface> <interface id = "RPC:out"> <description>interface RPC out</description> <direction>out</direction> </interface> <propriete exlPar="P" Cardinalite="2,2"/> </connector>	Create(coRPC,Connecteur,16) AddProperty(coRPC,Name,RPC,17) AddProperty(coRPC,InterfaceSource,ISRPC:in,18) AddProperty(coRPC,InterfaceCible,ICRPC:out,19) AddProperty (coRPC,exlPar,P,20) AddProperty (coRPC,Cardinalite,2,2,21)
Connecxion	<link id = "link1" > <description>premier lien</description> <point> <anchor href = "#Client:int1"/> </point> <point> <anchor href = "#RPC:out"/> </point> </link>	AddProperty (coRPC,LinkCi,ISClient,40)

Tableau 5.12: Transformation de XML vers PRAXIS de l'ADL xADL

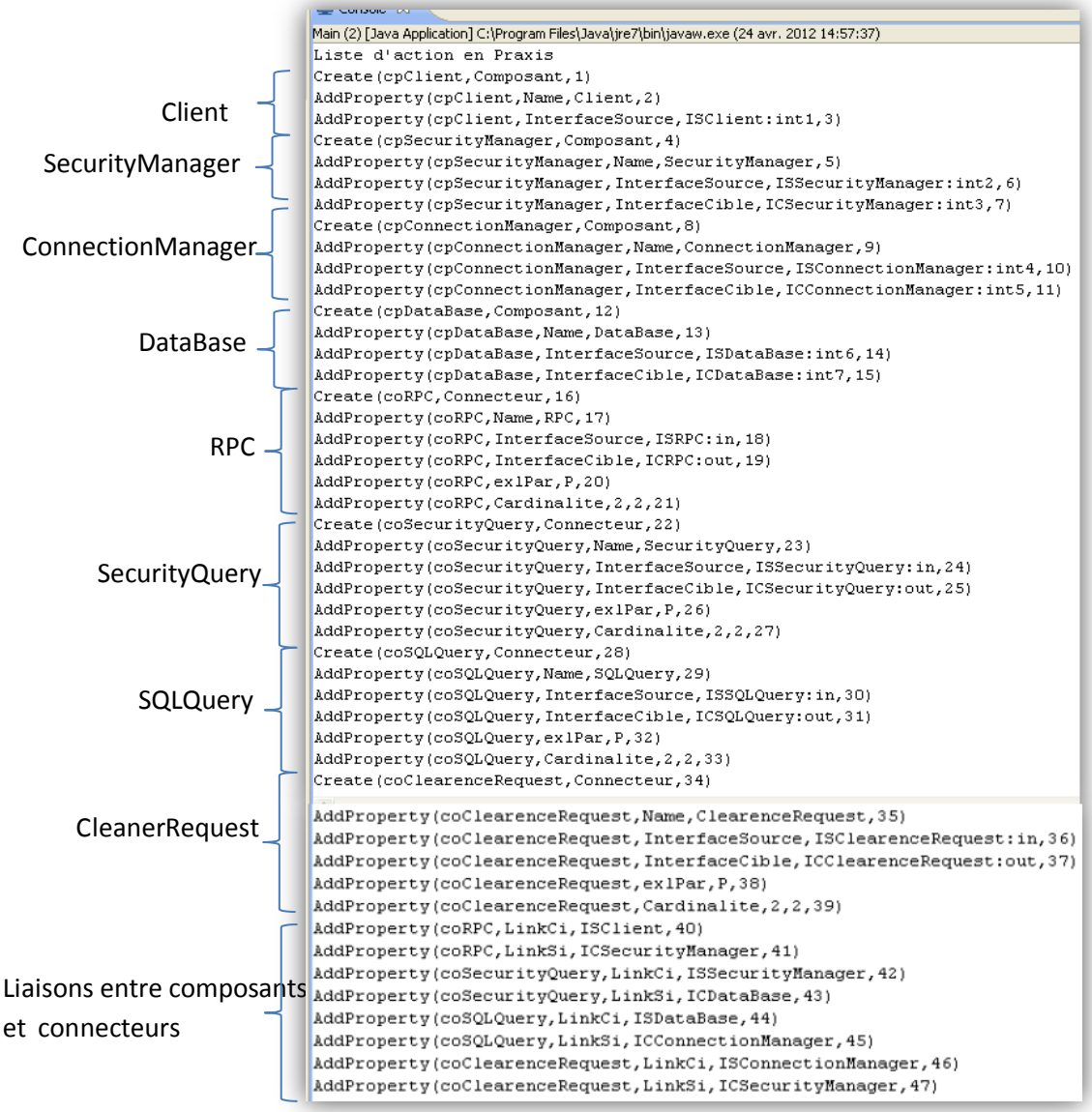


Figure 5.25: Modèle xADL sous format Praxis

2. Le modèle d'évolution constitué de l'opération **Update** est interprété en utilisant la stratégie d'évolution de **Modification** :

   a) Supprimer ancienne Propriété sémantique.

   b) Ajouter une nouvelle propriété sémantique avec la nouvelle valeur.

Cette stratégie est transformée en PRAXIS, en utilisant les règles de transformations, comme montrer dans la figure 5.26 ci-dessous :

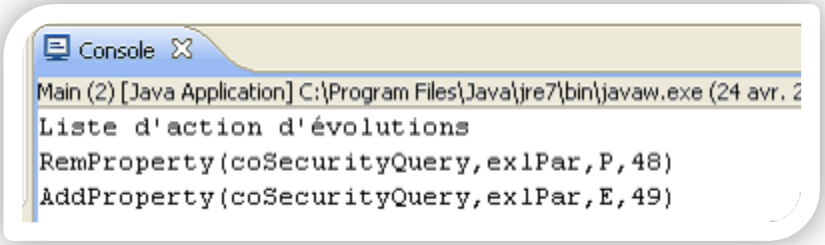


Figure 5.26: Liste d'action d'évolution

La liste d'action de la figure 5.26 est appliquée sur la séquence d'action du modèle en PRAXIS à fin de le faire évoluer :

a) En supprimant l'action `AddProperty(coSecurityQuery,exlpar,p,26)` *du modèle Praxis* en utilisant l'action d'évolution `RemProperty(coSecurityQuery,exlpar,p,48).`

b) En ajoutant l'action `AddProperty(coSecurityQuery,exlpar,E,49)` au modèle Praxis en utilisant l'action d'évolution `AddProperty(coSecurityQuery,exlpar,E,49).`

3. En utilisant la règle de vérification de cohérence pour la propriété exclusivité du connecteur (**SequrityManager**) ainsi que les invariants, la vérification de la cohérence génère la liste d'actions sources d'incohérences suivantes :

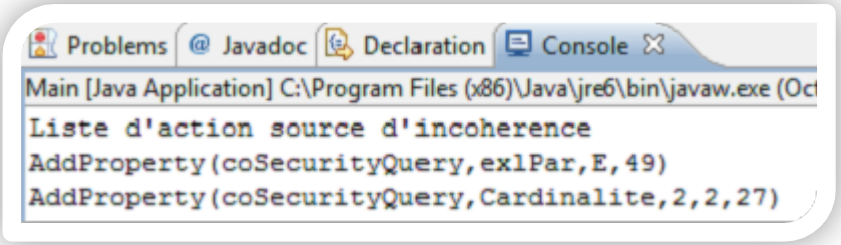


Figure 5.27: Liste d'action source d'incohérence

La nouvelle valeur de la propriété exlPar est exclusive *(E)* présentée par la première action dans la figure 5.27 c.-à-d. que le connecteur **SecutityManager** ne peut avoir qu'un seul lien source et un seul lien cible connecté à ces interfaces source et cible correspondante. Alors que la valeur de la propriété Cardinalité est *(2,2)*. Ce qui est contradictoire avec la définition de la propriété exclusivité.

4. Proposition du plan de réparation :

En utilisant les règles de solution correspondantes aux incohérences trouvées, le plan de réparation suivant est proposé :

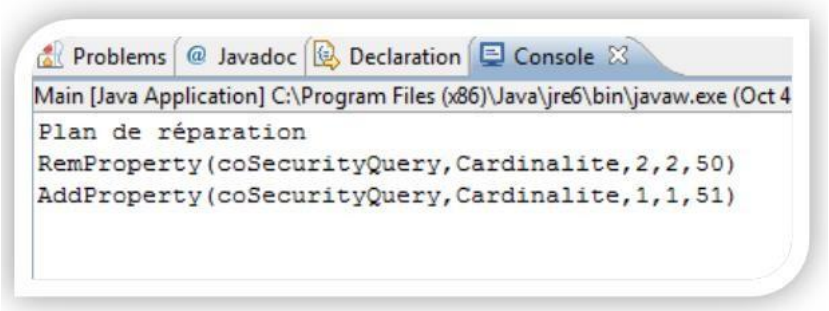


Figure 5.28: Plan de réparation

Le plan de réparation présenté dans la figure 5.28 propose de modifier la valeur de la propriété sémantique cardinalité par la nouvelle valeur *(1,1)* pour être en cohérence avec la valeur *(E)* de la propriété Exclusivité.

Dans les deux exemples ci-dessus le concepteur a le choix d'appliquer ou non le plan de réparation proposé. Dans le cas où il choisit de ne pas l'appliquer, l'architecture initiale avant évolution est récupérée et présentée au concepteur. Le code source de ces exemples est donné dans l'annexe A.

5.5 Pérformancés:

5.5.1 Introduction:

Dans cette section, les unités de mesure utilisées ainsi que la méthode adoptée pour le calcul de performances sont présentées. Nous exposons ensuite les performances et analyses pour les deux ADLs ACME [36] et xADL [49 ,51].

5.5.2 Les exemples d'expérimentation :

Le calcul de performances est fait, sous Windows XP, pour les deux ADL ACME [36] et xADL [49 ,51] sur un exemple d'architecture client-serveur. Pour ce faire, nous avons pris l'exemple d'une architecture client-serveur présentée dans [9] auquel nous avons ajouté des éléments architecturaux (*composants, connecteur, interface et liaisons*) pour obtenir plusieurs architectures allant de 10 jusqu'à 100 éléments architectural.

Pour chaque architecture est appliquées une suite d'opérations d'évolutions, comme opération d'évolution : supprimer un composant, supprimer un lien, modifier une propriété sémantique…etc. Le nombre de ces opérations d'évolutions est variable allant de 2 jusqu'à 20 opérations d'évolutions appliquées pour chaque architecture.

5.5.3 Paramètres d'influences :

Quelques paramètres peuvent avoir un impact sur le temps d'exécution ainsi que sur le résultat de la vérification de la cohérence :

- Taille du modèle : la taille du modèle à un impact sur le temps de vérification de la cohérence. Une plus grande taille implique une plus longue vérification. De plus une plus grande taille implique plus de possibilité de modification, en termes d'opération d'évolution et aussi comme possibilité de solution pour les incohérences trouvées.
- Nature du modèle : la nature du modèle à une influence sur le comportement des propriétés sémantiques, qui visent des éléments particuliers et des configurations du modèle. Aussi la nature et le nombre de ces propriétés sémantiques ont un impact sur le temps de vérification de cohérence. Un plus grand nombre de propriétés implique une plus longue vérification de cohérence.

- Nature d'opération d'évolution : la nature des opérations d'évolutions, en d'autre terme le type de changements effectués, peuvent être des changements faciles à vérifier ou bien des changements complexes qui demandent plus de temps de vérification de cohérence. Aussi le nombre d'incohérence dépend du choix et de la nature de ces changements. L'utilisateur peut choisir des opérations d'évolutions qui conduisent vers plus d'incohérences ou bien qui corrigent les incohérences provoquées par d'autres opérations d'évolutions.
- Nombre d'opération d'évolution : le nombre d'opération d'évolution à un impact sur le temps de vérification de la cohérence. Un plus grand nombre implique une plus longue vérification.

Afin de voir les différents impacts de ces paramètres, nous avons choisi deux types de modèle ACME [36] et xADL [49 ,51], nous avons fait varier la taille du modèle ainsi que le nombre d'opération d'évolution. La vérification de cohérence est faite pour les deux propriétés sémantiques Exclusivité/Partage et Cardinalité/Cardinalité inverse.

5.5.4 Calcul de performance et analyse :

Les figures 5.29, 5.30, 5.31 montrent l'impact de la taille du modèle sur le temps de vérification d'incohérence ainsi que le temps d'évolution. Avec plus d'éléments architecturaux, le processus de vérification de la cohérence met plus temps à parcourir tous les éléments du modèle à vérifier. De même le processus d'évolution met plus de temps à parcourir tous les éléments du modèle afin de trouver l'élément à supprimer ou bien à mettre à jour.

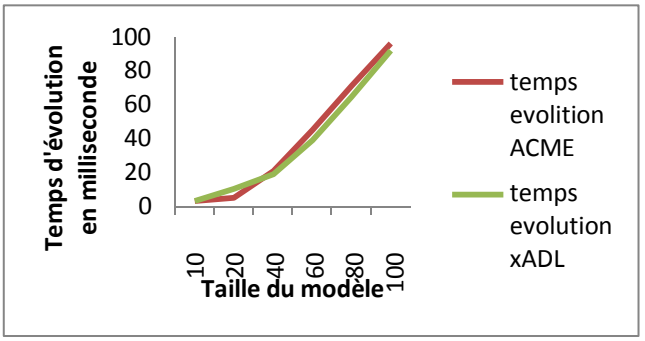


Figure 5.29: Temps d'évolution en fonction de la taille du modèle initial en nombre d'élément architectural

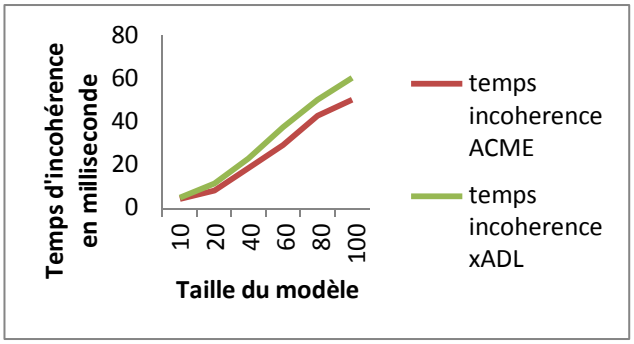


Figure 5.30: Temps de vérification de cohérence en fonction de la taille du modèle initial en nombre d'élément architectural

Le temps d'évolution est légèrement plus élevé pour le modèle ACME [36], car le nombre d'action en Praxis [12] est plus élevé que celui du modèle xADL [49 ,51], voir le tableau 5.13. Donc le processus d'évolution met plus de temps à parcourir toutes les actions en Praxis du modèle ACME afin de trouver l'élément à supprimer ou bien à mettre à jour. Le temps d'incohérence est légèrement plus élevé pour le modèle xADL, car la nature du modèle à une influence sur le comportement des propriétés sémantiques, qui visent des éléments particuliers et des configurations du modèle. Aussi la nature et le nombre de ces propriétés sémantiques ont un impact sur le temps de vérification de cohérence. Un plus grand nombre de propriétés implique une plus longue vérification de cohérence.

Nombre d'élement Architectural	**Nombre d'action élémentaire en Praxis équivalent pour ACME**	**Nombre d'action élémentaire en Praxis équivalent pour xADL**
10	57	55
20	109	107
40	219	204
60	360	322
80	469	432
100	552	525

Tableau 5.13: Transformation vers Praxis en nombre d'action élémentaire

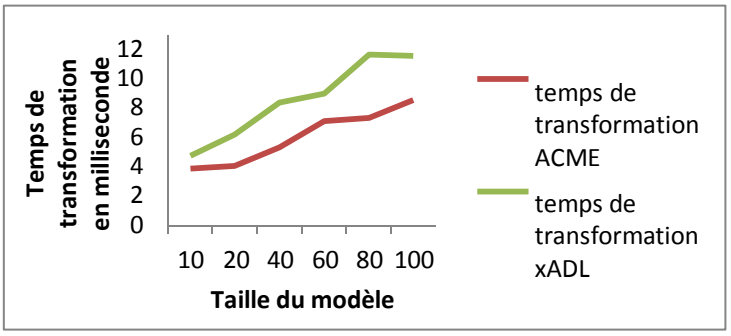


Figure 5.31: Temps de transformation en fonction de la taille du modèle initial en nombre d'élément architectural

Le temps de transformation vers Praxis est proportionnel à la taille du modèle. Chaque élément architectural est transformé en une suite d'action en Praxis, donc plus d'éléments architecturaux plus de temps de transformation. Le temps de transformation pour xADL est légèrement plus élevé que celui d'ACME, car le modèle en XML de xADL est plus long que celui d'ACME ce qui demande plus de temps de traitement et donc de transformation.

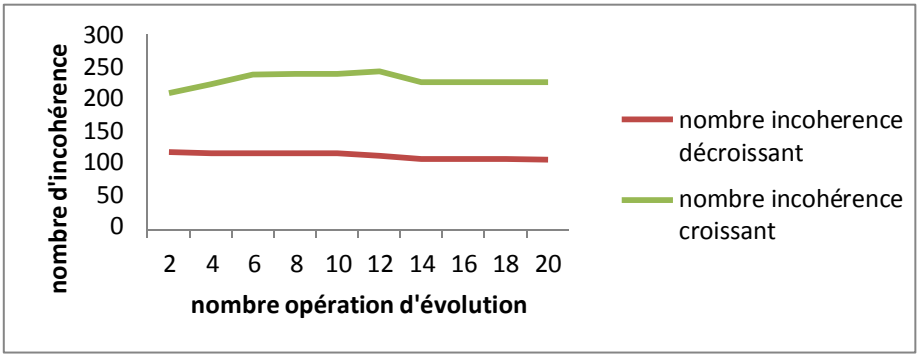


Figure 5.32: Nombre d'incohérence en fonction du nombre d'opération d'évolution

La figure 5.32 montre deux exemples sur le nombre d'incohérence selon le nombre et la nature d'opération d'évolution. Pour cela nous avons choisi un type de modèle avec une taille de 100 éléments. Le nombre d'incohérence décroissant revient à la nature des opérations d'évolutions choisies qui corrigent les incohérences détectées ou bien provoquées par d'autres opérations d'évolutions. Le nombre d'incohérence croissant revient à la nature des opérations d'évolutions choisies qui provoquent plus d'incohérences.

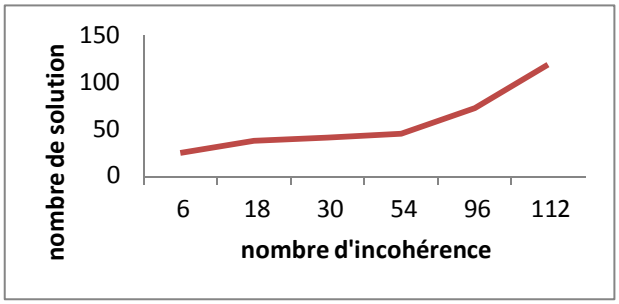


Figure 5.33: nombre de solution en fonction du nombre d'incohérence

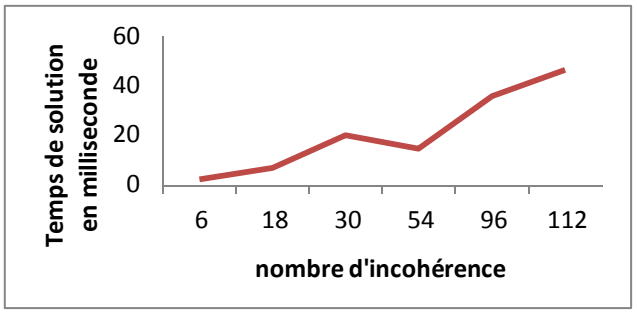


Figure 5.34: temps de solution en fonction du nombre d'incohérence

Les figures 5.33 et 5.34 ci-dessus montrent que le nombre de solution en terme d'action en Praxis ainsi que le temps de solution sont proportionnels au nombre d'incohérences détectées. Ceci dépend du processus du plan de réparation utilisé et plus précisément des règles de solutions prédéfinies. Le processus du plan de réparation et les règles de solutions peuvent être optimisés en proposant un plan de réparation optimal avec moins de nombre et temps de solution.

5.6 Conclusion :

A travers ce chapitre nous avons présenté une validation de nos travaux sur la vérification de cohérence sémantique :

• Nous avons d'abord donné une présentation générale de la manière de valider.

• Dans la deuxième partie, nous avons présenté l'implémentation de la partie indépendante du méta modèle.

• Dans la troisième partie, nous avons présenté l'implémentation de la partie dépendante du méta modèle.

• La quatrième partie présente un exemple appliqué pour chacun des deux ADLs ACME [36] et xADL[49 ,51].

• La cinquième partie et la dernière est consacrée au calcul de performances fait sur un ensemble d'architecture client-serveur allant de 10 jusqu'à 100 éléments architecturaux. Pour chaque architecture sont appliquées une suite d'opérations d'évolutions allant de 2 jusqu'à 20 opérations. Nous avons présenté des paramètres qui peuvent avoir un impact sur le temps d'exécution ainsi que sur le résultat de la vérification de la cohérence : taille du modèle, nature du modèle, nature d'opération d'évolution, nombre d'opération. Pour le calcul de performance nous avons fait varier la taille du modèle ainsi que le nombre d'opération d'évolution. La vérification de cohérence est faite pour les deux propriétés sémantiques Exclusivité/Partage et Cardinalité/Cardinalité inverse.

En analysant les performances nous avons conclus que :

- La taille du modèle à faire évoluer à un impact sur le temps d'incohérence ainsi que le temps d'évolution. Une plus grande taille implique une plus longue vérification.
- La nature du modèle à faire évoluer à une influence sur le comportement des propriétés sémantiques, qui visent des éléments particuliers et des configurations du modèle. Donc un impact sur le temps de vérification de cohérence.
- Le temps de transformation vers Praxis est proportionnel à la taille du modèle. Donc plus d'éléments architecturaux plus de temps de transformation.
- Le nombre de solution en terme d'action en Praxis ainsi que le temps de solution sont proportionnels au nombre d'incohérences détectées. Ceci dépend du processus du plan de réparation utilisé et plus précisément des règles de solutions prédéfinies
- Le nombre d'incohérences dépend de la nature des opérations d'évolutions choisies qui peuvent corriger les incohérences détectées ou bien provoquer plus d'incohérences.

Chapitre 6 :
Conclusion & perspective

Nous avons exposé dans ce travail une problématique liée à la gestion de l'évolution des modèles indépendamment de leurs métas modèles, en assurant une cohérence sémantique des modèles résultants. Pour cela, nous avons proposé une solution pour une gestion automatique de la cohérence sémantique indépendamment de tout méta modèle lors de l'évolution des modèles.

En se basant sur les différentes définitions et travaux trouvés dans la littérature, nous nous sommes positionnés par rapport à l'évolution et opération d'évolution. Nous considérons l'évolution comme un ensemble des changements et des transformations d'un système en utilisant les opérations d'évolutions d'ajout, suppression, modification et les activités de maintenance et d'adaptation comme étant des cas particuliers de l'évolution. Nous utilisons la technique de *transformation exogène* de modèle en transformant n'importe qu'elle modèle en formalisme de praxis à fin d'assurer l'indépendance par rapport au méta modèle. Nous avons constaté que tous les travaux étudiés traitent l'évolution statique d'une façon automatique en se basant généralement sur des contraintes et invariants. Quelques un utilisent des techniques de versionnement et des règles de transformations de modèle. La plus part s'intéressent à la cohérence structurelle. Par contre SAEV [9] aborde la cohérence sémantique mais uniquement dans le contexte des ADL [6]. Nous avons remarqué que seul PRAXIS [12] propose une solution traitant la cohérence indépendamment de tout méta modèle mais ne traite pas la cohérence sémantique. Notre proposition vient compléter ce manque d'une part en traitant la cohérence sémantique et d'une autre part proposer une solution indépendamment de tout méta modèle. Nous avons classé les différents travaux effectués sur la gestion d'évolution selon deux types de modélisation : orientée objet et orientée composant. Nous avons présenté un bilan composé de trois parties : la première partie représente deux comparaisons, une sur les différents travaux relatifs à la gestion d'évolution et une autre sur les différents travaux relatifs à la gestion de la cohérence. La deuxième partie représente un ensemble de constatations sur les limites des différents travaux étudiés. La troisième partie et la dernière présente nos objectifs à atteindre.
En se basant sur notre étude et la comparaison faite sur les différents travaux effectués sur l'évolution des modèles, nous avons défini les objectifs que nous voulons atteindre afin de proposer un modèle d'évolution qui doit assurer les points suivant :

1. Gérer l'évolution ainsi que son impact indépendamment de tout méta modèle.
2. Traiter la cohérence sémantique indépendamment de tout méta modèle.

3. Proposer un plan de réparation et une correction automatique en cas d'incohérence.

Nous avons présenté notre modèle IMoSCM, comme solution à la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs métas modèles, en assurant une cohérence sémantique des modèles. Dans IMoSCM le modèle en praxis représente toute architecture logicielle (*orientée composant, orientée objet*) susceptible d'évoluer. Nous avons introduit ce concept pour pouvoir modéliser toutes architectures logicielles, sous forme d'actions élémentaires en se basant sur le formalisme de Praxis. Ce qui permet une gestion d'évolution indépendamment du méta modèle de l'architecture à faire évoluer. Dans IMoSCM des règles de détections d'incohérences sémantiques sont prédéfinies, afin de permettre la détection des incohérences sémantiques après évolution en utilisant les propriétés sémantiques proposées dans SAEV [9]. Dans le but de corriger les incohérences trouvées, IMoSCM propose une suite d'action élémentaire sous forme de PRAXIS [12], comme un plan de réparation, à partir de la liste des incohérences trouvées, ainsi qu'une correction automatique des incohérences. Nous avons implémenté notre solution pour deux métas modèles différents, afin de prouver que notre solution est indépendante de tout méta modèle. Pour cela, l'implémentation est décomposée en deux parties: une partie indépendante du méta modèle et une partie dépendante du méta modèle.

Pour la partie dépendante, deux implémentations ont été effectuées pour les métas modèles xADL [49 ,51] et ACME [36] en utilisant le langage JAVA[50]. Cette partie doit être réalisé par un spécialiste de modélisation qui doit élaborer un *langage d'évolution* du modèle initial à faire évoluer ainsi que les stratégies d'évolutions, assurer la transformation du modèle d'évolution vers praxis et le modèle initial vers praxis et inversement. Une fois réalisés un utilisateur introduit le modèle à faire évoluer ainsi qu'une suite d'opération d'évolution, applique les opérations d'évolution et lance la vérification de la cohérence. L'application des opérations d'évolutions et la vérification de la cohérence représentent la partie indépendante du méta modèle.

Nous avons présenté une validation de nos travaux sur la vérification de cohérence sémantique :

• Nous avons d'abord donné une présentation générale de la manière de valider.
• Dans la deuxième partie, nous avons présenté l'implémentation de la partie indépendante du méta modèle.

• Dans la troisième partie, nous avons présenté l'implémentation de la partie dépendante du méta modèle.

• La quatrième partie présente un exemple appliqué pour chacun des deux ADLs ACME [36] et xADL[49 ,51].

• La cinquième partie et la dernière est consacrée au calcul de performances fait sur un ensemble d'architecture client-serveur allant de 10 jusqu'à 100 éléments architecturaux. Pour chaque architecture sont appliquées une suite d'opérations d'évolutions allant de 2 jusqu'à 20 opérations. Nous avons présenté des paramètres qui peuvent avoir un impact sur le temps d'exécution ainsi que sur le résultat de la vérification de la cohérence : taille du modèle, nature du modèle, nature d'opération d'évolution, nombre d'opération. Pour le calcul de performance nous avons fait varier la taille du modèle ainsi que le nombre d'opération d'évolution. La vérification de cohérence est faite pour les deux propriétés sémantiques Exclusivité/Partage et Cardinalité/Cardinalité inverse.

En analysant les performances nous avons conclus que :

- La taille du modèle à un impact sur le temps d'incohérence ainsi que le temps d'évolution. Une plus grande taille implique une plus longue vérification.
- La nature du modèle à une influence sur le comportement des propriétés sémantiques, qui visent des éléments particuliers et des configurations du modèle. Donc un impact sur le temps de vérification de cohérence.
- Le temps de transformation vers Praxis est proportionnel à la taille du modèle. Donc plus d'éléments architecturaux plus de temps de transformation.
- Le nombre de solution en terme d'action en Praxis ainsi que le temps de solution sont proportionnels au nombre d'incohérences détectées. Ceci dépend du processus du plan de réparation utilisé et plus précisément des règles de solutions prédéfinies
- Le nombre d'incohérences dépend de la nature des opérations d'évolutions choisies qui peuvent corriger les incohérences détectées ou bien provoquer plus d'incohérences.

Perspectives

Le travail présenté dans ce document comprend la proposition d'un modèle, nommé IMoSCM, comme solution à la problématique liée à la gestion de l'évolution des modèles indépendamment de leurs métas modèles, en assurant une cohérence sémantique des modèles résultants. Nous avons proposé une transformation, des différents modèles susceptibles d'évoluer, sous forme d'actions élémentaires en se basant sur le formalisme de Praxis [12]. Ce qui permet une gestion d'évolution indépendamment du méta modèle de l'architecture à faire évoluer. Nous avons utilisé des règles de détection d'incohérence sémantique pour la détection des incohérences sémantiques après évolution en utilisant les propriétés sémantiques proposées dans SAEV [9].

Ce travail peut être poursuit en plusieurs perspectives de recherche :

Implémentation des autres propriétés sémantiques :

Par faute de temps, nous avons implémenté que deux propriétés sémantiques, Exclusivité /Partage et Cardinalité/Cardinalité inverse. Les autres propriétés sémantiques, Dépendance/Indépendance et Prédominance/Non prédominance présentées dans [9], peuvent être prises en compte en ajoutant les sous classes correspondantes et en implémentant les méthodes de vérification et de détection de cohérence pour chaque propriété sémantique.

Prise en compte des opérations d'évolution plus complexes :

Nous avons appliqué et implémenté dans notre solution les opérations d'évolution simples et basiques : l'ajout, la suppression et la modification. D'autres opérations d'évolution plus complexes, tels que la *fusion*, le *transfert*, *déplacer etc.*, peuvent être appliquées en spécifiant les stratégies d'évolutions associées à ces opérations d'évolutions pour chaque élément du modèle.

Proposer et enrichir avec d'autres propriétés sémantiques :

Nous avons utilisé les propriétés sémantiques définies dans SAEV [9]. D'autres propriétés sémantiques peuvent être proposées afin de mieux gérer la cohérence sémantique. Les propriétés sémantiques définies dans SAEV [9] sont plus liées au nombre de liaison possible entre deux composants. Nous pouvons proposer d'autres propriétés sémantiques non

seulement au niveau connecteur, pour le cas des architectures logicielles à base de composant, mais aussi au niveau composant. Ces propriétés sémantiques peuvent véhiculer des informations utiles pour la cohérence sémantique d'une architecture.

Amélioration du plan de réparation :

Le plan de réparation proposé dans notre implémentation peut être amélioré, en proposant un plan de réparation le plus optimale possible en consommant moins de temps et en corrigeant le maximum d'incohérence. Pour cela, Nous proposons d'ajouter dans les actions de causes d'incohérences des informations qui serviraient au choix de la solution de l'incohérence en question. Ou bien, au moment de la construction du plan de réparation en parcourant la liste des causes d'incohérences, proposer une solution pour chaque incohérence en prenant en compte les solutions précédemment trouvées. Autrement dit, proposer une solution à l'incohérence numéro *x* en prenant en compte le plan de réparation construit au niveau de la *x-1* incohérence. Ce qui peut réduire la taille du plan de réparation ainsi que le temps de son application et en corrigeant le maximum d'incohérence.

Versionnement :

Nous avons utilisé un modèle d'évolution sous forme de liste d'action d'évolution en Praxis, pour représenter les changements appliqués sur le modèle à faire évoluer. Ce modèle représente la différence entre le modèle initial et le modèle après évolution, ce qui nous permet de revenir à l'état initial du modèle en annulant les actions dans le modèle d'évolution. Nous pouvons pour cela sauvegarder les différents modèles d'évolution appliqués sur le modèle à faire évoluer et les utilisés pour revenir aux versions précédentes. Pour cela, différentes techniques existent et présentées dans le chapitre 2.

Bibliographie

[1] Mounir GRARI. Thèse **Principes et états de l'art de l'approche MDA et applications pour des plates-formes PHP orienté 3-tier.** FACULTÉ DES SCIENCES OUJDA[PDF].

[2] Xavier Blanc. **MDA en action, livre Ingénierie logicielle guidée par les modèles**, 1ère édition 270 pages, 2005, ÉDITIONS EYROLLES, ISBN : 2-212-11539-3.

[3] Fleurey, Steel, Baudry. **Validation in Model-Driven Engineering: Testing Model Transformations**, International Conference on the UML, Lisbon, Portugal 2004, danshttp://www.metamodel.com/wisme-2004/papers.html [En ligne].

[4] Jean BÉZIVI, Olivier GERBÉ. **Towards Precise Definition of the OMG/MDA Framework.** In Proceedings of the 16th International Conference on Automated Software Engineering, 2001[PDF].

[5]. M. Boasson& H. Signaalapparaten, **The Artistry of Software Architecture**. IEEESoftware, pp. 13-16, November 1995.

[6] N. Medvidovic and R. N. Taylor.**A classification and comparison framework for software architecture description languages**. In IEEE Transactions on Software Engineering, volume 26, page 23, janvier 2000.

[7] C.Tibermacine. **Contractualisation de l'´évolution architecturale de logiciels à base de composants : une approche pour la préservation de la qualité**. Université de Bretagne Sud. 20 Octobre 2006.

[8] O. Baraise. **Construire et Maitriser l'Evolution d'une Architecture Logicielle à base de Composants**. Ecole doctorale SPI Lille. Novembre 2005.

[9] Nassima SADOU-HARIRECHE. **Evolution Structurelle dans les Architectures Logicielles à base de Composants**. l'UFR Sciences &Techniques, Université de Nantes. 2007.

[10] M.Cartier& A .Colovic. **Co-évolution et adaptabilité des réseaux : études de cas et simulation**. Université de Paris Dauphine. Juin 2006.

[11] C. Nebut et J.-R. Falleri. **Transformation de modèle**. Université de Montpellier 2. 2004

[12] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc, and RedaBendraou. **Towards Automated Inconsistency Handling in Design Models**.LIP6, UPMC Paris Universities, France 2010.

[13] Douglas C. **Schmidt. Model-DrivinIngineering.**Vanderbilt University . February 2006

[14] http://dictionnaire.reverso.net/francais-definition/%C3%A9volution.

[15] M. AHMED-NACER. **UN MODELE DE GESTION ET D'EVOLUTION DE SCHEMA pour les Bases de Données de Génie Logiciel**. INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE. Juillet 1994.

[16] Penny, D. and Stein, J. (1987). **Class modification in the Gemstone Object- Oriented DBMS**. In Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages, pages 111–117, Orlando.

[17] Banerjee, J., Kim, W., Kim, H. J., and Korth, H. F. (1987b).**Semantics and implementation of schema evolution in Object Oriented databases**. In Proceeding of the ACM SIGMOD conference, pages 311–323, San Francisco.

[18] Zdonik, S. B. (1986). **Version management in an object-oriented database**. In Proc IFIP2.4 Workshop on Advanced Programming Environments, Trondheim - Norway.

[19] Andany, J., Leonard, M., and Palisser, C. (1991). **Management of schema evolution in databases**. In Proc. of the 17th Conf. Very Large Data Bases, pages 161–170,Barcelona.

[20]Jean-Marc Hick, Jean-Luc Hainaut, Vincent Englebert, Didier Roland, Jean Henrard. **Stratégies pour l'évolution des applications de bases de données relationnelles : l'approche DB-MAIN**. Institut d'Informatique - Facultés Universitaires de Namur, Belgique. 1999.

[21]C.Favre**. Évolution de schémas dans les entrepôts de données : mise à jour de hiérarchies de dimension pour la personnalisation des analyses**. Université Lumière Lyon 2 École Doctorale Informatique et Information pour la Société. Décembre 2007.

[22]Barais O., Duchien L. : Transat : **maîtriser l'évolution d'une architecture logicielle**. L'OBJET 10, 2-3 (2004), 103–116.

[23] O. LE GOAER. **Styles d'évolution dans les architectures logicielles**. Université de Nantes. Octobre 2009.

[24] ChoukiTibermacine, R´egisFleurquin, and Salah Sadou.**On-demand quality-oriented assistance in component-based software evolution**. In Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06), Vasteras, Sweden, June 2006.Springer LNCS.

[25] M. Elaasar and L. Briand.**An Overview of UML Consistency Management**. Technical Report SCE-04-18, pages 1–51, 2004. 6.

[26] Huaxi (Yulin) Zhang, Christelle Urtado, Sylvain Vauttier. Dedal : **un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants**. LGI2P / Ecole des Mines d'Alès. 2009.

[27] Mohammed Karim GUENNOUN. **Architectures Dynamiques dans le Contexte des Applications à Base de Composants et Orientées Services**. Laboratoire d'Analyse et d'Architecture des Systèmes, LAAS-CNRS. Mars 2007.

[28] Hervé Verjus, SoranaCîmpan, Ilham Alloui, Flavio Oquendo. **Gestion des architectures évolutives dans ArchWare**. Université de Savoie B.P. 806, F - 74016 Annecy Cedex.2006.

[29] Jean-François Rolland. **Développement et validation d'architectures dynamiques**. l'Université Toulouse III - Paul Sabatier. décembre 2008.

[30] Djamal BENNOUAR. **UNE APPROCHE INTEGREE POUR L'ARCHITECTURE LOGICIELLE**. L'ECOLE NATIONALE SUPERIEURE D'INFORMATIQUE D'ALGER. Avril 2009.

[31] R. ALLEN et D. GARLAN. **The wright architectural specification language**. Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science,1996.
R. ALLEN.

[32] R. ALLEN. **A formal Approach to Software Architecture Description**. Thèse de Doctorat, Carnegie Mellon University, Technical Report Number : CMU-CS-97-144,1997.

[33] J.MAGEE et J.KRAMER. **Dynamic structure in software architecture**. Proceedings of ACM SIGSOFT'96: Fourth Symposium Foundation of Software engineering,
pages 3–14, 1996.

[34] J. MAGEE, N. DULAY, S. EISENBACH et J. KRAMER. **Specification and analysis of system architectures using rapide**. Proceedings of the fifth European Software Engineering conference, pages 336–355, Barcelona, Spain, 1995.

[35] N. MEDVIDOVIC, D.S. ROSENBLUM et R.N. TAYLOR. **A language and environment for architecture-based software development and evolution**. Proceeding of the 21st international conference on Software engineering, (ICSE'99), pages 44–53, 1999.

[36] D. GARLAN, R. MONROE et D.WILE. Acme :**An architecture description of component-bases systems**. Foundations of Component-Based Systems, Leavens Gray etSitaramanMurali (Réd.), pages 47–68, 2000.

[37] O. BARAIS**. Construire et Maîtriser l'Evolution d'une Architecture Logicielle à base de Composants**. Thèse de Doctorat, Université des Sciences et Technologies de Lille, 2005.

[38] O. BARAIS et L. DUCHIEN. Safarchie : **Maîtriser l'Évolution d'une architecture logicielle**. Langages, Modèles et Objets - Journées Composants - LMO 2004-JC 2004, L'objet, 10(2-3):103– 116, 2004.

[39] Object Management GROUP. **Uml 2.0 infrastructure specification, technical report**ptc/03-09-15.2003.

[40] A.V.D HOEK, M. RAKIC, R.ROSHANDEL et N. MEDVIDOVIC. **Taming architectural evolution**. Proceedings of the Sixth European Software Engineering Conference (ESEC) and the NinthACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9, pages 1–10,Vienna, Austria, 2005.

[41] R. ROSHANDEL, A. V.D.HOEK, M. MIKI-RAKIC et N. MEDVIDOVIC**. Mae-a system model and environment for managing architectural evolution**. ACM Transactions on Software Engineering and Methodology, 13(2).

[42] E.M. DASHOFY, A. van der HOEK et R.N. TAYLOR. **A highly-extensible, xml-based architecture description language**. Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), Amsterdam, Netherlands, 2001.

[43] E.M. DASHOFY et A. van der HOEK. **Representing product family architectures in an extensible architecture description language**. Proceedings of the International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, 2001.

[44] Antonio Cicchetti. **Difference Representation and Conflict Management in Model-Driven Engineering.** University di L'Aquila ViaVetoio, I-67100 L'Aquila, Italy. January 2008

[45] U. DAYAL, A. BUCHMMAN et D. MCCARTHY. **Rules are objects too: a knowledge model for an active object-oriented database systems**. Lecture Notes in Computer Science 334, pages 129–143, 1988.

[46] B. Ben Ammar, M. Tahar Bhiri, J. Souquières. **Schéma de refactoring de diagrammes de classes basé sur la notion de délégation.** LORIA - Nancy University, 2008.

[47] N. SADOU, D. TAMAZALIT et M. OUSSALAH. **How to manage uniformly software architecture at different abstraction levels**. *proceedings of the 24th ACM International conference on Conceptual Modeling (ER2005), pages 16–30, Klagenfurt, Austria, 2005.*

[48] N. SADOU, D. TAMAZALIT et M. OUSSALAH. **A unified approach for software architecture evolution at different abstraction levels**. International Workshop on Principles of Software Evolution September (IWPSE 2005) in association with 8th European Software Engineering Conference and 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), pages 65–70, Lisbon, Portugal, 2005.

[49] Dashofy, E.M., van der Hoek, A., Taylor, R.N.: **An infrastructure for the rapid development of xml-based architecture description languages**. In: Proceedings of the 24th International Conference on Software Engineering (ICSE2002) (2002).

[50] Eclipse FONDATIONS. http://www.eclipse.org.

[51] http://www.isr.uci.edu/projects/xarchuci/index.html ---official website of xADL.

Annexe A

Code source:

Nous présentons dans cette annexe le code source de l'exemple d'évolution cité dans la section 5.4, relative au modèle d'évolution et gestion de la cohérence IMoSCM, l'application est développée en JAVA sous ECLIPS.

```
/**
        * cette méthode permet la construction de l'opération d'évolution de
 * modification d'une propriété sémantique en DTD
 */
jbadd.addActionListener(new ActionListener(){

   public void actionPerformed(ActionEvent arg0) {
        String operation =opEvol.getText();
        String operationVerif="";
        EvolScrol.setVisible(true);

   if(modifier.isSelected()==true){
       operation=" \n"+"<update"+" " + "Name="+"\""+Name.getSelectedItem()+"\"  "+
       "type="+"\""+Type.getSelectedItem()+"\"  "+"parent="+"\""+
       Parent.getSelectedItem()+"\"  "+"valeur="+"\""+ value.getText()+"\"" +"/>";
        }
  try
    {
      File fichier = new File("C:\\Documents and Settings\\All
            Users\\Bureau\\Prolog\\verife2MaiNew\\"+jtf2.getText()+"Evolution.xml")
;
     PrintWriter out = new PrintWriter(new FileWriter(fichier)) ;
     out.write("<?xml version="+"\""+"1.0"+"\" "+"encoding="+"\""+"UTF-
            8"+"\""+"?>");
     out.println();
     out.write("<Model>");
     out.println() ;
     out.write(operationVerif) ;
     out.println() ;
     out.write("</Model>");
     out.close() ;

    }catch(Exception e){
   System.out.println("Fichier specifié est introuvable");
}

});
/**
        * cette méthode lance le moteur de vérification de cohérence
 */


public void lancerVerif(String fileName,String fileNameEvol){
       int k=1;

       long tempsInit=0;

       tempsInit=System.nanoTime();
                    if(fileNameEvol=="")
              {
              MoteurVerification.initTempsCourant();
```

```
                ElementModel.initCpComposant();
                map = Transformation.init(fileName,"ACME");
                tempsInit=System.nanoTime();
                map.transformation(); // Transformation du modèle initial vers Praxis
                temptransformationModelInitiale=System.nanoTime();

        temptransformationModelInitiale=temptransformationModelInitiale-tempsInit;

        System.out.println("Liste d'action en Praxis \n"
        +MoteurVerification.toStringList(MoteurVerification.getListInitial()));

                }
                if(fileNameEvol!="")
                {
                evol = Evolution.init(fileNameEvol,"ACME");
                tempsInit=System.nanoTime();
                evol.transformation();// Transformation du modèle d'évolution vers
Praxis
                temptransformationModelEvolution=System.nanoTime();

        temptransformationModelEvolution=temptransformationModelEvolution-tempsInit;

        System.out.println("Liste d'action d'évolutions \n"
        +MoteurVerification.toStringList(MoteurVerification.getlistEvolution()));
        long tempsEvolutionDebut= System.nanoTime();

        this.appliquerEvolution();

        tempsEvolution=System.nanoTime();
        tempsEvolution=tempsEvolution - tempsEvolutionDebut;

        long tempsCoherenceDebut=System.nanoTime();

        this.verifierCoherence();

        tempsCoherence=System.nanoTime();
        tempsCoherence=tempsCoherence - tempsCoherenceDebut;

        }

        if((!MoteurVerification.getListIncohrence().isEmpty()))
                {
                        trierListeIncoherence();
                        System.out.println("Liste d'action source d'incoherence
\n"+MoteurVerification.toStringList(MoteurVerification.getListIncohrence()));
                        long tempsSolutionDebut= System.nanoTime();
                        this.proposerPlanReparation(); tempsSolution=
                        System.nanoTime(); tempsSolution=tempsSolution-
                        tempsSolutionDebut;
                                        }
                else
                {
                        String i="N";
                        if(fileNameEvol=="")
                        {
                                System.out.println("Modèle est cohérent");
                        }
                        else
                        {
                        System.out.println("Modèle est cohérent, pour afficher la
liste d'action, tapper O \n");

                        Scanner sc = new Scanner(System.in);
                        i  = sc.next();
```

```
	System.out.println(" liste
d'action\n"+MoteurVerification.toStringList(MoteurVerification.getlistTemporaire())
);
                              System.out.println("****************************");
	map.afficherActionToXml(map.ActionToXml(MoteurVerification.getlistTemporaire(
)));
                              map.genererActionToXmlFile((fichier.getName()+"a.xml"),
map.ActionToXml(MoteurVerification.getlistTemporaire()));

                        }
                }

                 if(!MoteurVerification.getplanReparation().isEmpty())
                 {
                                                System.out.println("Plan de
réparation\n"+MoteurVerification.toStringList(MoteurVerification.getplanReparation(
)));

                 long tempsappliSolutionDebut=System.nanoTime();
                 this.appliquerSolution();
                 tempsappliSolution=System.nanoTime();
                tempsappliSolution=tempsappliSolution - tempsappliSolutionDebut;
                System.out.println("Plan de réparation appliqué");
                System.out.println("Nouvelle liste
d'action\n"+MoteurVerification.toStringList(MoteurVerification.getlistTemporaire())
);
                System.out.println("****************************");
	map.afficherActionToXml(map.ActionToXml(MoteurVerification.getlistTemporaire(
)));
                map.genererActionToXmlFile((fichier.getName()+"a.xml"),
map.ActionToXml(MoteurVerification.getlistTemporaire()));
                 }
                 else
                 {
                        if(fileNameEvol!="")
                              {
                               System.out.println("Pas de plan de réparation");
                              }
                 }

tempsGlobaleAvecTransformation=temptransformationModelInitiale/1000000.0+temptransf
ormationModelEvolution/1000000.0+tempsEvolution/1000000.0+tempsCoherence/1000000.0+
tempsSolution/1000000.0+tempsappliSolution/1000000.0;

tempsGlobale=tempsEvolution/1000000.0+tempsCoherence/1000000.0+tempsSolution/100000
0.0+tempsappliSolution/1000000.0;

                 if(fileNameEvol!="")
                       { System.out.println("tailleModele\n"+tailleModele);
                System.out.println("****************************");
                System.out.println("tailleModeleAction\n"+tailleModeleAction);
                System.out.println("****************************");
                System.out.println("nbrOperationEvolution\n"+nbrOperationEvolution);
                System.out.println("****************************");
	System.out.println("nbrOperationEvolutionAction\n"+nbrOperationEvolutionActio
n);
                System.out.println("****************************");
                System.out.println("nbrIncoherence\n"+nbrIncoherence);
                System.out.println("****************************");
                System.out.println("nbrSolution\n"+nbrSolution);
                System.out.println("****************************");
                System.out.println("tempsCoherence\n"+tempsCoherence/1000000.0);
                System.out.println("****************************");
                System.out.println("tempsEvolution\n"+tempsEvolution/1000000.0);
```

```
		System.out.println("*****************************");
		System.out.println("tempsSolution\n"+tempsSolution/1000000.0);
		System.out.println("*****************************");

	System.out.println("tempsappliSolution\n"+tempsappliSolution/1000000.0);
		System.out.println("*****************************");

	System.out.println("temptransformationModelInitiale\n"+temptransformationMode
lInitiale/1000000.0);
		System.out.println("*****************************");

	System.out.println("temptransformationModelEvolution\n"+temptransformationMod
elEvolution/1000000.0);
		System.out.println("*****************************");

	System.out.println("tempsGlobaleAvecTransformation\n"+tempsGlobaleAvecTransfo
rmation);
		System.out.println("*****************************");
		System.out.println("tempsGlobale\n"+tempsGlobale);
		System.out.println("*****************************");
			}
	}
/**
	 * cette méthode transforme le modèle ACME initiale  vers Praxis
 */
 public void transformation(){

		int j = 0, i = 0, k=0,l=0,m=0;
		this.initAction();

		Element racine = document.getRootElement();
		List<Element> element = racine.getChildren();
		Action e;
		AddProperty p;
		String id, valueProp="", nameProp="",
s[],description,direction,idInterface="" ;
		Element el;
		Attribute atr; List<Attribute> attrs;
		MoteurVerification.initListInitial();
		MoteurVerification.initlistTemporaire();
		int test= element.size();
		while (j<element.size()){
			el = element.get(j);
			if (el.getName()=="component"){
				MoteurVerification.tailleModele++;
				attrs =el.getAttributes() ;
				List<Element> element1 = el.getChildren();
				i = 0;
				while(i<attrs.size()){
					atr = attrs.get(i);
					valueProp = atr.getValue();
					nameProp = atr.getName();
					e=new Create("cp"+valueProp,
ElementModel.Composant,MoteurVerification.getNextTempsCourant());
					MoteurVerification.setListInitial(e);
					MoteurVerification.setlistTemporaire(e);

	TransformationMyAdlAction.setlistComposant(valueProp);
					e=new AddProperty("cp"+valueProp, "Name",
valueProp, MoteurVerification.getNextTempsCourant());
					MoteurVerification.setListInitial(e);
					MoteurVerification.setlistTemporaire(e);
					i++;
				}
				l=0;
```

```
                        while (l<element1.size()){
                                el = element1.get(l);
                                if (el.getName()=="port"){
                                        attrs =el.getAttributes() ;
                                        i = 0;
                                        while(i<attrs.size()){
                                                atr = attrs.get(i);
                                                idInterface = atr.getValue();
                                                nameProp = atr.getName();
                                                String
typeinterface=idInterface.substring(0,2);
                                                if(typeinterface.contentEquals("IC"))

        typeinterface="InterfaceCible";
                                                else typeinterface="InterfaceSource";
                                                e=new AddProperty("cp"+valueProp,
typeinterface, idInterface, MoteurVerification.getNextTempsCourant());
                                                MoteurVerification.setListInitial(e);

        MoteurVerification.setlistTemporaire(e);

        TransformationMyAdlAction.setlistComposant(valueProp);
                                                i++;
                                        }
                                }


                                l++;
                        }
                }
                String information="";
                if (el.getName()=="connector"){
                        MoteurVerification.tailleModele++;
                        attrs =el.getAttributes() ;
                        List<Element> element1 = el.getChildren();
                        i = 0;
                        while(i<attrs.size()){
                                atr = attrs.get(i);
                                valueProp = atr.getValue();
                                nameProp = atr.getName();
                                e=new Create("co"+valueProp,
ElementModel.Connecteur,MoteurVerification.getNextTempsCourant());
                                MoteurVerification.setListInitial(e);
                                MoteurVerification.setlistTemporaire(e);

        TransformationMyAdlAction.setlistComposant(valueProp);
                                e=new AddProperty("co"+valueProp, "Name",
valueProp, MoteurVerification.getNextTempsCourant());
                                MoteurVerification.setListInitial(e);
                                MoteurVerification.setlistTemporaire(e);
                                i++;
                        }
                        l=0;

                        while (l<element1.size()){
                                el = element1.get(l);
                                if (el.getName()=="role"){
                                        List<Element> elementprop =
el.getChildren();
                                        attrs =el.getAttributes() ;
                                        i = 0;
                                        while(i<attrs.size()){
                                                atr = attrs.get(i);
                                                idInterface = atr.getValue();
                                                nameProp = atr.getName();
                                                information=idInterface;
```

```
                                                String
typeinterface=idInterface.substring(0,2);
                                                if(typeinterface.contentEquals("IC"))

        typeinterface="InterfaceCible";
                                                else typeinterface="InterfaceSource";
                                                e=new AddProperty("co"+valueProp,
typeinterface, idInterface, MoteurVerification.getNextTempsCourant());
                                                MoteurVerification.setListInitial(e);

        MoteurVerification.setlistTemporaire(e);

        TransformationMyAdlAction.setlistComposant(valueProp);

                                                        i++;
                                                }
                                                int rech=0;
                                                while (rech<elementprop.size()){
                                                        el = elementprop.get(rech);
                                                        if (el.getName()=="userdata"){
                                                                attrs =el.getAttributes() ;
                                                                i = 0;
                                                                while(i<attrs.size()){
                                                                        atr = attrs.get(i);
                                                                        nameProp =
atr.getName();

        if(nameProp.contentEquals("key"))
                                                                                idInterface =
atr.getValue();


                                                                        i++;
                                                                }
                                                                e=new
AddProperty("co"+valueProp, "InformationRol"+information
,idInterface,MoteurVerification.getNextTempsCourant());

        MoteurVerification.setListInitial(e);

        MoteurVerification.setlistTemporaire(e);

                                                        }
                                                        rech++;
                                                }

                                        }
                                        if (el.getName()=="property"){
                                                attrs =el.getAttributes() ;
                                                i = 0;
                                                while(i<attrs.size()){
                                                        atr = attrs.get(i);

                                                        nameProp = atr.getName();
                                                        if(nameProp.contentEquals("name"))
                                                                idInterface = atr.getValue();
                                                        else
                                                                nameProp=atr.getValue();



                                                        i++;
                                                }
                                                e=new AddProperty("co"+valueProp,
idInterface, nameProp, MoteurVerification.getNextTempsCourant());
                                                MoteurVerification.setListInitial(e);
```

```
                                        MoteurVerification.setlistTemporaire(e);

                                }

                                l++;
                        }

                }
                if(el.getName()=="userdata")
                {
                        String Interface="";
                String type="";
                        List<Element> element1 = el.getChildren();
                        l=0;
                        while (l<element1.size()){
                                el = element1.get(l);
                                if (el.getName()=="data"){
                                        List<Element> element2 = el.getChildren();
                                        k=0;
                                        while (k<element2.size()){
                                                el = element2.get(k);
                                                if
(el.getName()=="portConnectionRule"){
                                attrs =el.getAttributes() ;
                                i = 0;
                                while(i<attrs.size()){
                                atr = attrs.get(i);
                                nameProp = atr.getName();

        if(nameProp.contentEquals("name"))
                                valueProp = atr.getValue();
                                        i++;
                                }
                                List<Element> element3 = el.getChildren();
                                m=0;
                                        while (m<element3.size()){
                                        el = element3.get(m);
                                        if (el.getName()=="association"){
                                                        attrs =el.getAttributes() ;
                                                        i = 0;


        while(i<attrs.size()){
                        atr = attrs.get(i);
                        nameProp = atr.getName();

        if(nameProp.contentEquals("portName"))

        Interface=atr.getValue();

        if(nameProp.contentEquals("roleName"))
        {
        type=atr.getValue();

        if(type.substring(0,2).contentEquals("IC"))

        type="LinkCi";

        else type="LinkSi";
                                }

                i++;
}
e=new AddProperty("co"+valueProp, type, Interface,
MoteurVerification.getNextTempsCourant());

        MoteurVerification.setListInitial(e);
```

```
MoteurVerification.setlistTemporaire(e);
                                                        }
                                                        m++;
                                                }

                                        }
                                                k++;
                                        }
                                }
                        l++;
                }

                }
                j++;
                        }


        }
/**
         * cette méthode transforme les opérations d'évolutions en une liste d'action
         * de PRAXIS
         */

        public void transformation(){
                int j = 0;
                int z=0;
                Action a;
                this.initAction();
                Vector<Action> v = MoteurVerification.getListInitial();
                z=0;
                while(z < v.size()){
                        a = v.get(z);
                        //MoteurVerification.setlistTemporaire(a);
                        MoteurVerification.setlistInitialRecup(a);
                        z++;
                }

                Element racine = document.getRootElement();
                List<Element> element = racine.getChildren();
                Element el;
                List<Attribute> attrs;

                while (j<element.size()){
                        el = element.get(j);
                        attrs =el.getAttributes() ;

                        if (el.getName()=="add"){
                                MoteurVerification.nbrOperationEvolution++;
                        strategieAjout(attrs, v);

                        }
                        v = MoteurVerification.getListInitial();
              if (el.getName()=="remove"){
                MoteurVerification.nbrOperationEvolution++;
                                strategieSuppression(attrs, v);


                        }
              v = MoteurVerification.getListInitial();
              if (el.getName()=="update"){
                MoteurVerification.nbrOperationEvolution++;
                        strategieModification(attrs, v);
                        }
                        j++;
                }
```

```
	}

public void appliquerEvolution(){

		Vector<Action> v = MoteurVerification.getlistTemporaire();
		Vector<Action> evol = MoteurVerification.getlistEvolution();
		ProprieteSemantique proInco = null;


		int j=0;
		int i=0;
		int index=0;
		Action a,s;
		nbrOperationEvolutionAction= evol.size();
		tailleModeleAction= v.size();
		while(j<evol.size()){ a=evol.get(j);
			if(a.getClass()==RemProperty.class)
			{
				RemProperty r= (RemProperty)a;
			i=0;
			while(i<v.size())
			{

				s=v.get(i);
				if(s.getClass()==AddProperty.class){
					AddProperty p = (AddProperty) s;

	if((p.getId().contentEquals(r.getId()))&&(p.getNameProp().contentEquals(r.get
NameProp()))&&(p.getValueProp().contentEquals(r.getValueProp()))){
							index=i;v.remove(index);
						}
						}

				i++;

			}

			}
			else
			{
			if(a.getClass()==Delete.class)
			{
				Delete r= (Delete)a;
			i=0;
			while(i<v.size())
			{

				s=v.get(i);
				if(s.getClass()==Create.class){
					Create p = (Create) s;
					if(p.getId().contentEquals(r.getId())){
						index=i;v.remove(index);
					}
					}

				i++;

			}
			}
			else
			{
				MoteurVerification.setlistTemporaire(a);

			}
```

```
                        }
                        j++;
                }

                System.out.println("Liste d'action en Praxis intermédiaire \n"

        +MoteurVerification.toStringList(MoteurVerification.getlistTemporaire()));

        }

private void verifierCoherence() {

                                MoteurVerification.initListIncohrence();
                Vector<Action> vtemp = MoteurVerification.getlistTemporaire();
                int i=0;
                Action a,s;
                                while(i<vtemp.size())
                {
                        a=vtemp.get(i);

                        if(a.getClass()==AddProperty.class)
                        {
                                AddProperty addP= (AddProperty)a;
                                if(addP.getNameProp().contentEquals("exlPar")){
                                        ExclusivitePartage pverif = new
ExclusivitePartage(addP.getNameProp(), addP.getValueProp(),
addP.getId(),addP.getTemps());
                                                                        pverif.verifierPS();

                        }

                                if(addP.getNameProp().contentEquals("Cardinalite")){
                                        Cardinalite pverif = new
Cardinalite(addP.getNameProp(), addP.getValueProp(), addP.getId(),addP.getTemps());
                                                                        pverif.verifierPS();
                                                                }

                }
                i++;
                }

                nbrIncoherence=MoteurVerification.getListIncohrence().size();
        }
/**
         * cette méthode lance la vérification de la propriété sémantique
exclusivité/partage
         */


public void verifierPS()
        {   String lienSource="";
            String lienCible="";
            String valeurProprieté="";

               if(this.getValeur().contentEquals("Ps-Ec")){
                        lienCible = MoteurVerification.map.getLienCible();
                }
                if(this.getValeur().contentEquals("Es-Pc")){
                        lienSource=MoteurVerification.map.getLienSource();
                }
                if(this.getValeur().contentEquals("E")){
                        lienCible = MoteurVerification.map.getLienCible();
                        lienSource=MoteurVerification.map.getLienSource();
                }
                if(lienCible!=""){
```

```
                        valeurPropriété=this.vérifierLienSource(lienCible);
                        if(valeurPropriété!=""){
                                vérifierIncohérence(valeurPropriété,lienCible);

                        }
                        }
                if(lienSource!=""){
                        valeurPropriété=vérifierLienSource(lienSource);
                        if(valeurPropriété!=""){
                                vérifierIncohérence(valeurPropriété,lienSource);

                        }
                        }
                verifierInfluence();
                }
/**
         * cette méthode vérifie les influences entre la propriété sémantique
exclusivité/partage et la propriété cardinalité
         */

        private void verifierInfluence()
        {
                Vector<Action> v = MoteurVerification.getlistTemporaire();
                int i=0;
                Action a;
                String cardinalite ="";
                String cardinaliteInvers="";
                AddProperty p=null;
                while(i<v.size())
                {
                        a=v.get(i);
                        if(a.getClass()==AddProperty.class)
                        {
                                p= (AddProperty)a;

        if((p.getId().contentEquals(this.getidElement()))&&(p.getNameProp().contentEq
uals("Cardinalite")))
                {
                        String separateur=",";
                        String card []= p.getValueProp().split(separateur);
                         cardinalite =card[0];
                         cardinaliteInvers=card[1];
                         i=v.size();
                }
                        }
                i++;
                }
                if(!(cardinalite=="")&&!(cardinaliteInvers==""))
                {
                        if(this.getValeur().contentEquals("Ps-Ec")){


                        if((Integer.parseInt(cardinaliteInvers)>1))
                        {

        MoteurVerification.setListIncohrence(p);
                        }
                }
                if(this.getValeur().contentEquals("Es-Pc")){
                        if((Integer.parseInt(cardinalite)>1))
                        {

        MoteurVerification.setListIncohrence(p);
                        }

                }
                if(this.getValeur().contentEquals("E")){
```

```
	if((Integer.parseInt(cardinalite)>1)||(Integer.parseInt(cardinaliteInvers)>1)
)

MoteurVerification.setListIncohrence(p);

		}
		}
	}
	/**
	 *
	 * @param valeurProprieté
	 * @param lien
	 * Vérifie la cohérence de la propriété ExclusivitéPartage
	 */
	private void vérifierIncohérence(String valeurProprieté, String lien) {

Vector<Action> v = MoteurVerification.getlistTemporaire();

		int i=0;
		Action a,b;
		Action plan;
		int Incoherence=0;

		while(i < v.size()){ a = v.get(i);
			if(a.getClass()==AddProperty.class){
				AddProperty p= (AddProperty)a;

	if(!(p.getId().contentEquals(this.getidElement()))&&(p.getNameProp().contentE
quals(lien))&&(p.getValueProp().contentEquals(valeurProprieté))&&(p.getId().substri
ng(0,2).contentEquals(this.getidElement().substring(0,2)))){

					Incoherence=1;
				MoteurVerification.setListIncohrence(a);
				MoteurVerification.setlistIncoherenceEvolutione(a);
				i++;

		}
				else
				{
if((p.getId().contentEquals(this.getidElement()))&&(p.getNameProp().contentEquals(l
ien))&&!(p.getValueProp().contentEquals(valeurProprieté))&&(p.getId().substring(0,2
).contentEquals(this.getidElement().substring(0,2)))){

					Incoherence=1;

				MoteurVerification.setListIncohrence(a);
				MoteurVerification.setlistIncoherenceEvolutione(a);
				i++;

		}
				else i++;

				}
			}
			else i++;
		}
		if(Incoherence==1)
		{
			plan= new
AddProperty(this.getidElement(),this.getNom(),this.getValeur(),this.getTemp());
			MoteurVerification.setListIncohrence(plan);

		}
```

```
	}
/**
 *
 * @param lien
 * @return
 * retourne le nom du lien pour le quel cette proporiete est définie
 */
	private String vérifierLienSource(String lien) {
		// TODO Auto-generated method stub
		String value="";
		Vector<Action> v = MoteurVerification.getlistTemporaire();
		int i=0;
		Action a;
		while(i < v.size()){
			a = v.get(i);
			if(a.getClass()==AddProperty.class){
				AddProperty p= (AddProperty)a;

	if((p.getId().contentEquals(this.getidElement()))&&(p.getNameProp()==lien)){
					value= p.getValueProp(); i=v.size();
		}
				else i++;

			}
			else i++;
		}
		return (value);
	}
/**
	 * cette méthode retourne la solution pour l'incohérence trouvée suite à la
*vérification de la propriété exclusivité/partage
	 */

public void Solution() {
	Vector<Action> v = MoteurVerification.getlistTemporaire();
	int i=0;
	int nombreLienSource=0,nombreLienCible=0;
	String cardinalite ="";
	String cardinaliteInvers="";

	Action a; Action
	Plan;
	while(i<v.size())
	{
		a=v.get(i);

		if(a.getClass()==AddProperty.class)
		{
			AddProperty p= (AddProperty)a;

	if((p.getId().contentEquals(this.getidElement()))&&(p.getNameProp().contentEq
uals("LinkSi")))
				{
			nombreLienSource++;

				}

	if((p.getId().contentEquals(this.getidElement()))&&(p.getNameProp().contentEq
uals("LinkCi")))
			{
				nombreLienCible++;

			}

	if((p.getId().contentEquals(this.getidElement()))&&(p.getNameProp().contentEq
uals("Cardinalite")))
			{
```

```
                                String separateur=",";
                                String card []= p.getValueProp().split(separateur);
                                 cardinalite =card[0];
                                 cardinaliteInvers=card[1];

                        }
                }
                i++;
        }
        if((nombreLienCible==1)&&(nombreLienSource==1))
        {
        if((Integer.parseInt(cardinalite)==1)&&(Integer.parseInt(cardinaliteInvers)==
1))
                {
                        //E

                        Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
                        MoteurVerification.setplanReparation(Plan);
                        Plan= new
AddProperty(this.getidElement(),this.getNom(),"E",MoteurVerification.getNextTempsCo
urant());
                        MoteurVerification.setplanReparation(Plan);
                }

        if((Integer.parseInt(cardinalite)>1)&&(Integer.parseInt(cardinaliteInvers)==1
))
                {
                        //Ps Ec
                        Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
                        MoteurVerification.setplanReparation(Plan);
                        Plan= new AddProperty(this.getidElement(),this.getNom(),"Ps-
Ec",MoteurVerification.getNextTempsCourant());
                        MoteurVerification.setplanReparation(Plan);
                }

        if((Integer.parseInt(cardinalite)==1)&&(Integer.parseInt(cardinaliteInvers)>1
))
                {
                // Es Pc
                        Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
                        MoteurVerification.setplanReparation(Plan);
                        Plan= new AddProperty(this.getidElement(),this.getNom(),"Es-
Pc",MoteurVerification.getNextTempsCourant());
                        MoteurVerification.setplanReparation(Plan);
                }

        if((Integer.parseInt(cardinalite)>1)&&(Integer.parseInt(cardinaliteInvers)>1)
)
                {
                //P
                        Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
                        MoteurVerification.setplanReparation(Plan);
                        Plan= new
AddProperty(this.getidElement(),this.getNom(),"P",MoteurVerification.getNextTempsCo
urant());
                        MoteurVerification.setplanReparation(Plan);
                }
        }
        if((nombreLienCible>1)&&(nombreLienSource==1))
```

```
	{
		//Ps Ec
		Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
		MoteurVerification.setplanReparation(Plan);
		Plan= new AddProperty(this.getidElement(),this.getNom(),"Es-
Pc",MoteurVerification.getNextTempsCourant());
		MoteurVerification.setplanReparation(Plan);
		if(!(Integer.parseInt(cardinalite)>1))
		{
			Plan=new
RemProperty(this.getidElement(),"Cardinalite",this.getValeur(),MoteurVerification.g
etNextTempsCourant());
			MoteurVerification.setplanReparation(Plan);
			Plan= new
AddProperty(this.getidElement(),"Cardinalite","1,"+nombreLienCible,MoteurVerificati
on.getNextTempsCourant());
			MoteurVerification.setplanReparation(Plan);
		}
	}
	if((nombreLienCible==1)&&(nombreLienSource>1))
	{
		//Ec Ps
		Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
		MoteurVerification.setplanReparation(Plan);
		Plan= new AddProperty(this.getidElement(),this.getNom(),"Ps-
Ec",MoteurVerification.getNextTempsCourant());
		MoteurVerification.setplanReparation(Plan);
		if(!(Integer.parseInt(cardinaliteInvers)>1))
		{
			Plan=new
RemProperty(this.getidElement(),"Cardinalite",this.getValeur(),MoteurVerification.g
etNextTempsCourant());
			MoteurVerification.setplanReparation(Plan);
			Plan= new
AddProperty(this.getidElement(),"Cardinalite",nombreLienSource+",1",MoteurVerificat
ion.getNextTempsCourant());
			MoteurVerification.setplanReparation(Plan);
		}
	}
	if((nombreLienCible>1)&&(nombreLienSource>1))
	{
		//P
		Plan=new
RemProperty(this.getidElement(),this.getNom(),this.getValeur(),MoteurVerification.g
etNextTempsCourant());
		MoteurVerification.setplanReparation(Plan);
		Plan= new
AddProperty(this.getidElement(),this.getNom(),"P",MoteurVerification.getNextTempsCo
urant());
		MoteurVerification.setplanReparation(Plan);
	if(!(Integer.parseInt(cardinalite)>1)&&!(Integer.parseInt(cardinaliteInvers)>
1))
		{
			Plan=new
RemProperty(this.getidElement(),"Cardinalite",this.getValeur(),MoteurVerification.g
etNextTempsCourant());
			MoteurVerification.setplanReparation(Plan);
			Plan= new
AddProperty(this.getidElement(),"Cardinalite",nombreLienSource+","+nombreLienCible,
MoteurVerification.getNextTempsCourant());
			MoteurVerification.setplanReparation(Plan);
		}
	}
```

```
}

private void proposerPlanReparation() {
		Vector<Action> vIncoh = MoteurVerification.getListIncohrence();
		Vector<Action> vEvol = MoteurVerification.getlistEvolution();
		//trier liste incohérences
		trierListeIncoherence();
		 MoteurVerification.initplanReparation();
		// parcourire liste incoérence
		int i=0,j=0,trouve=0;
		Action a,v;
		while(i<vIncoh.size())
		{
			a=vIncoh.get(i);
			if(a.getClass()==AddProperty.class)
			{
				AddProperty addP= (AddProperty)a;
				if(addP.getNameProp().contentEquals("exlPar")){
					ExclusivitePartage psolusion = new
ExclusivitePartage(addP.getNameProp(), addP.getValueProp(),
addP.getId(),addP.getTemps());
					//verifier si la propriete est modifier dans la
liste d'evolution
					while(j<vEvol.size())
					{
						trouve=0; v=vEvol.get(i);
						if(v.getClass()==AddProperty.class)
						{
							AddProperty addPEvol= (AddProperty)v;

if((addPEvol.getNameProp().contentEquals("exlPar"))&&(addPEvol.getId().contentEqual
s(addP.getId())))
							{
								psolusion.Solution();
									j=vEvol.size();
									trouve=1;
							}
							else j++;
						}
						else j++;

					}
					if(trouve==0)
					{
						psolusion.influence();
					}

				i++;
				}

				else
			if(addP.getNameProp().contentEquals("Cardinalite")){
				Cardinalite psolusion = new Cardinalite(addP.getNameProp(),
addP.getValueProp(), addP.getId(),addP.getTemps());
				//verifier si la propriete est modifier dans la liste
d'evolution
				j=0;
				while(j<vEvol.size())
					{
				trouve=0;
						v=vEvol.get(j);
						if(v.getClass()==AddProperty.class)
						{
							AddProperty addPEvol= (AddProperty)v;
```

```
if((addPEvol.getNameProp().contentEquals("Cardinalite"))&&(addPEvol.getId().content
Equals(addP.getId())))
                                                {
                                                       psolusion.Solution();
                                                             j=vEvol.size();
                                                             trouve=1;
                                                }
                                                else j++;
                                          }
                                          else j++;
                                    }
                              if(trouve==0)
                                    {
                                          psolusion.influence();
                                    }
                                    i++;
                              }else

                if(addP.getNameProp().contentEquals("LinkCi")){

                  addP.getSolutionC();
                  //supprimer le lien
                  i++;
                              }else
                if(addP.getNameProp().contentEquals("LinkSi")){
                  addP.getSolutionS();
                  //supprimer le lien
                  i++;
                              }
                else i++;
                        }
                        else i++;
                  }
                  //Appliquer plan sur une copie du modèle intermediaire
                  appliquerSolution();
                  verifierInvariant();
                   nbrSolution=MoteurVerification.getplanReparation().size();



        }

* strategie de modification pour les propriétés sémantiques
         */
        public void strategieModification(List<Attribute> attrs, Vector<Action> v)
        {String id=null, valueProp = null,
nameProp,name="",value="",PropSem=null,type="",parent="",source=null,cible=null,
s[],interfaceCible=null,interfaceSource=null  ;
        int i=0,z=0;
        Attribute atr;
        Action e,a,recup;
        String  propSemOld=null;
        while(i<attrs.size()){
                atr = attrs.get(i);
                value = atr.getValue();
                name = atr.getName();
                if(name=="Name"){
                        valueProp=value;
                }
                if(name=="type"){
                        type=value;

                }
                if(name=="parent"){
                        parent=value;
                }
```

```
                if(name=="source"){
                        source=value;
                }
                if(name=="cible"){
                        cible=value;
                }
                if(name=="valeur"){
                        PropSem=value;
                }
                i++;
        }
                recup=recupAction(v,"Name", parent);
                if(recup!=null)
                {
                if(recup.getClass()==AddProperty.class){
                        AddProperty recupid= (AddProperty)recup;
                        id=recupid.getId();
                }}
                recup=recupAction(v,valueProp, id);
                if(recup!=null)
                {
                if(recup.getClass()==AddProperty.class){
                        AddProperty recupvalp= (AddProperty)recup;
                        propSemOld=recupvalp.getValueProp();
                        e = new RemProperty(id, valueProp,propSemOld
,MoteurVerification.getNextTempsCourant());
                        MoteurVerification.setlistEvolution(e);
                        MoteurVerification.setListInitial(e);
                    e = new AddProperty(id, valueProp,PropSem
,MoteurVerification.getNextTempsCourant());
                    MoteurVerification.setlistEvolution(e);
                    MoteurVerification.setListInitial(e);
                }}


        }
```