République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Saad Dahlab Blida

N° D'ordre :.....

Faculté des sciences

Département d'informatique

Mémoire Présenté par :

MAIZI Naima OULHADJ Aziza

En vue d'obtenir le diplôme de master

Domaine : Mathématique et informatique

Filière : Informatique
Spécialité : Informatique
Option : Ingénierie de logiciel

Sujet :

Vérification d'architecture logicielle par transformation de modèles.

Soutenu le :

Mme. MANCER Yasmine	Président
Mr. BAOUYA	Examinateur
Mme. GUESSOUM Dalila	Promotrice

Promotion
2015 / 2016

Dédicace

Je dédie ce travail a toute ma famille surtout ma mère mon père qui m'encouragent toujours, mes frères et sœurs, mes neveux et nièces et toutes mes amies.

A ceux qui m'ont accompagné durant ce travail.

Naima

Dédicaces

Je dédie ce travail à mes chers parents et ma petite Nina et à tous ceux qui m'ont soutenu durant toute la durée de ce travail.

Aziza

Remerciements

Nous remercions Allah pour nous avoir donné santé, courage et patience afin de nous aider à réaliser ce travail.

Nous remercions chaleureusement notre promotrice Mme Dalila GUESSOUM pour le temps qu'elle nous a consacré et pour ses conseils constructifs.

Nos plus sincères remerciements pour ceux qui nous ont aidé pour réaliser ce projet de fin d'étude.

Résumé :

Ce modeste travail s'inscrit dans le cadre des architectures logicielles à base de composants, le but de ce travail est d'effectué une vérification d'architecture logicielle par transformation de modèle, pour cela nous avons choisi la méthode formelle B comme une technique de vérification et le ATL comme un langage de transformation dans notre projet.

Mots-clés : Architecture Logicielle, Composant, Transformation de modèle, techniques de vérification, modèle architectural, ATL.

Abstract :

This work aims to make a Software architecture verification by making a model transformation, thus we've chosen method B as a verification method for our architectural model and ATL as the transformation language in our project.

Keywords : Software architecture , model transformation , verification methods , architectural model, ATL.

Liste des Tableaux

Liste des Figures

Table des matières

Introduction générale :

Depuis les débuts du génie logiciel, la taille et la complexité des logiciels développés augmentent de plus en plus rapidement **[FLE,06]**.Les architectures logicielles jouent donc un rôle prépondérant **[TIB,13]** de réduire cette complexité et sont devenues un artefact central dans le cycle de vie de ces systèmes informatiques, car elles permettent aux différents protagonistes d'avoir une idée synthétique de leur organisation **[TIB,13]**. Elles s'imposent de plus en plus comme une étape indispensable du développement des systèmes logiciels en permettant au concepteur de raisonner sur les propriétés fonctionnelles et non fonctionnelles du système à un haut niveau d'abstraction **[SAD, 07]**.

Un des autres problèmes ouverts en génie logiciel est le développement correct de systèmes informatiques non triviaux. Il s'agit de pouvoir construire de systèmes complexes ou de très grands systèmes, à l'aide de méthodes établies et d'en fournir la garantie des fonctionnalités et des propriétés voulues et spécifiées formellement à partir du cahier de charges. Les seules techniques qui permettent d'établir en toute rigueur la correction d'un programme ou d'un système sont celles basées sur des approches formelles (ayant des fondements mathématiques qui servent de support au raisonnement) **[ATT,07]**.

Les méthodes, dites formelles, ont été élaborées afin d'assurer un niveau aussi élevé que possible en matière de précision et de cohérence. Leur avantage majeur réside, en somme, dans le fait qu'elles sont basées sur les mathématiques, ce qui permet, d'une part, de neutraliser tous les risques d'ambiguïté et d'incertitude, et d'autre part, de parvenir à un produit fini qui répond aux spécifications requises **[IDA,06]**.

Dans la continuité, les grandes tendances actuelles du génie logiciel sont l'utilisation de langages dédiés, les approches orientées-aspects et les approches à base de composants. Un des points communs à ces approches est l'utilisation de modèles pour permettre une montée en abstraction par rapport aux langages de programmation. Ces techniques sont regroupées sous l'intitulé Ingénierie Dirigée par les Modèles (IDM) ou Model-Driven Development (MDD) **[FLE,06]**.

L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique **[COM,08]**. Un des aspects essentiels de l'IDM consiste

à automatiser la transformation de modèles grâce à l'utilisation de langages de transformation **[PEM,07]** comme l'ATL (ATLAS Transformation Language).

La thématique abordée dans notre projet est la vérification d'architecture logicielle par transformation de modèles. Plus précisément l'objectif de ce projet est de réaliser un outil permettant la transformation de modèle d'un modèle de spécification d'architecture logicielle en un modèle de vérification.

Pour assurer une meilleure présentation du travail effectué et garantir la clarté du mémoire, outre cette introduction générale, ce mémoire se compose de cinq chapitres, une conclusion générale.

Le premier chapitre présente une étude sur les architectures logicielles. Il introduit les éléments architecturaux. Une section est réservée à l'étude de quelques styles architecturaux. Une autre section est consacrée à l'étude sur les langages de description des architectures ADL. Une dernière section est consacrée aux rôles et avantages de l'architecture logicielle.

Le deuxième chapitre présente une étude comparative sur les techniques de vérifications. Il introduit les outils et les techniques formelles pour la vérification d'architecture logicielle Ces techniques formelles sont aussi étudiées pour voir leur possibilité de décrire l'architecture logicielle. Une dernière section est consacrée é la comparaison entre deux méthodes de vérification.

Le troisième chapitre propose une approche IDM qui fait évoluer l'usage des modèles, qui peuvent être interprétés ou transformés. Une section est réservée à la Définition des concepts fondamentaux (modèle, métamodèle, métamétamodèle, la transformation de modèles).Une autre section est représenté les quatre niveaux des modèles qui sont : le niveau M0, le niveau M1,le niveau M2 et le niveau M3. Une autre section est consacrée aux relations entre les modèles Une dernière section est consacrée à la Spécification des règles de transformation offrant trois approches : approche par programmation, approche par template et approche par modélisation.

Le quatrième chapitre présente le modèle de vérification choisi ainsi le style architectural et leurs métas modèle. Une section est consacrée aux règles de transformation et les règles d'évolution.

Le cinquième chapitre présente le langage de transformation choisi ATL ses avantages. Une section est consacrée aux interfaces de notre application.

Chapitre 1 :

Etat de l'art sur les architectures logicielles

1.1 Introduction :

Les architectures logicielles sont considérées comme une sous-discipline du génie Logiciel. Une architecture est considérée comme l'organisation nécessaire d'un système caractérisé par ses composants, leurs relations avec l'environnement, et les principes qui guident leur conception et évolution. Les architectures logicielles forment la colonne vertébrale pour construire des logiciels complexes et de grande taille.

La description des architectures permet d'avoir l'abstraction nécessaire pour modéliser les systèmes logiciels complexes durant leur développement, déploiement et évolution.

1.2 Architecture logicielle :

L'architecture est une vue abstraite d'un système en terme d'éléments architecturaux. Ces éléments sont **[CHA,09]**:

- Les composants qui décrivent les fonctionnalités métier de l'application.
- Les connecteurs qui décrivent les communications et connexions entre les composants.
- La configuration qui décrit la topologie des connexions entre composants et connecteurs.

Une architecture logicielle est définie comme un niveau de conception qui contient la description des composants à partir desquels un système est construit, les spécifications comportementales de ces composants, les modèles et les mécanismes de leurs interactions (connecteurs) et enfin un modèle définissant la topologie (configuration) d'un système **[ALT , 11]**.

1.3 Eléments architecturaux :

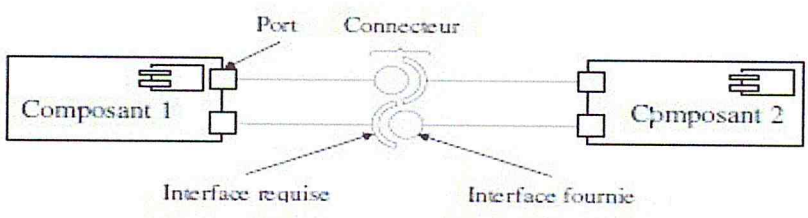

Figure 1.1 : Les éléments d'une architecture logicielle **[HAD,08]**.

1.3.1 Le Composant :

Un composant logiciel est une unité de composition possédant des interfaces spécifiées par contrat et des dépendances contextuelles explicites **[CHA,09]**.

Le composant représente le principal élément de calcul et de stockage des données dans un système **[GOA , 09]**.

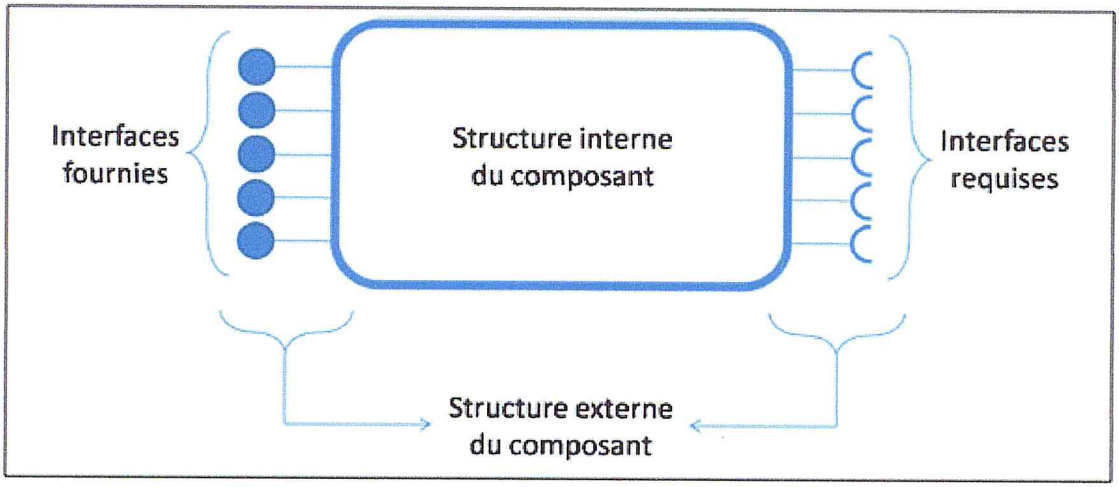

Figure 1.2 : Structure d'un composant **[CHA,09]**.

1.3.1.1 Structure externe et interne d'un composant :

- La structure externe :
  - Les interfaces du composant : Elle sont la spécification des services fournis et requis par le composant **[CHA,09]**.
  - Les propriétés du composant : Elles servent à documenter l'architecture en décrivant les aspects relevant de la conception ou de l'analyse du composant **[CHA,09]**.
- La structure interne :

  Il existe deux types de structures internes des composants. D'abord, elle peut être constituée par une description ou une implémentation, dans un langage de programmation, des fonctionnalités du composant **[CHA,09]**.

  Les composants possédant une telle structure interne sont des composants atomiques. Ils sont les blocs de base de l'architecture **[CHA,09]**.

Le second type de structures internes est composé d'autres composants. Ces composants sont des composites constitués de composants internes. Ces composants internes peuvent eux aussi être atomiques ou composites **[CHA,09]**.

1.3.2 Le connecteur :

Les connecteurs gèrent les interactions entre les composants, c'est-à-dire, ils établissent les règles qui gouvernent les interactions entre composants et spécifient tous les mécanismes auxiliaires nécessaires **[CHA,09]**.

Les connecteurs sont des entités architecturales de communication qui modélisent de manière explicite les interactions (transfert de contrôle et de données) entre les composants. Ils contiennent des informations concernant les règles d'interaction entre les composants. Ainsi, l'objectif des connecteurs est d'atteindre une meilleure réutilisabilité lors de l'assemblage des composants **[ALT , 11]**.

En effet, la raison de l'existence des connecteurs est de faciliter le développement d'applications à base de composants logiciels. Les composants s'occupent du calcul et stockage tandis que les connecteurs s'occupent de gérer les interactions (communication/coordination) entre les composants **[ALT , 11]**.

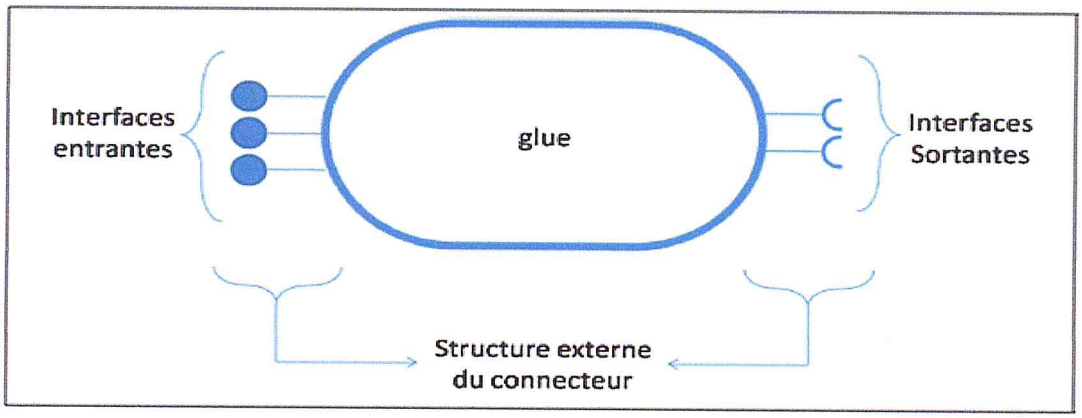

Figure 1.3 : Structure d'un connecteur **[CHA,09]**.

1.3.2.1 Structure externe d'un connecteur :

- **Les interfaces du connecteur :**

Elles servent à déclarer les participants à l'interaction décrite par le connecteur. Comme celles des composants, elles constituent les points de connexion entre les connecteurs et les composants. Néanmoins, à la différence des composants, les interfaces

ne décrivent pas de services fonctionnels mais des mécanismes de connexion. Elles décrivent également le rôle de chacun des composants impliqués **[CHA,09]**.

- **Les propriétés du connecteur :**

Les propriétés des connecteurs sont de deux types **[CHA,09]**:

- **propriétés non fonctionnelles** : elles spécifient les besoins du connecteur pour une implémentation correcte. Par exemple, elles peuvent concerner la performance ou la sécurité. Comme pour les composants, ces propriétés marquent une séparation claire entre les aspects fonctionnels et non fonctionnels.
- **contraintes** : ces propriétés définissent les conditions d'utilisation du connecteur. Comme pour les contraintes portant sur les composants, ces contraintes doivent être vérifiées pour que le système soit considéré comme cohérent.

1.3.2.2 Structure interne d'un connecteur :

De la même manière que pour les composants, on distingue deux types de structures internes pour les connecteurs : structure atomique ou composite **[CHA,09]**:

- **Connecteur atomique :** La structure interne des connecteurs atomiques est appelée glu. Elle forme la passerelle entre les interfaces du connecteur. Pour cela, elle décrit le protocole de communication entre les interfaces, points d'accès des composants vers le connecteur.
- **Connecteur composite :** Les connecteurs composites ont une structure interne plus complexe que celle des connecteurs atomiques. A l'image des composants composites, les connecteurs composites possèdent une structure interne composée de composants, de connecteurs et d'une configuration.

1.4 Configuration :

Une configuration est un graphe de composants et de connecteurs. Cette information est nécessaire pour déterminer si les composants sont bien reliés, que leurs interfaces s'accordent, que les connecteurs correspondants permettent une communication correcte et que la combinaison de leurs sémantiques aboutit au comportement désiré. Elle définit la façon dont ils sont reliés entre eux **[ALT , 11]**.

La conception d'une architecture logicielle, a une importance capitale pour la réussite d'un projet informatique. Elle est souvent liée au savoir-faire de l'architecte. Une architecture logicielle doit tenir compte des contraintes suivantes **[HAD, 08]**:

- **La réutilisabilité** : est la capacité à rendre générique des composants et à concevoir et construire des boites noires susceptibles de fonctionner avec des langages et des environnements variés.
- **La maintenabilité** : est la capacité de modifier et d'adapter une application afin de la maintenir sur une période de vie assez longue. Une architecture bien spécifiée doit être maintenue tout au long de son cycle de vie. La prévision de l'intégration des extensions à l'architecture et la correction des erreurs sont nécessaires d`es la phase de conception.
- **La performance** : c'est l'optimisation du temps mis par une application pour répondre à une requête donnée. La performance d'une application dépend de l'architecture logicielle choisie, de son environnement d'exécution, de son implémentation et de la puissance des infrastructures utilisées (ex. débit du réseau utilisé).

1.5 Style architectural :

Un style architectural caractérise une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques **[ALT, 11]**.

L'utilisation des styles architecturaux a un certain nombre d'avantages significatifs. D'abord, elle favorise la réutilisation de la conception. En effet, des applications modélisées selon un style et des propriétés architecturales bien définies peuvent être réappliquées à des nouvelles applications. Elle facilite, pour les autres, la compréhension de l'organisation de l'architecture de l'application si les structures conventionnelles sont bien appliquées et utilisées. Par exemple, concevoir une application selon le style Client-Serveur peut donner une idée claire sur le type de composants utilisés et les interactions entre les différents composants. Finalement, l'utilisation des styles architecturaux peut mener à la réutilisation significative du code : souvent les aspects invariants d'un style architectural se prétendent à des implémentations partagées. **[GUE, 12]**.

Parmi les principaux styles architecturaux, nous citons le style "client-serveur", le style"publier-souscrire" et le style "pipe and filtre".

1.5.1 Style "client-serveur" :

Comme le montre la figure 1.4, il se base sur deux types de composants : un composant de type serveur, offrant un ensemble de services, écoute des demandes sur ses services. Un composant de type client, désirant qu'un service soit assuré, envoie une demande (requête) au serveur par l'intermédiaire d'un connecteur. Le serveur rejette ou exécute la demande et envoie une réponse de nouveau au client. La contrainte qui s'impose dans ce type est qu'un composant ne peut être qu'un fournisseur de services ou un demandeur de services. Le style client-serveur consiste alors à structurer un système en terme d'entités serveurs et d'entités clientes qui communiquent par l'intermédiaire d'un protocole de communication à travers un réseau informatique **[HAD, 08]**.

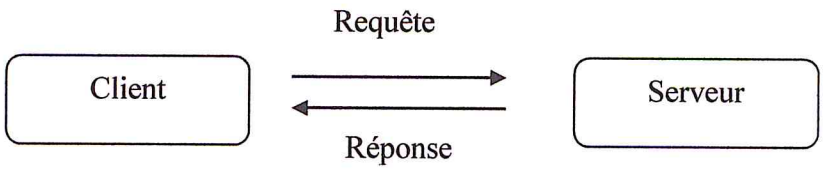

Figure 1.4 : Le style ''client-serveur'' **[HAD, 08]**.

1.5.2 Style "publier-souscrire" :

Ce style, comme le montre la figure 1.5, se base sur trois composants. Un composant producteur qui va produire des informations. Un composant consommateur qui va les consommer et un composant service d'événement qui va assurer l'échange d'informations entre les producteurs et les consommateurs. Ces derniers ne communiquent donc pas directement et ne gardent même pas les références des uns et des autres. De plus, les producteurs et les consommateurs n'ont pas besoin de participer activement à l'interaction selon un mode synchrone. Le producteur peut publier des événements pendant que le consommateur est déconnecté, et réciproquement, le consommateur peut être notifié à propos d'un évènement pendant que le producteur, source de cet événement, est déconnecté **[HAD, 08]**.

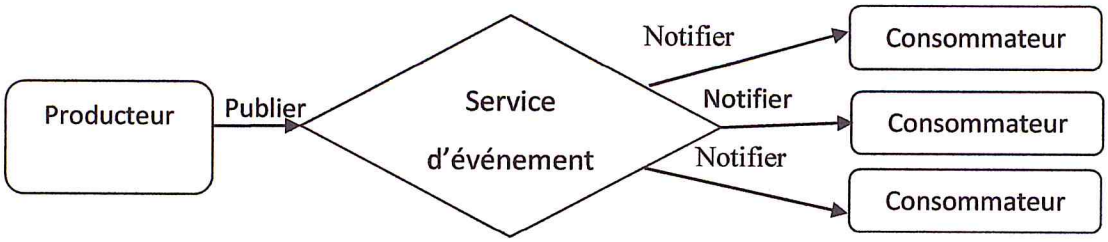

Figure 1.5 : Le style "publier-souscrire" **[HAD, 08].**

1.5.3 Style "pipes and filters"

Dans ce style, comme le montre la figure 1.6, Les composants sont appelés *filters* **[GAS,94]**et les connecteurs sont appelés *pipes***[GAS,94]**.

Chaque composant a un ensemble de données en entrée et un ensemble de données en sortie. Un composant lit un ensemble de données en entrée et produit un ensemble de données en sortie **[GAS,94]**.

Les spécifications des filtres peuvent contraindre les entrées et les sorties. Un connecteur, quant à lui, est appelé pipe puisqu'il représente une sorte de conduite qui permet de véhiculer les sorties d'un filtre vers les entrées d'un autre **[HAD, 08].**

Le style "pipes and filters" exige que les filtres soient des entités indépendantes et qu'ils ne connaissent pas l'identité des autres filtres. De plus, la validité d'un système conforme à ce style ne doit pas dépendre de l'ordre dans lequel les filtres exécutent leur traitement **[HAD, 08].**

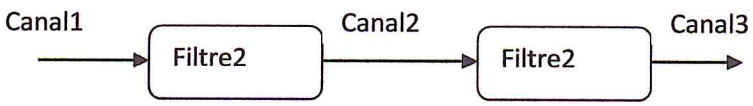

Figure 1.6 : Le style "pipes and filters" **[HAD, 08].**

Avantages des styles architecturaux :

L'utilisation des styles architecturaux a un certain nombre d'avantages significatifs. D'abord, elle favorise la réutilisation de la conception. En effet, des applications modélisées selon un style et des propriétés architecturales bien définies peuvent être réappliquées à des nouvelles applications. Elle facilite, pour les autres, la compréhension de l'organisation de l'architecture de l'application si les structures conventionnelles sont

bien appliquées et utilisées. Par exemple, concevoir une application selon le style Client-Serveur peut donner une idée claire sur le type de composants utilisés et les interactions entre les différents composants. Finalement, l'utilisation des styles architecturaux peut mener à la réutilisation significative du code : souvent les aspects invariants d'un style architectural se prétendent à des implémentations partagées **[HAD, 08]**.

1.6 Les langages de description d'architectures (ADL : Architecture Description Language) :

Un ADL se concentre plutôt sur la structure de haut niveau de l'ensemble de l'application, que sur les détails d'implémentation. Parmi les ADL les plus connus, nous citons : Wright, ACME, Rapide, Unicon, C2, Darwin et AESOP . les propriétés principales qu'un ADL doit avoir sont divisées en trois sous-familles comme suit **[GRA , 07]**:

1.6.1 Les exigences minimales fondamentales

Elles couvrent la spécification des composants, des connecteurs, des configurations et éventuellement des styles architecturaux **[GRA , 07]**.

- **Un composant** est une unité de calcul ou de données qui peut être atomique ou composite et qui possède une interface décrivant les points d'interaction du composant avec l'environnement. Un ADL doit permettre la spécification d'un type de composant et de son interface **[GRA , 07]**.
- **Un connecteur** modélise les interactions entre les composants et aussi les règles qui régissent ces interactions. Un connecteur a les mêmes caractéristiques qu'un composant : une interface et un type qui représente une description abstraite de l'interaction qu'il incarne. Il est important qu'un ADL permette de spécifier un connecteur comme une entité de première classe. Ceci permet de le réutiliser **[GRA , 07]**.
- **Une configuration** décrit la structure complète d'un système sous forme d'un graphe connexe regroupant des composants et des connecteurs. Un ADL doit fournir des aptitudes pour modéliser la configuration **[GRA , 07]**.
- **Un style architectural** décrit une classe générique d'architecture telles que : client-serveur, filtre-pipe et architecture par couches. Il est important qu'un ADL permette de spécifier explicitement un style architectural et un mécanisme

d'instanciation adéquat afin d'engendrer des configurations conformes au style architectural[GRA , 07].

1.6.2 Les exigences souhaitables

Ces exigences favorisent des implémentations correctes des architectures. Elles couvrent la modélisation de la sémantique, la spécification des contraintes, la composition hiérarchique et les propriétés non fonctionnelles [GRA , 07].

- **La modélisation de la sémantique** concerne à la fois les composants et les connecteurs. Il s'agit de décrire les comportements dynamiques des composants et des connecteurs. Par exemple, dans Wright, ces comportements sont décrits en CSP (Communicating Sequential Processes) de Hoare [GRA , 07].
- **La spécification des contraintes** concerne les composants, les connecteurs et les configurations. Par exemple, dans Wright, les contraintes relatives aux composants et connecteurs sont spécifiées par des types d'interfaces. Tandis que celles relatives aux styles architecturaux sont exprimées par des prédicats [GRA , 07].
- **La composition hiérarchique** concerne les composants et les connecteurs. Un composant ou un connecteur peut être lui-même un ensemble de composants et connecteurs. Une architecture entière peut être représentée par un composant. Par exemple, dans Wright, les composants et les connecteurs peuvent être composites [GRA , 07].
- **Les propriétés non fonctionnelles** (sécurité, performance, etc.) couvrent des exigences qui ne sont pas inhérentes à la sémantique des concepts architecturaux (composant, connecteur, configuration). Cependant, leur description est nécessaire pour une implémentation correcte de ces concepts. Par exemple, l'ADL permet d'exprimer certaines propriétés comme la planification [GRA , 07].

1.6.3 Les exigences désirables mais non fondamentales

Ces exigences supportent l'activité de l'architecte. Elles couvrent la réutilisation, l'évolution et une boîte à outils [GRA , 07].

- **La réutilisation** : il est préférable qu'un ADL offre des possibilités permettant la réutilisation des concepts architecturaux. Par exemple, dans Wright, les types

des composants et des connecteurs peuvent être réutilisés. Egalement, les types peuvent être paramétrés. De même, les styles peuvent être étendus **[GRA , 07]**.

- **Evolution** : Un ADL doit supporter des mécanismes permettant de faire évoluer -par le changement des propriétés- des composants, des connecteurs et des configurations. Par exemple, Darwin supporte l'instanciation dynamique des composants **[GRA , 07]**.
- **Outils de support** : La disponibilité d'un outil renforce l'utilité d'un ADL. Ces outils peuvent être de divers types : analyseur de syntaxe et type, générateur de code, analyseur statique des propriétés et traducteur vers des outils de vérification formelle **[GRA , 07]**.

1.7 Importance des architectures logicielles :

La description de l'architecture logicielle s'impose de plus en plus comme une étape indispensable du développement des systèmes logiciels en permettant au concepteur de raisonner sur les propriétés fonctionnelles et non fonctionnelles du système à un haut niveau d'abstraction. Il est bien admis aujourd'hui qu'une bonne architecture peut amener à un produit qui répond aux besoins des utilisateurs et qui peut être modifié facilement et qu'une mauvaise architecture peut avoir des conséquences désastreuses sur le système **[SAD , 07]**.

D'autres avantages ont été reconnus pour les architectures logicielles et qui ont été repris par **[CHA,09]** :

- **La compréhension du système :** l'architecture fournie une représentation d'un système à un haut niveau d'abstraction. Cette vue synthétique du système met en valeur la plupart des décisions de conception ainsi que les conséquences de ces mauvaises décisions.
- **La réutilisation :** les descriptions architecturales favorisent la réutilisation à plusieurs niveaux. (réutilisation des bibliothèques de composants). La conception architecturale supporte la réutilisation de grands composants *(large components)*, ainsi que les Framework ou les composants peuvent être intégrés.
- **L'évolution :** l'architecture fournie un squelette du système. Ce squelette permet d'identifier les parties fortement utilisées ainsi que les parties

potentiellement fragiles. L'architecture permet ainsi de mettre en valeur les parties nécessitant une attention particulière lors de l'évolution du système. Mais l'architecture permet également de révéler une image précise des dépendances entre les composants. Cette image est nécessaire pour connaitre les impacts de la modification d'un composant sur les autres composants du système et donc les conséquences des différentes évolutions.

- **L'analyse :** la vue abstraite fournie par l'architecture permet de mesurer différents attributs tels que la consistance du système, son respect du style architectural ou encore d'autres attributs de qualité. Elle permet également de vérifier que les changements prévus dans le système sont conformes au style et aux objectifs de qualité fixés à la conception.
- **La gestion de projets :** l'architecture permet une gestion plus précise des coûts et des risques de modifications, en particulier en soulignant les dépendances entre les composants. Elle permet également une évaluation des qualités du système dans son ensemble, mais aussi des qualités de chaque composant.

1.8 Conclusion :

L'architecture logicielle c'est l'organisation des briques d'un logiciel en une structure cohérente qui assure au logiciel d'être en mesure de remplir les objectifs et de respecter les propriétés attendus et ce chapitre porte sur la notion de l'architecture logicielle, son rôle et ses avantages et les éléments architecturaux bien détaillés.

Chapitre 2 :

Les techniques de vérification

2.1 Introduction :

La vérification est une approche s'appuyant sur un raisonnement mathématique qui permet de prouver que la description formelle d'un système satisfait certaines propriétés souhaitées. Ces propriétés s'étendent des propriétés indépendantes du système sous vérification comme la consistance et la complétude au niveau de ses spécifications aux propriétés relatives à ses comportements et ses fonctionnalités désirés.

2.2 La méthode formelle :

Les méthodes formelles peuvent être définies comme les langages, techniques et outils pour la spécification et la vérification de systèmes logiciels et matériels. Ceci englobe essentiellement deux types de méthodes formelles : des langages de spécification formelle et les techniques de vérification formelle **[FEL ,12]**.

2.3 Outils pour la vérification d'architectures logicielles :

L'activité de vérification est l'étape qui permet de s'assurer que le système est développé correctement c'est-à-dire possède certaines propriétés. Cette activité est supportée par des notations formelles comme VDM, Z, B et CSP, CCS et Réseaux de Petri **[GRA , 07]**.

Les techniques formelles peuvent être classées principalement en trois catégories **[HAD, 08]** comme le montre la figure 2.1 .

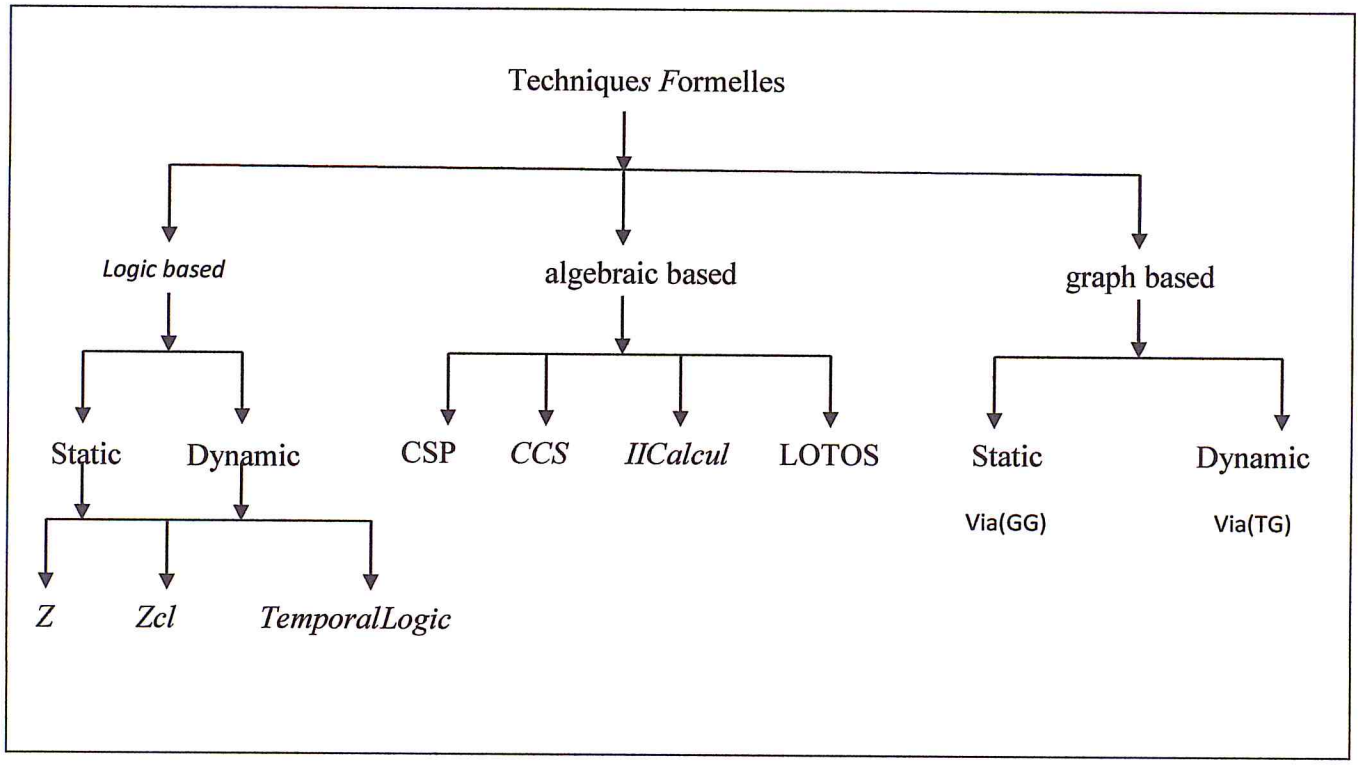

Figure 2.1 : Les techniques formelles **[HAD, 08]**.

Les méthodes de spécifications formelles sont utilisées en génie logiciel pour raisonner sur des modèles mathématiques. L'intérêt est de pouvoir prouver ou vérifier des propriétés sur ces modèles. Malgré les coûts supplémentaires liés au travail d'analyse et de conception en spécification formelle, l'utilisation de telles méthodes est de plus en plus justifiée pour des logiciels qui impliquent des données ou des conditions de sécurité critiques, car elles permettent d'assurer leur bon fonctionnement et d'éviter ainsi des risques d'erreur **[GER,04]**.

Nous définissons par la suite quelques méthodes formelles :

2.3.1 La méthode B :

2.3.1.1 Définition :

La méthode B[1] est une méthode formelle destinée au développement de logiciels. Cette méthode englobe le processus complet de développement depuis la spécification jusqu'à l'implémentation, et permet de prouver que l'implémentation du logiciel est conforme à sa spécification **[CAS,02]**.

2.3.1.2 Le langage B :

Est un langage de spécification formel, basé sur la notion de machine abstraite. Les fondements théoriques de la méthode sont spécifiés dans le B-Book de J-R.Abrial .

Une machine abstraite représente un état spécifié par une partie statique (à l'aide de variables d'état et des propriétés d'invariance) et une partie dynamique (à l'aide d'opérations). Le langage pour la description de la statique repose sur la théorie des ensembles et sur la logique du premier ordre. Les variables sont ainsi typées par des ensembles et les invariants sont spécifiés à l'aide de conjonctions de prédicats du premier ordre **[GER,06]**.

L'état de la machine abstraite ne peut être modifié que par des opérations. Le langage permettant d'exprimer la partie dynamique est un langage de substitutions généralisées. Il permet de décrire les opérations qui font évoluer l'état du système modélisé. Lors des phases initiales de spécification, le langage est abstrait : les instructions des opérations utilisent des préconditions et de l'indéterminisme **[GER,06]**. Les différentes clauses d'une machine abstraite B sont présentées dans le tableau 2.1.

Clauses	Description
MACHINE	Nom et paramètres de la machine
CONSTRAINTS	Définition des propriétés des paramètres de la machine
SETS	Liste des ensembles abstraits et définition des ensembles énumérés

[1] *Le choix de la lettre B est un hommage à Nicolas Bourbaki, un mathématicien imaginaire, dont le nom a été utilisé par un groupe de mathématiciens français qui entreprit depuis 1939 une refonte des mathématiques.*

CONSTANTS	Liste des constantes de la machine
PROPERTIES	Définition des propriétés des constantes et des ensembles
VARIABLES	Liste des variables d'état de la machine
INVARIANT	Définition des types et des propriétés des variables
DEFINITIONS	Liste d'abréviations pour les prédicats, les expressions ou les substitutions
INITIALISATION	Initialisation des variables d'état
OPERATIONS	Liste des opérations de la machine

Tableau 2.1 : Clauses d'une machine abstraite en B **[GER,06]**.

2.3.2 Réseaux de Pétri :

2.3.2.1. Définition :

Les réseaux de Pétri constituent un formalisme mathématique particulièrement adapté à la modélisation des systèmes où les aspects d'évènements concurrents et d'évolutions simultanées sont présents. Ils sont utilisés dans la spécification et la validation des protocoles de communication en particulier et des systèmes distribués en général. Il permettent l'évaluation des performances des systèmes discrets et même la conception des interfaces homme-machine **[SAL,02]** .

Les réseaux de Pétri permettent la vérification de certaines propriétés des systèmes qu'ils décrivent. Par exemple, ils vérifient des propriétés de sûreté **[SAL,02]**.

Un réseau de Pétri est un quadruplet R = (P, T, Pre, Post), où **[SAL,02]**:

– P est un ensemble fini de places.

– T est un ensemble fini de transitions.

– Pre : P× T ⟶ N est la fonction incidence avant.

– Post : P ×T ⟶ N est la fonction incidence arrière.

2.3.3 La méthode CCS

CCS (Calculus of Communicating Systems) a été défini par R. Milner. CCS est fondé sur l'observation d'agents qui représentent une partie unitaire d'un système concurrent à modéliser **[GER,04]**.

2.3.3.1 Agent :

La notion d'agent est vague et peut désigner des parties plus ou moins atomiques du système. Un agent peut donc être composé de plusieurs sous agents. Les deux notions fondamentales de CCS sont concurrence et communication. La concurrence est caractérisée en CCS par l'indépendance des actions d'un agent par rapport aux autres agents du système. La communication des agents en CCS est de deux types : les actions peuvent agir à l'intérieur d'un agent ou bien interagir avec ses agents voisins. Le comportement d'un système est défini en CCS par l'observation de ses actions **[GER,04]**.

Un agent contient deux ports qui permettent de communiquer avec l'extérieur : un port d'entrée et un port de sortie. Une action de label a entrant dans un agent est dénotée simplement par a, tandis qu'une action sortante de même label est dénotée par ā. Les actions a et ā sont dites *complémentaires* **[GER, 04]**.

2.3.4 La méthode CSP :

Le langage CSP (CommunicatingSequentialProcesses) est une notation utilisée pour décrire des systèmes concurrents. Le langage est supporté par quelques outils, qui permettent d'analyser et de vérifier les spécifications en cours ou existantes. CSP a été inventé par C.A.R. Hoare et développé à l'Université de Oxford dans les années 80 **[GER, 04]**.

2.3.4.1Syntaxe de CSP :

En CSP, les processus sont des entités, indépendantes les unes des autres, mais qui peuvent communiquer entre elles. Un processus peut exécuter des événements (ou actions). Les événements permettent de décrire le comportement des processus. L'ensemble des événements que le processus P peut exécuter est appelé son alphabet (ou interface) et est dénoté par α(P). Le comportement le plus simple d'un processus est de ne rien faire : un tel processus est dénoté par *STOP* **[GER,04]**.

2.3.5 La méthode de développement de Vienne :

Est un langage de spécification basée à la fois sur la théorie des ensembles et la logique des prédicats. Ses points faibles restent le non prise en compte des aspects de concurrence **[ISM,96]**. VDM est une méthode très proche de celle de Raise et de Z **[CAS, 02]**.

2.3.6 Langage Z :

Le langage Z est basé sur la théorie des ensembles et sur la logique du premier ordre. Le schéma est la notion de base des spécifications Z. Un schéma est une boîte contenant des descriptions utilisant les notations Z. Les schémas sont utilisés pour décrire les états d'un système, les états initiaux ou bien les opérations **[GER,06]**.

2.3.6.1 Statique :

La partie statique permet de définir les états et les relations d'invariant qui sont préservées lors des transitions d'états. Elle est décrite en Z sous la forme d'un schéma d'état**[GER,06]**.

2.3.6.2 Dynamique :

Les aspects dynamiques concernent les opérations, les relations entre les entrées et les sorties, et les changements d'états **GER,06]**.

2.3.7 Langage LOTOS

LOTOS[2] est un langage de spécification, généralement applicable dans les systèmes distribués et concurrents **[BOL,87]**.

2.4 Comparaison entre deux méthodes de vérification :

	Z	B
Année de parution	Accepté par l'université d'Oxford comme " BSI Standard " en 1981.	Les années 1980.
Auteurs et Affiliations	Suggéré par Abrial et après développé par l'université d'Oxford	développé par Jean-Raymond Abrial
Concepts de base	-Logique de prédicat de premier ordre. -Théorie des ensembles.	-Logique de prédicat de premier ordre. -Théorie des ensembles.

[2] Language Of Temporal Ordering Specification

Notation	*Schema Name* *Signature Part* *Predicate Part*	Machine Name Sets Variables Constants Initialisation Invariants Operations

Tableau 2.2. La comparaison entre les méthodes formelles B et Z **[PES, 15]**.

2.5 Conclusion :

La plupart des techniques formelles offrent les bases nécessaires pour vérifier et valider la spécification d'une architecture logicielle, elles sont basées sur des notations mathématiques qui exigent des connaissances et des expertises assez importantes. Dans ce chapitre, nous avons présenté des méthodes de vérification ainsi une étude comparative entre deux techniques et nous avons choisi la Méthode B pour l'utiliser dans notre projet comme une technique de vérification.

Nous présentons, dans ce qui suit un aperçu sur les transformations de modèle en présentant le principe et les concepts fondamentaux de cette démarche.

Chapitre 3 :

L'ingénierie dirigée

par

les modèles

3.1 Introduction :

Dans l'introduction de ce travail, nous avons indiqué que notre proposition repose sur les transformations de modèles qui sont au cœur de la démarche de l'Ingénierie Dirigée par les Modèles (IDM)[1] . Dans ce chapitre, nous présentons le principe et les concepts fondamentaux de cette démarche.

3.2 Principes de l'approche :

L'IDM spécifie l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles **[AZA,07]**. L'IDM a pour but d'apporter une nouvelle vision permettant de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux coûts excessifs de migration technologique **[MEN, 10]**. En ce sens, l'IDM fait évoluer l'usage des *modèles*, qui peuvent être interprétés ou transformés. Pour que ces modèles peuvent être interprétés ou transformés, il faut que ces modèles soient définis dans un langage. Ce langage est appelé métamodèle dans la littérature relative à la démarche de l'IDM. Les métamodèles utilisés pour la définition de modèles peuvent être différents. Pour gérer cette diversité, l'IDM préconise d'utiliser un langage commun appelé métamétamodèle pour décrire tous les métamodèles impliqués dans la description des modèles afin de permettre leur intégration dans les outils de mise en œuvre du processus de construction.

Les travaux menés sur l'IDM font suite à la définition de MDA[2] par l'OMG[3]. MDA a recours aux différents standards de l'OMG afin de décrire les démarches basées sur l'ingénierie des modèles. En effet, MDA est considéré comme un exemple particulier d'ingénierie dirigée par les modèles. Néanmoins, la plupart des travaux sur l'IDM font référence à MDA.

[1] Model Driven Engineering (MDE) en anglais.
[2] Model Driven Architecture .
[3] The Object Management Group (OMG), cf. http://www.omg.org/.

3.3 Définition des concepts fondamentaux :

Nous proposons de définir les concepts de base de l'IDM afin d'appréhender de manière optimale cette approche. Cependant il n'existe pas de définitions standards de ces concepts .Cette section présente donc, les définitions qui nous semblent être acceptées par la plupart des utilisateurs de l'IDM dans la littérature.

3.3.1 Modèle :

Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé. Un modèle représente donc un système selon un certain point de vue, à un niveau d'abstraction facilitant par exemple la validation ou la conception de cet aspect particulier du système **[LAF, 13]**. Un système est une construction théorique que forme l'esprit sur un sujet (ex. : une idée expliquant un phénomène physique et représentée par un modèle mathématique) **[TUR, 08]**.

3.3.2 Métamodèle :

La notion de modèle dans l'IDM fait explicitement référence à la notion de langage bien défini. En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être clairement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé métamodèle **[LAF, 13]**.

Un métamodèle est un modèle de modèles. C'est aussi un langage utilisé pour décrire des modèles, dans la mesure où il englobe un ensemble de concepts nécessaires à la description d'une famille de modèles donnée. A titre d'exemple, UML[4] est un métamodèle qui offre des concepts permettant de décrire les différents modèles (Diagramme de classe, Diagramme de cas d'utilisation, …) d'un système **[DLC ,09]**.

La notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée conformeA **[LAF, 13]**.

[4] Unified Modeling Language.

3.3.3 Métamétamodèle :

Un métamétamodèle est un modèle qui décrit un langage de méta-modélisation, c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même **[LAF, 13]**.

Afin de contrôler et d'éviter l'émergence de méta-modèles incompatibles, le Model Driven Architecture (MDA) de l'OMG a, entre autres, proposé un langage de définition de méta-modèle sous le forme d'un modèle : le méta-méta-modèle MOF (Meta-Object Facility). Aussi, pour limiter le nombre de niveaux d'abstraction, ce méta-méta-modèle a une propriété de méta circularité : il se décrit lui-même **[LAF, 13]**.

3.3.4 Transformation de modèles

Les transformations de modèles sont au cœur de l'approche de l'ingénierie dirigée par les modèles **[DJM+,11]**.Hubert Kadima (2005) nous donne les définitions suivantes :

> **Définition 1.** Une transformation de modèle est une opération qui consiste à générer un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources conformément à une définition de transformation.
>
> **Définition 2.** Une définition de transformation est un ensemble de règles de transformation qui décrivent globalement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible.
>
> **Définition 3.** Une règle de transformation est une description de la manière dont un ou plusieurs éléments du modèle source peuvent être transformés en un ou plusieurs éléments du modèle cible.

3.4 Les quatre niveaux des modèles

L'application de l'approche de l'IDM repose sur une architecture à quatre niveaux qui structure les différents modèles qui peuvent être produits . Cette architecture (Figure 3.1) comporte quatre niveaux d'abstraction que nous allons détailler dans ce qui suit.

Le niveau M0

Le niveau M0, représente les objets du monde du réel. Il représente, par exemple, un compte bancaire avec son numéro et son solde actuel (Figure3.1) **[AZA,07]**.

Le niveau M1

C'est au niveau M1 que les modèles sont édités. Ces modèles sont conformes aux métamodèles définis au niveau M2. Ainsi, MDA considère que si l'on veut décrire des informations appartenant au niveau M0, il faut d'abord construire un modèle appartenant au niveau M1. De ce fait, un modèle UML (comme le diagramme de classes ou le diagramme d'état/transition) est considéré comme appartenant au niveau M1. Il représenterait des objets manipulés dans le monde réel (décrits au niveau M0) **[AZA,07]**.

Le niveau M2

Le niveau M2, est le lieu de définition des métamodèles. Un métamodèle peut être considéré comme un langage spécialisé pour un aspect du système. Il peut aussi décrire les aspects spécifiques aux différents domaines, chaque aspect étant pris en compte dans un métamodèle spécifique. Les métamodèles contenus dans le niveau M2 sont tous des instances du niveau M3 (notons qu'au niveau M3, il ne peut y avoir qu'un seul métamétamodèle). Dans le cadre de MDA, c'est le métamodèle d'UML qui est le plus utilisé, celui-ci définit la structure interne des modèles UML **[AZA,07]**.

Le niveau M3

Le niveau supérieur, M3 correspond au métamétamodèle. Il définit les notions de base permettant l'expression des métamodèles (niveau M2), et des modèles (M1). Pour éviter la multiplication des niveaux d'abstraction, le niveau M3 est réflexif, c'est-à-dire qu'il se définit par lui-même. Le plus souvent, c'est le métamétamodèle MOF (*Meta Object Facility*) qui est utilisé. Celui-ci est standardisé par l'OMG. Cependant d'autres méta-métamodèles ont été proposés tels que Ecore (EMF) et OWL **[AZA,07]**.

Afin d'avoir une idée plus précise de l'architecture à quatre niveaux d'OMG, nous illustrons par la figure 3.1, les modèles pouvant appartenir à chaque couche.

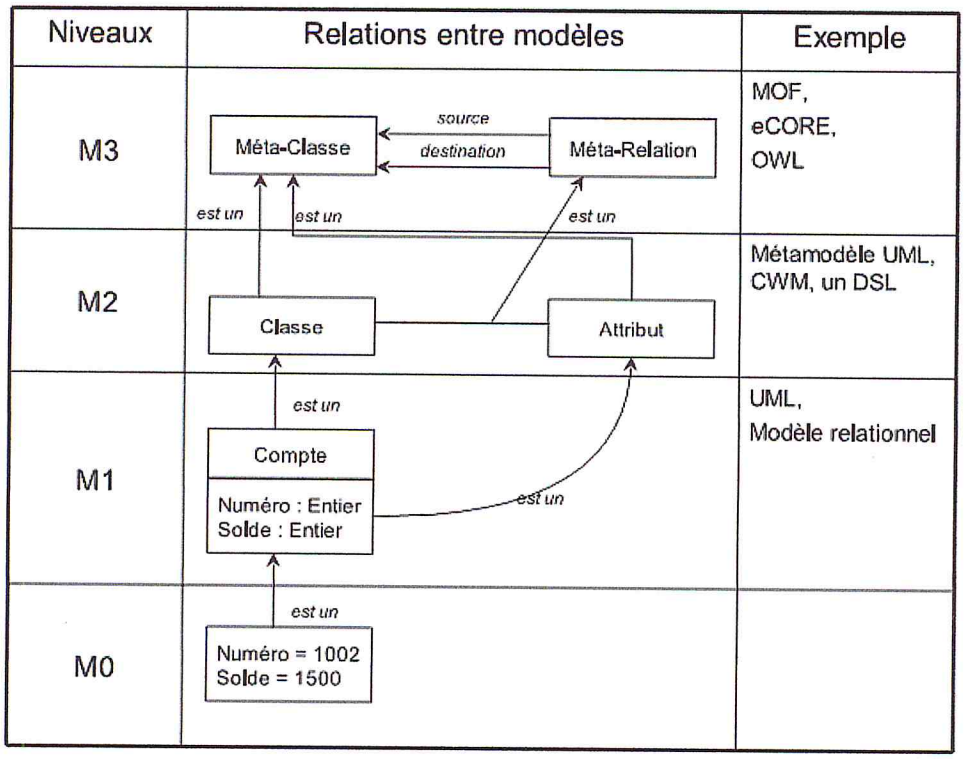

Figure 3.1: Les quatre niveaux de MDA **[AZA,07]**.

3.5 Les relations entre les modèles

Il est communément admis qu'un modèle est la simplification subjective d'un système. Dans ce contexte, le modèle sert à obtenir des réponses par rapport au système qu'il représente **[BEG,01]**. Par exemple, une carte géographique peut jouer le rôle de modèle, alors que la région étudiée jouera celui de système modélisé. De même qu'une carte elle-même peut jouer le rôle du système modélisé **[AZA,07]**comme le montre la figure 3.2. Dans cette figure, la carte est elle même représentée par un schéma XML **[AZA,07]**.

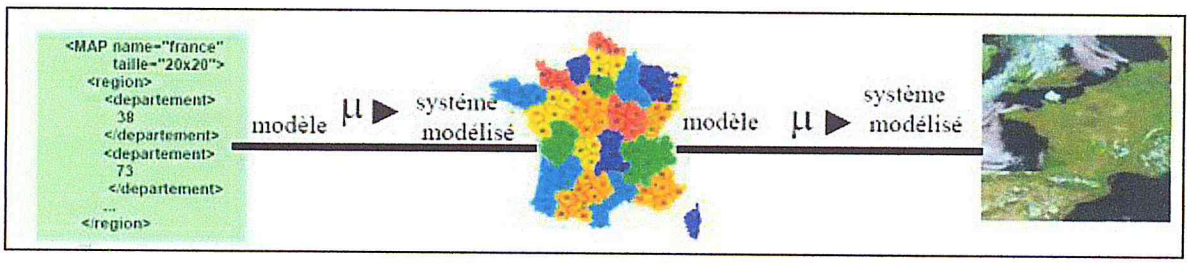

Figure 3.2: La relation μ **[BBB+,05]**.

La relation μ a été ici combinée deux fois, la carte est un modèle de la France, et le fichier XML à gauche est un modèle de la carte, pourtant on n'a pas fait intervenir la notion de langage ou de métamodèle. Le modèle en XML n'est pas un métamodèle de la France. C'est un modèle d'un autre modèle. Contrairement à une idée parfois véhiculée, un métamodèle n'est pas un modèle d'un modèle **[BBB+,05]**, un métamodèle est un modèle qui définit le langage qui exprime le modèle. Ainsi, la relation "représentation de" peut décrire le lien entre la couche M0 et M1, mais ne décrit pas le lien entre les couches M1, M2 et M3 **[AZA,07]**.

3.5.1 La relation "*être conforme à*"

La relation χ "*être conforme à*" permet d'assurer qu'un modèle est correctement construit. A partir de ce moment, il devient envisageable de lui appliquer des transformations automatisées **[AZA,07]**. La figure 3.3 démontre la relation entre une carte géographique et son métamodèle qui est constitué d'une notation graphique représentant les régions et les départements.

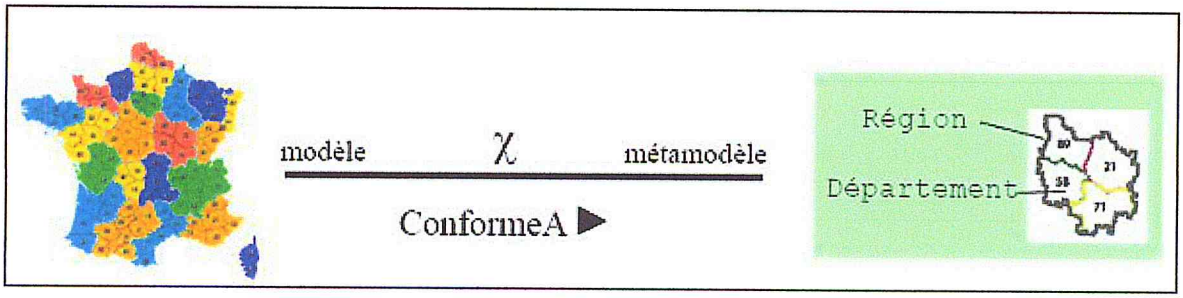

Figure 3.3: La relaion χ **[BBB+,05]**.

3.6 Transformations de modèles

Une transformation de modèle est une opération qui Prend en entrée des modèles (source) et fournit en sortie des modèles (cibles) Généralement un seul modèle source et un seul modèle cible.

Partant de la nature des métamodèles source et cible, on distingue encore selon une classification **[DLC,09]** les transformations dites endogènes et exogènes combinées à des transformations dites verticales et horizontales**[DLC,09]**.

Transformations endogènes : Une transformation est dite endogène si les modèles cible et source sont issus du même métamodèle (Figure 3.6). Une transformation endogène permet **[GUE,12]**:

- L'optimisation : la transformation a pour but d'améliorer par exemple la qualité d'exécution (en termes de performance) du modèle, tout en préservant la sémantique du modèle ;
- La restructuration : la transformation entraine un changement de la structure interne du modèle afin d'améliorer la qualité de certaines caractéristiques comme la modularité et la réutilisation, sans modifier le comportement du modèle ;
- La simplification : la transformation va permettre de simplifier la complexité de la syntaxe du modèle.
- Exemple : transformation d'un modèle UML en un autre modèle UML

Transformations exogènes : Une transformation est dite exogène si les modèles cible et source possèdent des métamodèles différents (Figure 3.6).Une transformation exogène permet **[GUE,12]**:

- La migration : la transformation d'un modèle écrit dans un certain langage en un modèle écrit dans un autre langage, tout en gardant le même niveau d'abstraction.
- La synthèse : la transformation d'un modèle de haut niveau, très abstrait (spécification, modèle fonctionnel) vers un modèle de bas niveau plus concret (code généré, exécutable).
- La rétro-ingénierie : il s'agit d'une transformation de synthèse inverse qui permet d'extraire des spécifications de haut niveau d'un modèle de bas niveau.
- Exemples : Transformation d'un modèle UML en programme Java, transformation d'un fichier XML[5] en schéma de BDD.

Transformation verticale Une transformation est dite verticale si elle met en jeu différents niveaux d'abstraction dans la transformation **[GUE,12]**.

Transformation horizontale Une transformation est dite horizontale lorsque les modèles source et cible impliqués dans la transformation sont au même niveau d'abstraction. La restructuration, la normalisation et l'intégration des patrons sont des exemples de transformation endogène et horizontale ; tandis que la migration des plates-formes et la fusion de modèles sont des exemples de transformation exogène et horizontale **[GUE,12]**.

[5] eXtended Markup Language.

La figure 3.4 ci- dessous résume les combinaisons possibles entre transformations de modèles.

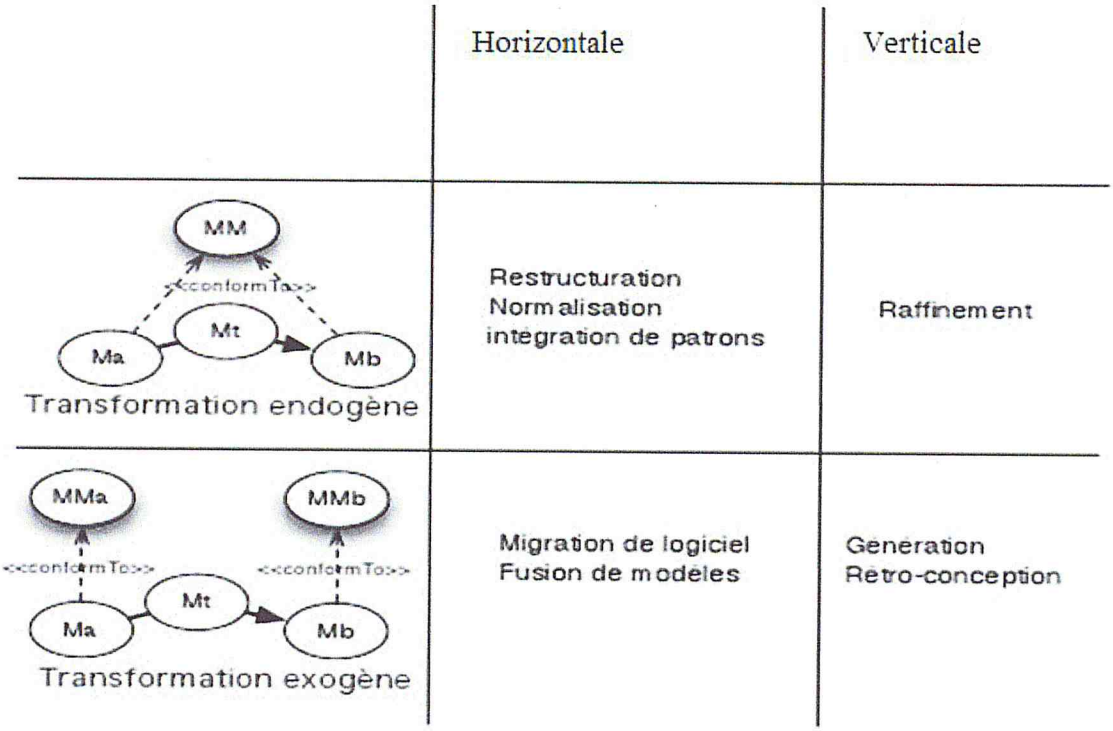

Figure 3.4 : Taxonomie des transformations de modèles **[COM,08]**.

3.7 Spécification des règles de transformation :

Ce qui différencie les différentes approches permettant l'élaboration des transformations de modèles est la façon dont sont spécifiées les règles de transformations **[GUE, 12]**. Dans **[BLA, 05]** trois approches de transformations sont retenues : l'approche par programmation, l'approche par template et l'approche par modélisation **[DLC ,09]**.

L'approche par programmation consiste à utiliser les langages de programmation en général, et plus particulièrement les langages orientés objet. Dans cette approche, la transformation est décrite sous forme d'un programme informatique à l'image de n'importe quelle application informatique (exemple Java). Cette approche reste très utilisée car elle réutilise l'expérience accumulée et l'outillage des langages existants.

L'approche par template consiste à définir des canevas des modèles cibles souhaités. Ces canevas sont des modèles cibles paramétrés ou des modèles template. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres

par les valeurs d'un modèle source. Cette approche par template est implémentée par exemple dans Softeam MDA Modeler[6].

L'approche par modélisation consiste quant à elle à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs, en exprimant leur indépendance vis-à-vis des plates-formes d'exécution. Le standard MOF 2.0 QVT de l'OMG a été élaboré dans ce cadre et a pour but de définir un métamodèle permettant l'élaboration des modèles de transformation de modèles. A titre d'exemple, cette approche a été choisie par le groupe ATLAS à travers son langage de transformation de modèles ATL.

3.8 Rapide panorama de l'ingénierie des modèles :

Il existe actuellement de multiples langages dans le paysage de l'ingénierie des modèles. Une taxonomie de ces langages peut être organisée autour de trois axes principaux **[GUE,12]**:

- les langages de métadonnées avec par exemple EMOF[7] de l'OMG, ou les schémas XML du W3C[8];
- les langages de transformation QVT[9] issu de l'OMG , ATL issu de l'INRIA , ou XSLT[10] issu duW3C;
- les langages de requête avec par exemple OCL issu de l'OMG ou XQUERY issu du W3C .
- les langages liés à un domaine/espace technologique par exemple XSLT dans le domaine XML, AWK pour fichiers texte ...
- Atelier de méta-modélisation avec langage d'action avec par exemple Kermeta .

[6] www.objecteering.fr/products_mda_modeler.php
[7] Essential MOF.
[8] World Wide Web Consortium
[9] Query/View/Transformation.
[10] Extensible Stylesheet Language Transformations

3.9 Conclusion

Dans ce chapitre, nous avons présenté sommairement le principe et les concepts essentiels et stables de l'IDM.

Ainsi, cette technique d'ingénierie semble être la plus adaptées pour répondre à notre objectif qui s'appuie sur une transformation exogène dont le métamodèle source est celui de X3ADL (ADL de l'approche IASA) et le métamodèle cible est celui langage B .

Dans le chapitre suivant, nous allons exposer notre approche basée sur les transformations de modèles.

Chapitre 4 :

Transformation du modèle d'architectures logicielles au modèle de vérification

4.1 Introduction :

La transformation de modèle est une opération fondamentale dans toute approche orientée modèles. Elle permet de générer un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources. Dans notre cas, une transformation de modèle est nécessaire pour passer d'un modèle source d'architecture logicielle (X3ADL[1]) à un modèle cible du langage B pour la vérification des propriétés après un changement évolutif qui survient dans l'architecture logicielle. Ce chapitre est consacré pour spécifier les éléments essentiels de notre proposition.

4.2 Style architectural :

La prise en compte de la complexité croissante des systèmes distribués, dynamiques et évolutifs et les contraintes inhérentes de ces systèmes font qu'il est nécessaire de pouvoir disposer d'un support permettant de *cadrer* les changements dynamiques et évolutifs qui peuvent survenir au sien de ces systèmes. Pour cela, nous proposons un style architectural, à base de composants, pour la définition des types de composants pouvant intervenir dans le système et des connexions entre ces composants. Il définit aussi l'ensemble des propriétés architecturales qui doivent être satisfaites par toutes les configurations appartenant à ce style. Le méta-modèle du style architectural étend l'ADL de l'approche IASA X3ADL. Il est décrit par un ensemble de concepts caractérisant la structure d'une architecture logicielle selon l'approche IASA, basé sur XML, ce langage définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture en termes de composants, d'interfaces, et de connecteurs.

4.3 Le Méta-modèle :

4.3.1 Le Méta-modèle de style architectural :

La figure 4.1 décrit le métamodèle du style architectural comme suit :

[1] eXtensible Architecture, Aspect and Action Description Language : C'est un langage riche, extensible et souple qui rejoint la famille des langages de description d'architecture basés sur le langage XML.

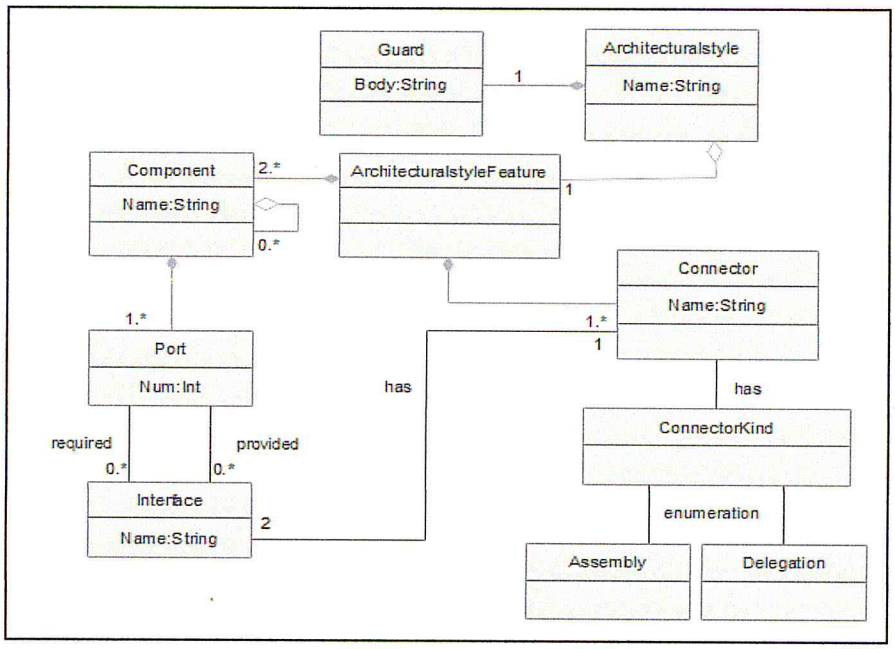

Figure 4.1. Le méta-modèle de style architectural **[HAD,08]**.

La méta-classe **<ArchitecturalStyle>** est composée de deux méta-classes **<Guards>**et**<ArchitecturalStyleFeature>**. La méta-classe**<Guards>** décrit les contraintes de l'architecture que le système doit respecter durant son évolution. La méta-classe**<ArchitecturalStyleFeature>** est composé par les méta-classes "Component" et "Connector". Il spécifie l'ensemble des types de composants et de connecteurs qui constituent le style architectural d'un système **[HAD,08]**.

La méta-classe "Component", éventuellement composée de plusieurs composants, représente une partie modulaire d'un système. Chaque "Component" a une ou plusieurs interfaces fournies et/ou requises exposées par l'intermédiaire de ports. La méta-classe "Port" représente le point d'interaction pour un composant. La cardinalité [1..*] entre "Component" et "Port" exprime qu'un composant peut avoir un ou plusieurs ports. La méta-classe "Interface" représente l'interface d'un composant **[HAD,08]**.

Une interface peut être soit de type fourni +/provided ou requis +/required. La méta-classe "Connector" définit un lien qui rend possible la communication entre deux ou plusieurs composants. La méta-classe "ConnectorKind" spécifie deux types de connecteurs. Les connecteurs de délégation "Delegation Connector" et les connecteurs d'assemblage "AssemblyConnector".Un connecteur de type "Delegation" exprime un

lien entre deux composants partant d'une interface requise vers une interface requise ou d'une interface fournie vers une interface fournie. Un connecteur de type "Assembly" exprime un lien entre deux composants partant d'une interface requise vers une interface fournie **[HAD,08]**.

4.3.2 Le Méta-modèle de langage B :

La figure 4.2 décrit le métamodèle du langage B comme suit :

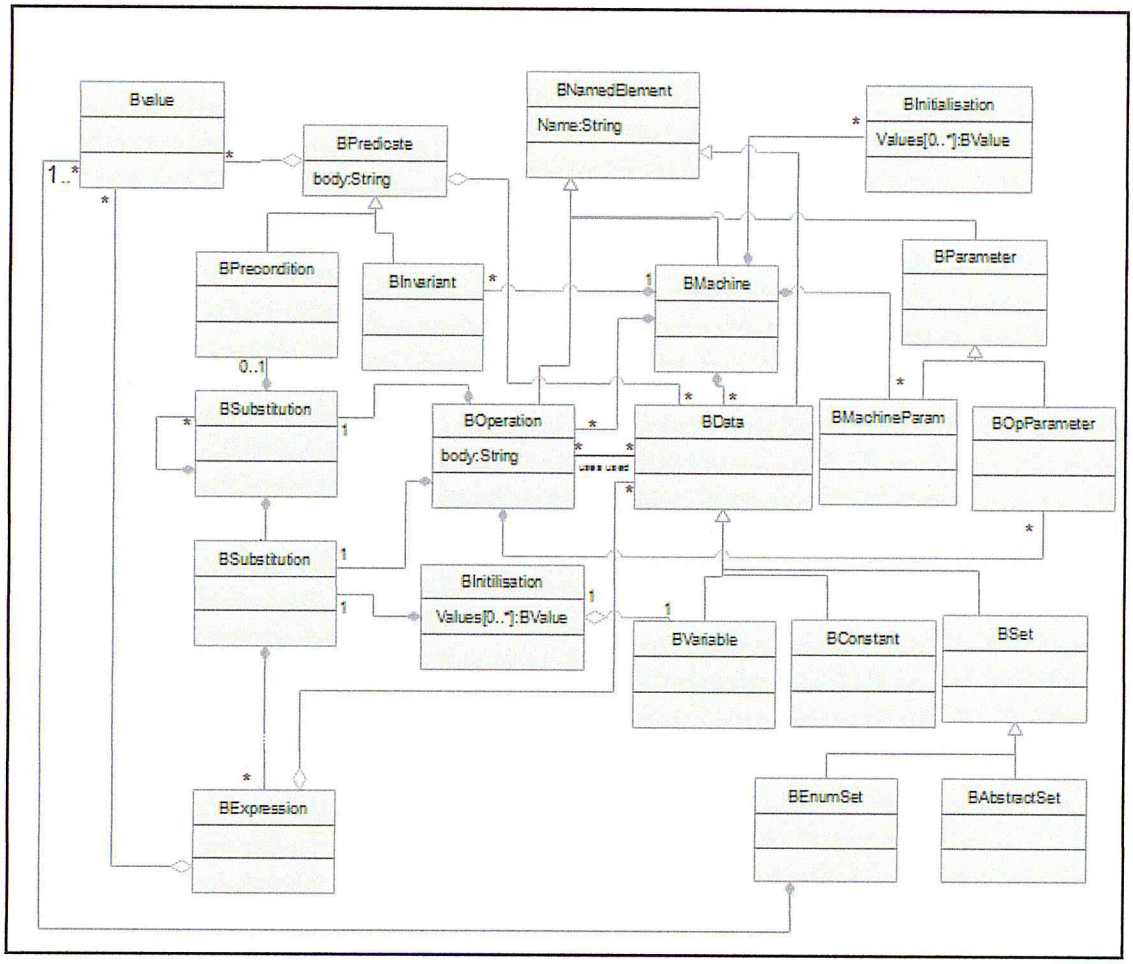

Figure 4.2. Le Méta –modèle du langage B **[IDA,06]**.

4.3.2.1 Spécification de machines abstraites :

Une machine abstraite, représentée par la méta-classe BMachine, est constituée de plusieurs clauses permettant de spécifier les parties statiques et dynamiques du système. La partie statique correspond aux déclarations d'ensembles, de constantes et de variables et une caractérisation de ces données en termes de propriétés des constantes (clause

PROPERTIES) et d'invariants. Nous ne faisons pas cette distinction par clauses au niveau du méta-modèle relatif aux machines abstraites. Nous définissons la méta-classe abstraite BData pour spécifier toutes les données déclarées au niveau d'une machine B (ensembles abstraits, constantes et variables). Quant à la partie dynamique, elle est spécifiée par les deux méta-classes BOperation et BInitialisation qui permettent de représenter respectivement les opérations et l'initialisation d'une spécification B **[IDA,06]**.

4.4 Description du méta modèle :

4.4.1 Le langage XML :

Pour la spécification de nos modèles, nous avons choisi d'utiliser le langage XML(eXtended Markup Language). Le choix XML est justifié par le fait que notre modèle source X3ADL est basé sur une notation XML. De plus, le langage XML permet de structurer les documents de manière logique et arborescente ; le document XML est un ensemble de nœuds imbriqués correspondant à des éléments signifiants sur le plan structurel et sémantique. C'est uniquement une structuration du contenu, sans propriétés de mise en page ; diverses présentations du même document pourront être générées ensuite. Le rôle de chaque élément, le type de la valeur, les liens entre éléments sont précisés par des attributs. XML est un standard ouvert qui s'est imposé dans l'échange de documents. Intégralement basé texte, il est indépendant des formats des fichiers binaires ou des systèmes d'exploitation11. Il s'associe à n'importe quel jeu de caractères, notamment Unicode. Il est donc pérenne pour le stockage de documents à long terme et interopérable. Il est de plus modulaire et extensible : c'est un métalangage dont les bases peuvent être utilisées pour créer d'autres langages reposant sur les règles du XML **[MOR,07]**, ce qui nous permet d'intégrer facilement de nouveaux concepts au niveau de la description de nos modèles.

4.4.2 Règles pour le style architectural

Nous proposons une représentation de notre style architectural en DTD comme suit :

```
< ! DOCTYPE ArchitectureLogicielle [

< ! ELEMENT  architectureStyle ( architectureStyleFeature, guards) >

< ! ELEMENT  guards (constraints)* >
```

```
< ! ELEMENT  architectureStyleFeature (components , connecteurs) >

< ! ELEMENT  components (component)+ >

< ! ELEMENT component (ports) + >

< ! ELEMENT interface (provided|required) >

< ! ELEMENT  connectors  (connector )+  >

< ! ELEMENT  connector  (interaction)*  >

< ! ELEMENT  interaction EMPTY>

< ! ELEMENT ports (port) + >
< ! ELEMENT port (interface) + >

<! ATLIST component
name   ID    #required
                    Type ("atomique"|"composite") #required >

<! ATLIST connector
name   ID    #required
connectorKind ("DelegationConnector" |"AssemblyConnector") #required >

<! ATLIST port
name   ID    #required>

<! ATLIST connector
name   CDATA    #required
source   CDATA    #required
target   CDATA    #required   >

<! ATLIST interaction
name   CDATA    #required
sequenceAction    CDATA    #required>

<! ATLIST interface
name   ID    #required
                    Type  CDATA    #required>

] >
```

4.5 Les Règles de Transformation

Une règle de transformation permet de décrire la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible. Dans notre cas, les constructions dans notre modèle source peuvent être transformées en constructions dans notre modèle cible à l'aide des règles de transformation suivantes :

- Chaque **ArchitecturalStyle** dans l'architecture logicielle devient dans le langage B une **Nom Du MACHINE.**
- Chaque **component** dans l'architecture logicielle devient dans le langage B un **SET.**
- Chaque **connector** dans l'architecture logicielle devient dans le langage B est un **SET.**
- Chaque **Interface** dans l'architecture logicielle devient dans le langage B une **VARIABLE.**
- Chaque **port** dans l'architecture logicielle devient dans le langage B une **SET.**
- Chaque **Guards** dans l'architecture logicielle devient dans le langage B une **VARIABLE.**

Style architectural	**Langage B**
ArchitecturalStyle	Nom de la machine
Guards	Dans la clause INVARIANT
Component	Un ensemble de base (clause SETS)
Connector	Un ensemble de base (clause SETS)
Port	Un ensemble de base (clause SETS)
Interface	une variable est définie (dans la clause VARIABLES)

Tableau 4.1. Les règles de transformation.

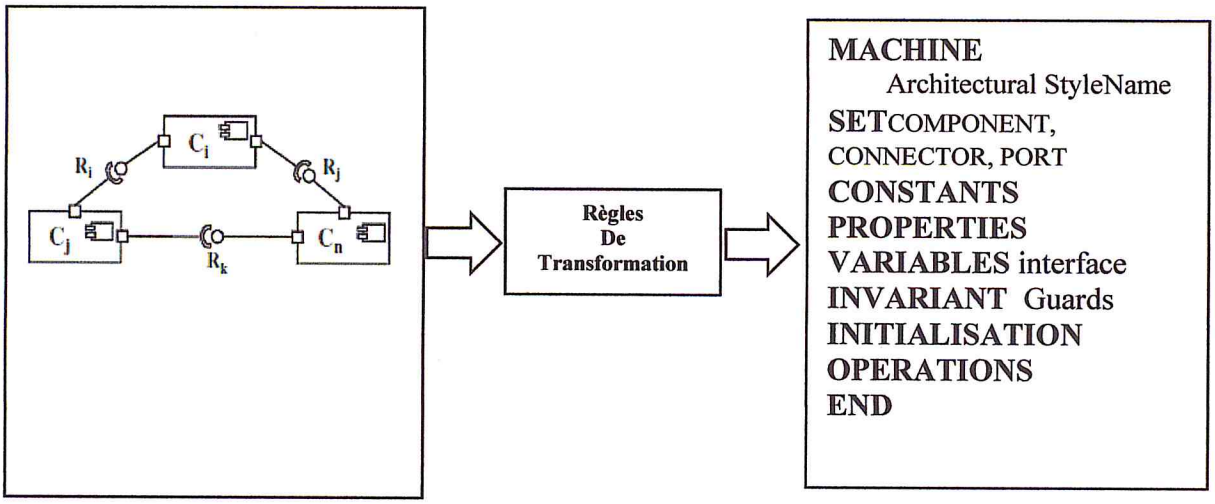

Figure 4.3 : La transformation d'un modèle architectural au langage b (machine abstraite).

- **Exemple d'une machine abstraite :**

Cet exemple illustre une machine abstraite qui représente un style client-serveur comme suit **[MHU+,14]** :

MACHINE Arch_concepts
INCLUDES Basic_concepts
SETS ARCHS; COMPS; COMP_NAMES
VARIABLES
architecture,arch_components,arch_connections,component;
comp_name; connection; comp_interfaces; client; server
arch_clients; arch_servers
INVARIANT
/* Un composant « component » a un nom et un ensemble d'interfaces */
component⊆COMPS ^
comp_name∈ component → COMP_NAMES ^
comp_interfaces ∈ component ↣P(interface) ^
/* Un client (resp. serveur « server») est un couple d'un composant « component » et une interface */
client∈ component ↔ interface ^
server∈component ↔ interface ^
/* Une connexion « connection » est une relation entre un client et un serveur « server » */
connection∈client ↔ server ^
/* Une architecture a un ensemble de composants « components » et des connections */
architecture⊆ARCHS ^
arch_components∈architecture → P (component) ^
arch_connections ∈ architecture →P(connection)
/* Arch clients (resp. arch servers) est la liste des clients connectés (resp. serveurs « servers ») avec une architecture */
arch_clients∈ architecture → P (client) ^
arch_servers∈architecture→P (server)

Pour assurer une évolution valide et fiable, des changements d'architectures logiciels doivent être vérifiées, cette vérification doit être effectuée avant et après le changement en se basant sur des règles d'évolution. Grace à la transformation de modèles qui nous a permis d'instancier le modèle B correspondant à une description d'architecture logicielle. Cette intégration permet de proposer un environnement de développement conjuguant les avantages des approches IDM et formelle : la conception d'architectures avec l'outillage de X3ADL (modeleur graphique) ; la vérification des architectures et la gestion de l'évolution avec l'outillage de B (animateur, model-checker, solver).

Règles d'évolution sont des opérations spécifiques qui sont composés d'opérations de manipulation de modèle. Ils gèrent et contrôlent l'accès à ces opérations en utilisant des conditions préalables liées au changement, selon son origine, le niveau, ou le sujet.

Prédicat qui doit être satisfaite par le modèle architectural après évolution règles d'évolution sont des opérations spécifiques qui sont composés d'opérations de manipulation de modèle. Ils gèrent et contrôlent l'accès à ces opérations en utilisant des conditions préalables liées au changement, selon son origine, le niveau, ou le sujet. L'objectif d'évolution fixe toutes les conditions qui doivent être remplies par le modèle architectural après le changement. Fondamentalement, il est composé de deux parties : les propriétés d'architecture (à savoir, la consistance et la cohérence) et la post-condition du changement initié.

4.6 Les règles d'évolution :

Le concept de règle d'évolution permet d'exprimer et de spécifier les évolutions qui peuvent être menées sur une architecture logicielle. Une règle d'évolution permet la description de l'application d'une opération d'évolution (ajout/suppression/modification/ substitution) sur un élément architectural, en spécifiant les conditions nécessaires pour le faire, ainsi que les éventuels impacts qu'elle peut engendrer sur les autres éléments architecturaux **[SAD,07]**.

Pour exprimer et spécifier les évolutions qui sont menées sur l'architecture logicielle nous avons défini un ensemble de règles d'évolution basé sur le langage B comme spécifié ci-dessous :

a) L'ajout d'un composant :

Pour ajouter un composant (R) il faut spécifier le nom du composant d'abord et vérifier comme un précondition que ce composant ainsi que ses interfaces n'existent pas dans l'ensemble de composant de notre architecture si c'est le cas alors on l'ajout directement à l'ensemble.

Addcomponent(R) =

PRE
c∈ COMPONENT – componentΛ
(!∃i /.i∈interface c⟶i ∉ client Λc⟶i ∉ server)
THEN
component := componentU{c}||
client :=client U{c⟶i} ||
server :=server U{c⟶i}
END

b) La suppression d'un composant :

Pour supprimer un composant (R) il faut spécifier le composant à supprimer (nom) d'abord et vérifier comme un précondition que ce composant ainsi que ses interfaces (client ou serveur) existe dans l'ensemble de composant de notre architecture si c'est le cas alors on le supprime directement de l'ensemble.

RemoveComponent(R) =
PRE c ∈ component Λ(∃i/ i∈interfacec⟶i∈ client V c⟶i∈ server)
Λ cn ∈ connector Λ c-> cn ∈ relation
THEN
component :=component –{c}||
comp-Interface := {c} ⩤ comp-interface ||
client :=client –{c⟶i} ||
server :=server-{c⟶i} ||
connector := connector-{cn} ||
interface:= interface-{i} || relation:= relation-{ c-> cn }
END

c) La vérification de l'existence d'un composant :

La vérification de l'existence d'un composant consiste à retourner « vrai ou true» si le composant existe sinon elle retourne « faux ou false » :

Pour vérifier l'existence d'un composant (R) il faut spécifier le nom du composant d'abord et vérifier comme un précondition que ce composant ainsi que ses interfaces (client ou serveur) existe dans l'ensemble de composant de notre architecture ainsi que ses interfaces (client ou serveur) si c'est le cas alors on obtient un « vrai » sinon la réponse est « faux » .

```
b←verifierExistence (a ,c)=
PRE a Є ARCHI Λ c Є component Λ(∃ i /interface c→iЄ client V c→iЄ server)
THEN b:=v
END
```

d)Ajout d'un connecteur :

Pour ajouter un connecteur (C) il faut spécifier le nom du connecteur d'abord et vérifier comme un précondition que ce connecteur ainsi que ses interfaces n'existent pas dans l'ensemble de connecteurs de notre architecture si c'est le cas alors on l'ajout directement à l'ensemble.

```
Addconnector( C ) =

PRE
C Є CONNECTOR - connector Λ (!∃i/ iЄ interface c⟶iЄ client
V c⟶i Є server)
THEN
connector := connector U {C}
END
```

Nous aurons pu faire d'autre règles d'évolution comme (substitution d'un composant, suppression d'un connecteur, vérifier l'existence d'un connecteur,…).

4.7Conclusion :

Nous avons présenté dans ce chapitre, les méta-modèles de l'architecture logicielle proposé ainsi celui du langage B. Nous avons présenté aussi les règles de transformation permettant de générer la description du langage B à partir de notre modèle d'architecture. Nous avons aussi présenté un ensemble de règles d'évolution basé sur le langage B qui peuvent être spécifiée avec notre approche.

Chapitre 5 :

Implémentation

5.1 Introduction :

Dans ce chapitre on base sur la possibilité d'automatiser la transformation. Il existe de nombreux langages pour effectuer les transformations de modèle. Notre choix s'est porte sur le langage ATL [1] qui est intègre à la boite à outils TOPCASED[2] d'Eclipse.

5.2 ATLAS Transformation Language (ATL) :

ATL est un langage de transformation de modèles dans le domaine de l'ingénierie dirigée par les modèles ou MDE (Model-Driven Engineering). Il fournit aux développeurs un moyen de spécifier la manière de produire un certain nombre de modèles cibles à partir de modèles sources **[FOU, 10]**.

5.3 Présentation de l'ATL :

Dans cette section, nous présentons les caractéristiques du langage ATL.

5.3.1 Structure globale de la transformation Définition :

En ATL, une transformation s'appelle un module. Un module contient un en-tête, un ensemble d'importation de bibliothèques de fonctions et un ensemble de fonctions et de règles de transformation. Les fonctions sont appelées *helper* en ATL. L'en-tête donne le nom du module de transformation et déclare les modèles source et cible. Le listing 7.1 donne un exemple d'en-tête **[JOU,06]**.

```
moduleSimpleClass2SimpleRDBMS;
createOUT : SimpleRDBMSfrom IN : SimpleClass;
```

L'en-tête commence par le mot-clé module suivi du nom du module. Ensuite, les modèles source et cible sont déclarés comme des variables typées par leurs métamodèles. Le mot-clé create indique les modèles cible. Le mot-clé *from* indique les modèles source. Dans notre exemple, le modèle cible est représenté par le variable OUT à partir du modèle source représenté par IN. Les modèles source et cible sont respectivement conformes aux métamodèles *SimpleClass* et *SimpleRDBMS*. En général, plus d'un modèle source et d'un modèle cible peuvent être listés dans l'en-tête. Les fonctions et règles de transformation

[1] *ATLAS Transformation Language.*
[2] *Toolkit in Open Source for Critical Applications & Systems Development.*

sont les constructions utilisées pour définir une transformation. Elles sont expliquées dans les deux sections suivantes **[JOU,06]**.

5.3.2 Helpers :

Le terme "Helper" provient de la spécification OCL, qui définit deux types de "Helpers" : *operation* et *attribute Helpers* **[JAB+,08]**.

Les *helpers* opération peuvent être utilisés pour définir des opérations dans le contexte d'un élément de modèle ou du module de transformation. Le rôle principal des helpers opération est de réaliser la navigation des modèles source. Ils peuvent avoir des paramètres et peuvent utiliser la récursivité **[JOU,06]**.

Les *helpers* attribut sont utilisés pour associer des valeurs nommées en lecture seule sur les éléments de modèles source. Comme les opérations, ils ont un nom, un contexte et un type. La différence est qu'ils ne peuvent pas avoir de paramètre. Leur valeur est définie par une expression OCL **[JOU,06]**.

5.3.3 Règles de transformation :

La règle de transformation est la construction élémentaire en ATL pour exprimer la logique de transformation. Les règles ATL peuvent être soit déclaratives soit impératives **[JOU,06]**.

5.4 OCL (Object Constraint Language):

OCL est un langage formel, basé sur la logique des prédicats du premier ordre, pour annoter les diagrammes UML de en permettant notamment l'expression de contraintes **[HUC , 08]**.

5.4.1 Objectif du langage

Voici les arguments avancés pour l'introduction d'OCL **[HUC , 08]** :

– Accompagner les diagrammes UML de descriptions :

- Précises.
- Non ambiguës.

– Eviter cependant les désavantages des langages formels traditionnels qui sont peu utilisables par les utilisateurs et les concepteurs qui ne sont pas rompus à l'exercice des mathématiques :

– Rester facile à écrire ...

– Et facile à lire.

Dans le cadre de l'ingénierie des modèles, la précision du langage OCL est nécessaire pour pouvoir traduire automatiquement les contraintes OCL dans un langage de programmation afin de les vérifier pendant l'exécution d'un programme **[HUC , 08]**.

5.5 Réalisation une transformation ATL avec ATL d'Eclipse :

5.5.1 Création des méta-modèles :

La création des méta-modèles est une étape très importante dans notre pour commencer le travail .

Nous avons besoin de deux méta-modèles : méta-modèle source et méta-modèle cible .

- **Méta modèle source (Ecore) :**
  Dans notre cas le méta-modèle source c'est le méta-modèle de l'archiecure logicielle

Ces méta-classes sont les suivantes :

- Architectural style.
- Guard.
- Architectural style Feature.
- Component.
- Connector.
- Interface.
- Port.
- Connector kind .
-Assembly.
-Delegation.

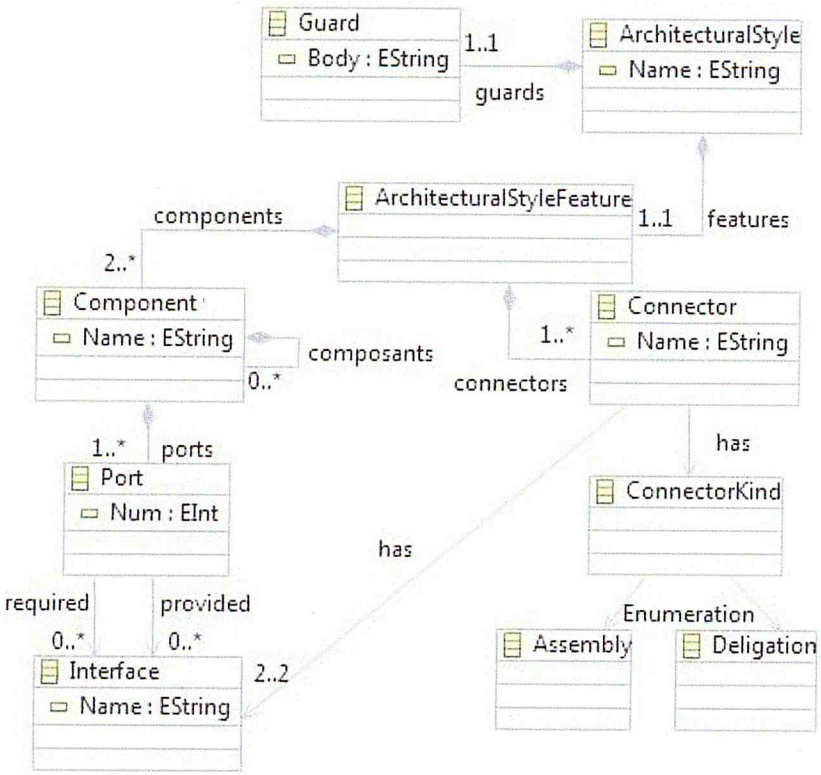

Figure5.1 : Méta-modèle d'une architecture logiciel.

- **Méta modèle cible (Ecore) :**

Le méta-modèle cible dans notre projet est le méta-modèle du langage B.

Ce méta-modèle est composé de plusieurs méta-classes comme le montre la figure 5.2.

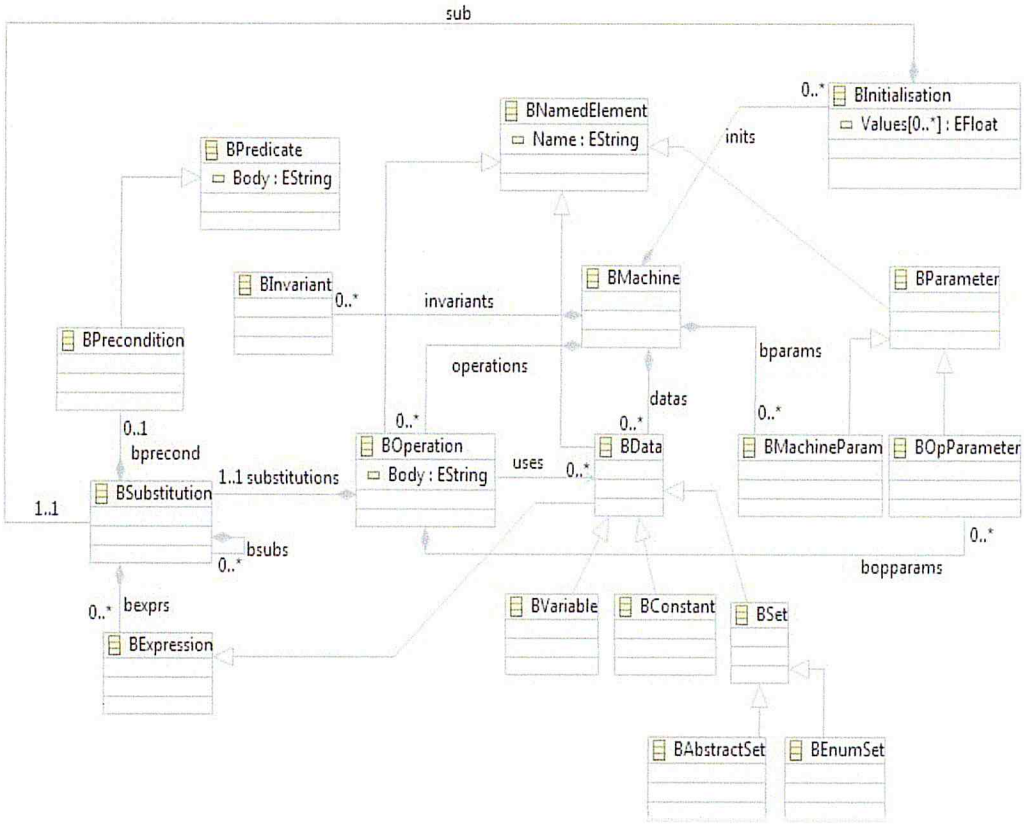

Figure 5.2 : Méta-modèle d'un langage B.

5.5.2 Création du fichier ATL :

Dans cette phase on crée un fichier ATL en sélectionnant le méta-modèle source dans la case « Input Models » et le méta-modèle cible dans la case « output Models » (Figure 5.3).

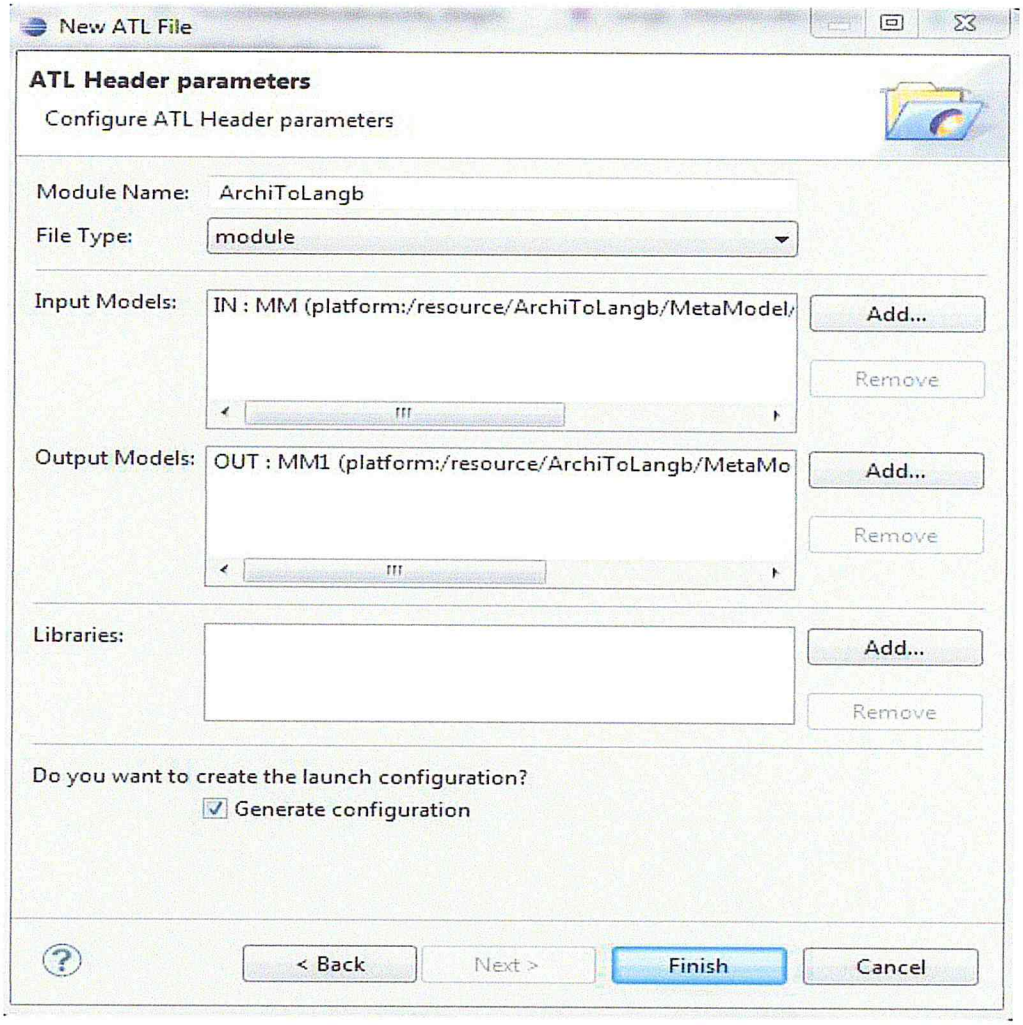

Figure5.3 : Création du fichier ATL.

5.5.3 Création des règles de transformation :

Dans cette phase on crée les règles de transformation qui décrivent les transformations de modèle source en modèle cible. Dans la partie source « partie from » on déclare l'élément de model source à transformer en utilisant la déclaration des variables.

Dans la partie cible (partie to) on déclare l'élément de modèle cible en lequel le modèle source doit être transformé à l'aide de déclaration des variables (Figure 5.4).

Myrules.atl

```
1  -- @path MM=/ArchiToLangb/MetaModel/ArchiMetaModel.ecore
2  -- @path MM1=/ArchiToLangb/MetaModel/Langb_MetaModel.ecore
3
4  module Myrules;
5  create OUT : MM1 from IN : MM;
6
7  rule architecturalStyle2Machine{
8      from A:MM!architecturalStyle
9      to M:MM1!BMachine
10     (Name <- A.Name)}
11
12 rule Guards2Invariant{
13     from G:MM!Guards
14     to I:MM1!BInvariant
15         (Name <- G.Body)
16 }
17 rule Component2Sets{
18     from C:MM!Component
19     to S:MM1!BSet
20     (Name <-C.Name)
21 }
22 rule Connector2Sets{
23     from Cn:MM!Connector
24     to S:MM1!BSet
25     (Name <-Cn.Name)
26 }
27 rule Port2Sets{
28     from P:MM!Port
29     to S:MM1!BSet
30     (Name <-P.Name)
31 }
32 rule Interface2Variable{
33     from I:MM!Interface
34     to V:MM1!BVariable
35     (Name <-I.Name)}
36
37
```

Figure 5.4 : Création des règles de transformation.

5.5.4 Compilation du fichier ATL :

Dans cette phase on fait la compilation de fichier . Elle donne toutes les informations nécessaire à l'exécution de la transformation : les chemins des fichiers ATL, modèles, métamodèles et librairies).L'onglet ATL Configuration permet de spécifier les chemins et URIs de la configuration de lancement. Les champs sont pré-remplis à partir du module ATL, mais vous pouvez en ajouter. Comme le montre la figure 5.5.

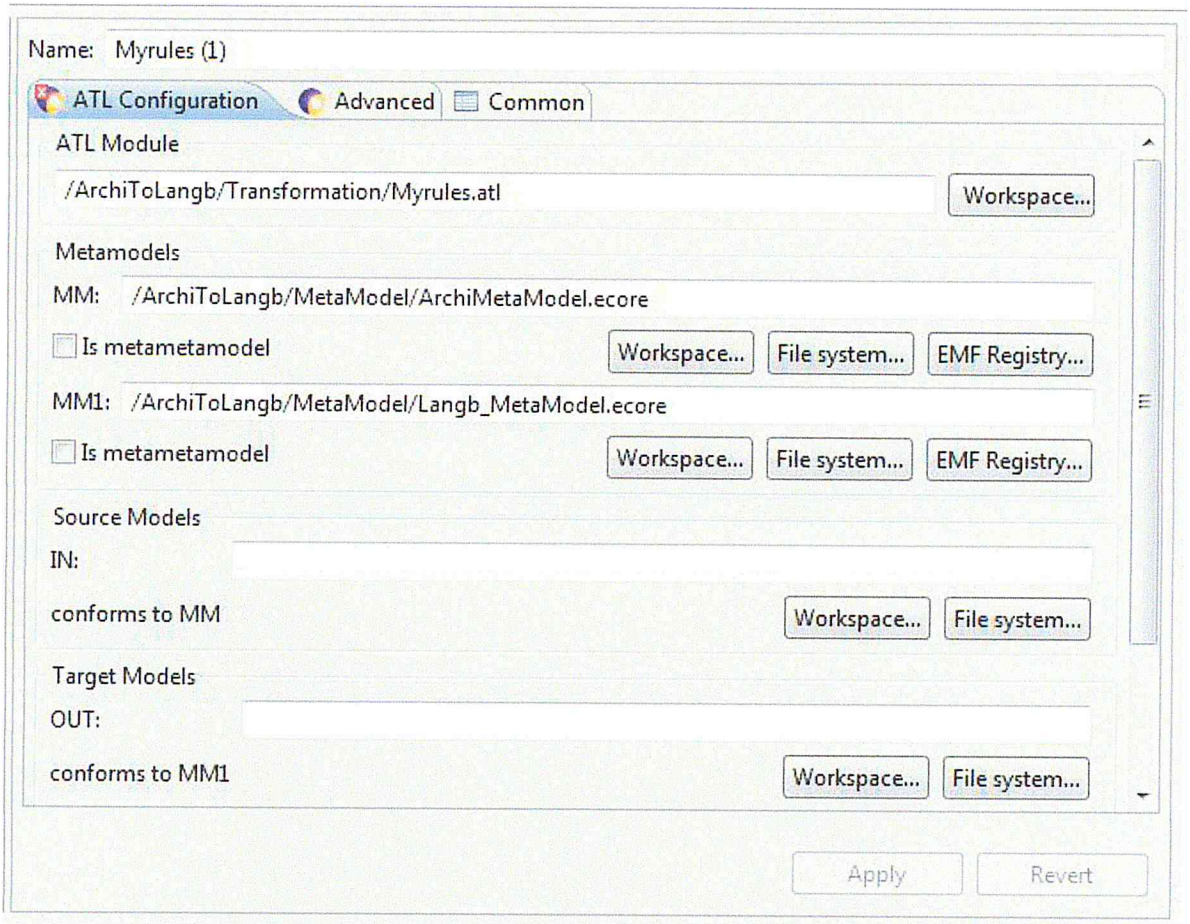

Figure5.5 : Compilation du fichier ATL.

5.6 Conclusion :

Nous avons présenté dans ce chapitre le langage de transformation utilisé dans notre travail, ce langage combine les approches de transformation relationnelles et impératives, rendant son utilisation assez simple. De plus, il est régulièrement maintenu et est intégré dans les environnements de développements tels que Eclipse.

Grace à notre étude, nous avons pu créer une transformation qui permet de faire une vérification d'architecture logicielle par transformation de modèles.

Conclusion Générale

Le projet que nous avons présenté dans ce mémoire est la vérification d'architecture logicielle par transformation de modèles.

L'objectif de notre projet était la réalisation d'un outil permettant la transformation de modèle d'un modèle de spécification d'architecture logicielle en un modèle de vérification.

Pour la réalisation de notre projet, Nous avons choisi la méthode B comme une méthode de vérification et le langage ATL(ATLAS Transformation Language) comme un langage de transformation.

Nous avons commencé par un mapping entre l'architecture logicielle et le langage B en utilisant des règles de transformation permettant de générer la description du langage B à partir de notre modèle d'architecture. Puis définir un ensemble des règles d'évolution pour spécifier les évolutions qui sont menées sur l'architecture logicielle.

Durant le travail, nous avons rencontré des difficultés avec la construction des règles d'évolutions à cause de la dépendance entre les éléments architecturaux ce qui implique que chaque operation effectuée sur un élément architectural aura une influence sur les autre éléments architecturaux.

La multitude des langages de transformation était un autre problème Car c'était difficile de choisir un langage parmi tous les langages existants.

Nous suggérons comme perspectives de notre travail de :

- Développer Cet outil pour qu'il puisse transformer un modèle de spécification d'architecture logicielle en plusieurs modèles de vérification (B, Z, Réseau de Petri…..).

Références Bibliographiques

[ALT,11] Adel ALTI, Coexistence de la modélisation à base d'objets et de la modélisation à base de composants architecturaux pour la description de l'architecture logicielle, IN : http://www.univ-setif.dz/Tdoctorat/images/stories/pdf_theses/facultes/Science/These_Doctorat_AltiAdel.pdf (2011).

[ATT,07] Christian Attiogbé , Contributions aux approches formelles de développement de logiciels : Intégration de méthodes formelles et analyse multifacette ,IN : https://tel.archives-ouvertes.fr/tel-00481602/document (2007).

[AZA,07] Selma Azaiez, Approche dirigée par les modèles pour le développement de systèmes multi-agents,IN : https://tel.archives-ouvertes.fr/tel-00519195/document (2007).

[BBB+,05] J. Bézivin, M. Blay, M. Bouzhegoub, J. Estublier, J.M. Favre, S. Gérard, and J.M. Jezequel. Rapport de Synthèse de l'AS CNRS sur le MDA (Model Driven Architecture), IN : http://www.irisa.fr/triskell/publis/2004/Jezequel04e.pdf (2005).

[BEG,01] Jean Bézivin, Olivier Gerbé , Towards a Precise Definition of the OMG/MDA Framework ,IN: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.6645&rep=rep1&type=pdf (2001).

[BLA,05] Xavier Blanc, MDA en action, (2005).

[BOL,87] Tommaso BOLOGNESI , Introduction to the ISO Specification Language LOTOS,IN:https://pdfs.semanticscholar.org/cd06/3c2257ea2b668d47f3c0febab4da365499ef.pdf (1987).

[CAS, 02] Ludovic Casset , Construction Correcte de Logiciels pour Carte à Puce Développement formel d'un vérifieur embarqué de byte code Java Card à l'aide de la

méthode B,IN : http://www.atelierb.eu/wp-content/uploads/sites/3/pdf/theseLudovic.pdf (2002).

[CHA,09] Sylvain Chardigny, Extraction d'une architecture logicielle à base de composants depuis un système orienté objet. Une approche par exploration ,IN : https://tel.archives-ouvertes.fr/tel-00456367/document (2009).

[COM,08] Benoît Combemale , Ingénierie Dirigée par les Modèles (IDM) État de l'art, IN : https://hal-univ-tlse3.archives-ouvertes.fr/hal-00371565/document (2008).

[COM,08] Benoît Combemale, Approche de métamodélisation pour la simulation et la vérification de modèle Application à l'ingénierie des procédés ,IN : http://ethesis.inp-toulouse.fr/archive/00000666/01/combemale.pdf (2008).

[DJM +,11] Maha Driss, Yassine Jamoussi, Naouel Moha, Jean-Marc Jézéquel , Henda Hajjami Ben Ghézala , Une approche centrée exigences pour la composition de services web , IN : https://hal.inria.fr/hal-00648159/file/ISIDriss11.pdf (2011).

[DLC ,09] Samba Diaw ,Rédouane Lbath, Bernard Coulette , Etat de l'art sur le développement logiciel dirigé par les modèles , IN : ftp://ftp.irit.fr/IRIT/MACAO/Article_TSI-IDM-final-coulette.pdf (2009).

[FEL , 12] Abderrahmane Feliachi , Semantics-Based Testing for Circus Synthèse ,IN : https://tel.archives-ouvertes.fr/tel-00821836/file/VA2_FELIACHI_ABDERRAHMANE_12122012_synthese_annexe.pdf (2012).

[FLE,06] Franck Fleurey , Langage et méthode pour une ingénierie des modèles fiable ,IN : https://tel.archives-ouvertes.fr/tel-00538288/document (2006).

[FOU,10] Farah FOURATI , Une approche IDM de transformation exogène de Wright vers Ada , IN : https://arxiv.org/ftp/arxiv/papers/1207/1207.6831.pdf (2010).

[GAS,94] David Garlan , Mary Shaw , An Introduction to Software Architecture,IN: http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf (1994).

[GER,04] Frédéric GERVAIS , EB4 : Vers une méthode combinée de spécification formelle des systèmes d'information,IN : https://cedric.cnam.fr/fichiers/RC658.pdf (2004).

[GER,06] Frédéric GERVAIS, Combinaison de spécfications formelles pour la modélisation des systèmes d'information ,IN : https://cedric.cnam.fr/fichiers/RC1103.pdf (2006).

[GOA , 09] Olivier Le Goaer , Styles d'évolution dans les architectures logicielles , IN : https://tel.archives-ouvertes.fr/tel-00459925/document (2009).

[GRA , 07] Mohamed GRAIET, Contribution à une démarche de vérification formelle d'architectures logicielles ,IN : https://tel.archives-ouvertes.fr/tel-00182871/document (2007).

[GUE,12] D. Guessoum , Spécification hautement flexible d'architecture logicielle,(2012).

[HAD, 08] Mohamed HADJ KACEM, Modélisation des applications distribuées à architecture dynamique : Conception et Validation ,IN : https://tel.archives-ouvertes.fr/tel-00354738/document (2008).

[HUC , 08] Marianne Huchard , Object Constraint Language (OCL) Une introduction , IN : http://www.labunix.uqam.ca/~tremblay/INF3140/Liens/coursOCL20.pdf (2008).

[IDA,06] Akram IDANI, B/UML mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B, IN : https://hal.archives-ouvertes.fr/tel-00118718v1/document (2006).

[ISM,96] Tarek BEN ISMAIL, Synthèse au niveau système et conception de systèmes mixtes logiciels/matériels, IN : https://tel.archives-ouvertes.fr/tel-00010766/document (1996).

[JAB+,08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, ATL: A model transformation tool , IN : http://universite.jejemaes.net/MA_2/SysDis%20MA/Articles_de_r%C3%88f%C3%88rence/Bezivin___ATL-_A_model_transformation_tool.pdf (2008).

[JOU, 06] Frédéric JOUAULT, Contribution à l'étude des langages de transformation de modèles, IN : file:///C:/Users/INFOMANS/Downloads/pdfNatif%20(1).pdf (2006).

[LAF,13] Pierre Laforcade, L'Ingénierie Dirigée par les Modèles et le Domain-Specific Modeling, IN : http://www-lium.univ-lemans.fr/~laforcad/graphit/wp-content/uploads/2014/08/D2-7-IDM-et-DSM-v1.pdf (2013).

[MEN,10] Ludovic MENET, Formalisation d'une approche d'Ingénierie Dirigée par les Modèles , IN : http://www.iut.univ-paris8.fr/files/webfm/recherche/linc/theseLudovicMenet.pdf (2010).

[MHU+,14] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, Huaxi (Yulin) Zhang , Formal rules for reliable component-based architecture evolution , (2014).

[MOR,07] MOREL-PAIR Catherine , Métadonnées et XML Des standards efficients de l'environnement numérique ,IN : http://www.enssib.fr/bibliotheque-numerique/documents/1842-metadonnees-et-xml-des-standards-efficients-de-l-environnement-numerique.pdf (2007).

[PEM, 07] Jorge Luis PEREZ-MEDINA, Stéphanie MARSAL-LAYAT, Transformation et vérification de cohérence entre modèles du Génie Logiciel et modèles de l'Interface Homme-Machine ,IN :http://iihm.imag.fr/publs/2007/InforsidPerezVersionFinal.pdf.pdf (2007).

[PES, 15] Tulika Pandey, Saurabh Srivastava , Comparative Analysis of Formal Specification Languages Z, VDM and B , IN : http://inpressco.com/wp-content/uploads/2015/06/Paper1082086-2091.pdf (2015).

[SAD, 07] Nassima Sadou, Evolution Structurelle dans les Architectures Logicielles à base de Composants, IN : https://tel.archives-ouvertes.fr/tel-00488005/document (2007).

[SAL,02] Aziz Salah, Genération automatique d'une spécification formelle à partir de scènarios temps-réels , IN : de http://www.info2.uqam.ca/~salah_a/Publications/thesis.pdf (2002).

[TIB,13] Chouki TIBERMACINE , Chapitre 2-Architectures logicielles : contraintes d'architecture , IN : http://www.lirmm.fr/~tibermacin/papers/2013/CT_ChapterOussalah_2013.pdf (2013).

[TUR,08] Guy TURCHA, La théorie des systèmes et systémiques: Vue d'ensemble et définitions ,IN : http://www.prof-turchany.eu/culture/La_theorie_des_systemes.pdf (2008).