N° d'ordre : 08 / 2013 – M / INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE D'ELECTRONIQUE ET DE L'INFORMATIQUE

Mémoire

Présenté pour l'obtention du diplôme de magistère

En : Informatique

Spécialité : Système Intelligent et Ingénierie du Logiciel

Par : GUEMRAOU Lila

Sujet

MAPL : UN MODELE DE DESCRIPTION D'ARCHITECTURE POUR LES PROCEDES LOGICIELS

Soutenu publiquement le 21 mai 2013 devant le jury composé de :

Mme. ALIMAZIGHI Zaia	Professeur à l'USTHB	Présidente
M. AHMED-NACER Mohamed	Professeur à l'USTHB	Directeur de mémoire
Mlle. MAHDAOUI Latifa	Maitre de Conférences/A à l'USTHB	Examinatrice
M. FEREDJ Mohamed	Maitre de Conférences/A à l'USTHB	Examinateur
Mlle. HACHICHI Assia	Maitre de Conférences/B à l'USTHB	Invitée

N° d'ordre : 08 / 2013 – M / INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE HOUARI BOUMEDIENE

FACULTE D'ELECTRONIQUE ET DE L'INFORMATIQUE

Mémoire

Présenté pour l'obtention du diplôme de magistère

En : Informatique

Spécialité : Système Intelligent et Ingénierie du Logiciel

Par : GUEMRAOU Lila

Sujet

MAPL : UN MODELE DE DESCRIPTION D'ARCHITECTURE POUR LES PROCEDES LOGICIELS

Soutenu publiquement le 21 mai 2013 devant le jury composé de :

Mme. ALIMAZIGHI Zaia	Professeur à l'USTHB	Présidente
M. AHMED-NACER Mohamed	Professeur à l'USTHB	Directeur de mémoire
Mlle. MAHDAOUI Latifa	Maitre de Conférences/A à l'USTHB	Examinatrice
M. FEREDJ Mohamed	Maitre de Conférences/A à l'USTHB	Examinateur
Mlle. HACHICHI Assia	Maitre de Conférences/B à l'USTHB	Invitée

Pour tous les êtres chers qui ne sont plus de ce monde, leur mémoire maintient notre passé vivant. Et pour tous ceux qui nous entourent, leur soutien nous aide à construire l'avenir et aller de l'avant.

Remerciements

N'étant pas initialement une spécialiste dans le domaine du génie logiciel, je n'aurais pas pu effectuer un tel travail sans le soutien professionnel et moral de plusieurs personnes.

Tout d'abord, je ne saurais trop remercier M. AHMED NACER Mohamed, professeur de l'université d'USTHB, mon directeur de mémoire, qui a accepté de m'accueillir dans son équipe de recherche pour sa direction pertinente, sa patience et sa disponibilité.

J'exprime mes vifs remerciements aux membres du jury : Mme. ALIMAZIGHI Zaia, professeur à l'USTHB, pour m'avoir fait l'honneur de présider le jury de ce mémoire ; Mlle. MAHDAOUI Latifa, maitre de conférences, à l'USTHB, et également M. FEREDJ Mohamed, maitre de conférences, à l'USTHB, pour avoir accepté d'examiner et juger mon travail.

Merci sincèrement à Mlle. HACHICHI Assia, maitre de conférences, à l'USTHB, pour sa sympathie, pour le temps qu'elle m'a consacré, pour ses conseils éclairés, et pour les corrections qu'elle a apporté à ce manuscrit.

Je remercie également Mlle. AOUSSAT Fadila, maitre assistante, à l'université de Blida, pour son aide précieuse, et pour ses conseils éclairés.

Mes amis les plus proches ne sont pas oubliés, bien au contraire. Je leurs adresse ici un grand merci. Ils ont toujours su être là, quand et comme il le faut.

Enfin, le meilleur pour la fin, mes derniers remerciements sont pour les membres de ma chère famille, pour leur soutien. Qui chacun à sa manière, par ses paroles et ses gestes, m'aident et m'encouragent depuis toujours. Et plus particulièrement, mes remerciements les plus profonds pour mes enfants Mehdi et Amira. Votre naissance au cours de cet ouvrage m'a redonné beaucoup de volonté et m'a poussé à aller devant. Désolée pour tout le temps dont je vous ai privé au profit de ce mémoire.

Résumé

Ce mémoire relate une approche pour la réutilisation et l'interopérabilité des procédés logiciels hétérogènes. Elle consiste à voir un procédé logiciel comme étant un ensemble de sous procédés manipulables via leurs interfaces externe. Ces sous procédés sont modélisés de manière abstraite, ils définissent de manière explicite les interactions entre les différents fragments d'un processus et fournissent ainsi, un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments.

Pour cela, on utilise les principes d'architecture logicielle en tant que « perspective haut niveau d'un système logiciel », reconnue comme le cœur d'une discipline à part entière et dont la principale caractéristique réside à être un modèle abstrait significatif de la structure et du comportement d'un système logiciel.

Toutefois, afin de rapprocher ces deux axes de recherche, à savoir l'ingénierie des procédés et les architectures logicielles, nous avons eu recours aux méthodes de transformation de modèles qui fournissent une migration technologique cohérente.

Ainsi, la méthode donne la possibilité d'améliorer et de concevoir de nouveaux modèles de procédés logiciels par simple intégration des composants de procédés réutilisables.

Mots clés:

Procédé logiciels, Composants de procédé, Architecture logicielle, Ingénierie dirigée par les modèles, Réutilisation, Interopérabilité.

Abstract

This document describes an approach for the reuse and interoperability of heterogeneous software processes. It is seeing a software process as a set of sub-processes which can be manipulated via their external interfaces. These processes are modeled in an abstract manner without going into implementation details, they define explicitly the interactions between the different pieces of a process and thus provide a modeling support to help designers to structure and compose different elements.

For this, we use the principles of software architecture as "high-level perspective of a software system", and whose main feature is to be a significant abstract model of the structure and behavior of a software system.

However, to bring closer these two research areas: process engineering and software architectures, we were compelled to resort to the models transformation methods. These methods offer a coherent technology migration.

Thus, the method gives the possibility of improving and of conceiving new models of software processes by simple integration of reusable process fragments.

Keywords:

Software process, Process components, Software architecture, Model driven engineering, Reuse, Interoperability.

Sommaire

Table des matières

CHAPITRE 3. ARCHITECTURES LOGICIELLES À BASE DE COMPOSANTS

CHAPITRE 4. INGÉNIERIE DIRIGÉE PAR LES MODÈLES

PARTIE II : CONTRIBUTION

CHAPITRE 5. CONCEPTION DE MAPL

CHAPITRE 6. REALISATION

CHAPITRE 7. VALIDATION

CONCLUSION ET PERSPECTIVES

BIBLIOGRAPHIE

ANNEXE

Listings

Liste des Figures

Liste des tableaux

Glossaire

ADL	:	**A**rchitecture **D**escription **L**anguage
AGL	:	**A**telier de **G**énie **L**ogiciel
AL	:	**A**rchitecture **L**ogcielle
API	:	**A**pplication **P**rogramming **I**nterface
BNF	:	**B**ackus **N**aur **F**orm
CASE	:	**C**omputer **A**ided **S**oftware **E**ngineering
CBSE	:	**C**omponent-**B**ased **S**oftware **E**ngineering
CMOF	:	**C**omplete **M**eta **O**bjet **F**acility
DTD	:	**D**ocument **T**ype **D**efinition
ECP	:	**E**nvironnement de développement logiciel **C**entré **P**rocédés
EIPL	:	**E**nvironnements **I**ntégrés de **P**roduction de **L**ogiciel
EMF	:	**E**clipse **M**odeling **F**ramework
EMOF	:	**E**ssential **M**eta **O**bjet **F**acility
FOPL	:	**F**irst **O**rder **P**redicate **L**ogic
HFSP	:	**H**ierarchical and **F**unctional **S**oftware **P**rocess
IDE	:	**I**ntegrated **D**evelopment **E**nvironment
IDM	:	**I**ngénierie **D**irigée par les **M**odèles
ISPW	:	**I**nternational **S**oftware **P**rocess **W**orkshop
MADL	:	**M**eta **A**rchitectural **D**escription **L**anguage
MAPL	:	**M**odèle **A**rchitectural des **P**rocédés **L**ogiciels
MDE	:	**M**odel-**D**riven **E**ngineering
MOF	:	**M**eta **O**bjet **F**acility
MP	:	**M**odèle de **P**rocédé
MSL	:	**M**arvel **S**trategy **L**anguage
OMG	:	**O**bject **M**anagement **G**roup
OPC	:	**O**pen **P**rocess **C**omponents
OTAN	:	**O**rganisation du **T**raité de l'**A**tlantique **N**ord
PA	:	**P**rocess **A**rchitecture
PC	:	**P**rocess **C**omponent
PCM	:	**P**rocess **C**omponent **M**odel
PDL	:	**P**rocess **D**escription **L**anguage
PML	:	**P**rocess **M**odeling **L**anguage
PSEE	:	**P**rocess **S**oftware **E**ngineering **E**nvironment
SPEM	:	**S**oftware & systems **P**rocess **E**ngineering **M**etamodel
SPML	:	**S**oftware **P**rocess **M**odeling **L**anguage
TS	:	**T**echnical **S**pace
UML	:	**U**nified **M**odeling **L**anguage
W3C	:	**W**orld **W**ide **W**eb **C**onsortium
XML	:	e**X**tensibls **M**arkup **L**anuage
XPATH	:	**X**ml **PATH** language
XSLT	:	**E**xtensible **S**tylesheet **L**anguage **T**ransformation

INTRODUCTION GENERALE

INTRODUCTION GENERALE

I- DOMAINE DU MEMOIRE

Avec la révolution informatique, les systèmes logiciels sont devenus plus complexes (*systèmes de télécommunications, transport terrestre et aérien, production et transport d'énergie, etc*.) et évoluant sans cesse, leur développement exige plus de labeur et un coût plus élevé. En effet, les utilisateurs, de plus en plus exigeants, attendent de leurs logiciels un nombre de services de plus en plus important, et le respect de certaines contraintes comme un faible coût ou la rapidité de livraison et de maintenance. Ainsi, le produit logiciel est devenu complexe dont la réalisation doit s'intégrer dans une démarche méthodologique qui représente les meilleures pratiques dans le développement, sous forme de modèles de procédé logiciel. En conséquence, pour améliorer la qualité du produit développé, il est important de maitriser son procédé de développement dont la problématique s'apparente à celle du logiciel lui-même **[OST, 87].**

Ceci a conduit à accorder une attention renouvelée aux moyens et méthodes permettant d'étayer et d'améliorer les procédés de développement et de maintenance des produits logiciels.

Effectivement, les procédés logiciels sont aussi importants que les produits logiciels, car ils sont devenus plus complexes et donc plus difficile à comprendre, à contrôler et à mesurer que les produits eux-mêmes **[OST, 87].** En effet, ce domaine est distingué par le fait que ses objets sont peut-être plus grands, moins bien définis et plus mal compris que ceux dans des produits logiciels (*peut contenir des millions de lignes de code source, mais contenir aussi les volumes de documentation, d'énormes quantités d'information de conception, des spécifications (cahier des charges), test-cases et résultats de test, aussi bien que des manuels d'utilisateur, des manuel de maintenance, etc*.). D'où l'importance d'expliciter les politiques dirigeant les activités de développement et de maintenance.

Notre travail s'inscrit dans le cadre des efforts de la communauté des procédés logiciels qui visent à proposer de nouveaux formalismes, paradigmes et architectures pour surmonter les limitations des systèmes actuels et assister le procédé logiciel de manière plus efficace.

II- PROBLEMATIQUE ET MOTIVATION

Comme pour le développement du logiciel, le double défi auquel est confrontée l'ingénierie des procédés est celui de la qualité et de la productivité. Cependant, pour l'ingénierie des procédés logiciels, ces défis sont particulièrement difficiles à relever car les procédés de développement de logiciel sont des produits intrinsèquement complexes, pouvant être mis en œuvre de manière répartie, coopérative, itérative, par différents types d'agents, avec différentes contraintes de performance, et nécessitant diverses ressources matérielles ou humaines.

Comme pour le développement du logiciel, la réutilisation est considérée comme un moyen efficace pour surmonter ces défis. A cet effet, plusieurs travaux de recherche ont été menés (*citons :* **[CREb, 97], [BEL, 96]**), parmi eux se trouve ceux qui se sont orientés vers la génération de nouvelles approches, qui se basent généralement sur la décomposition d'un procédé en plusieurs sous-procédés. Ce découpage est fait suivant des critères divers et variés issus du domaine du génie logiciel. Ainsi, un procédé consiste en une très large collection de fragment de procédé réutilisable.

Bien que ces travaux aient plus ou moins répondu aux limites accordées aux environnements à procédé global et monolithique, ils n'ont pas pu atteindre de manière satisfaisante les objectifs attendus. En effet, ces procédés restent spécifiques à leurs environnements. Ils tentent de gérer, moyennant un système unique, les divers aspects du procédé.

L'acuité du problème, ainsi posé, est évidente lorsqu'on considère des compagnies possédant déjà leurs propres outils, méthodes et environnements. Au lieu d'ignorer ce support, il s'avère nécessaire de l'optimiser et de réintégrer les composants déjà existants au sein de la compagnie.

Cependant, dû à l'hétérogénéité des composants, le problème d'interopérabilité entre les différents composants se pose sur plusieurs niveaux :

- Au niveau sémantique, vu que les concepts différent d'un composant à un autre.
- Au niveau syntaxique, sachant que chaque composant utilise un formalisme différent.
- Au niveau d'exécution, étant donné que les composants peuvent être dans des environnements hétérogènes et partagent les même données.

Dans ce document, on s'est intéresse aux problématiques du niveau sémantique et syntaxique, alors que le niveau d'exécution ferra l'objet d'une de nos plus importantes perspectives.

III- APERÇU DE LA SOLUTION PROPOSEE

Afin de répondre à la problématique ainsi posée, nous avons proposé une contribution qui se base sur la modélisation des procédés sous forme d'une composition de composants réutilisables, indépendamment de leur implémentation interne, dans l'objectif de combler le trou séparant les lignes de codes des éléments gros-grains et la structure de leurs interactions.

Pour atteindre ce but, il s'avère nécessaire d'avoir un niveau d'abstraction élevé. Une des réponses possibles est la définition d'une architecture du système logiciel.

Effectivement, les architectures logicielles à base de composants **[MED, 00]**, semblent bien être une réponse à ces exigences. Sous certaines conditions, il est aujourd'hui parfaitement envisageable de mettre en place des composants logiciels réutilisables, qui permettront une maîtrise sans précédent dans le développement d'applications.

La description architecturale est un artefact central dans la conception logicielle. Elle permet d'abstraire un système comme un ensemble de composants, connectés par des connecteurs au sein d'une configuration architecturale. Fournissant ainsi, une description de haut niveau de la structure d'un système.

Plusieurs formalismes de description d'architectures (*ADL : Architecture Description Language*) ont été proposés **[MED, 00],** afin de décrire la structure et les propriétés des systèmes logiciels d'une manière qui est assez abstraite pour en réduire la complexité et les détails, mais qui demeure intelligible pour les concepteurs, tout en permettant l'analyse de la conception et le choix du mode de mise en œuvre.

En ce sens, l'utilisation d'un ADL est un pas vers la réutilisation de composants, et la projection d'un procédé conçu à partir d'une description en ADL sur divers environnements est par conséquent réalisable. Ce qui répond en partie à notre problématique.

Toutefois, afin de pouvoir rapprocher les travaux liés aux procédés logiciels de ceux liés aux architectures logicielles, nous jugeons important de disposer d'une vue plus abstraite et plus facilement maîtrisable que les langages (*manipulation du code*), une vue qui nous offre des moyens permettant de définir et d'exploiter les ADL et les PML **[ZUH, 01] [LIU, 95]** (*Process Modeling Language*), au-delà de leurs syntaxes.

Dans cette intention, notre approche repose sur l'utilisation des techniques de méta-modélisation ou plus précisément l'ingénierie dirigée par les modèles (IDM). Cette approche permet une plus grande souplesse dans la mise en correspondance des concepts des architectures et ceux des procédés par rapport à une simple approche de conversion code à code. On propose de modéliser un modèle de procédé logiciel à l'aide d'un ADL en s'appuyant sur les techniques de transformation de modèle.

En effet, les techniques de transformation de modèle, s'expriment directement entre les syntaxes abstraites des différents modèles, permettant ainsi, de se concentrer sur les concepts et d'aboutir, par la suite à une migration technologique cohérente et fiable.

En conséquence, on utilise ses principes pour représenter les modèles d'architectures logicielles et ceux des composants procédés, ainsi, que le scénario de transition entre ces deux approches.

VI- PLAN DU MEMOIRE

Outre cette introduction, le présent manuscrit est subdivisé en sept (07) chapitres et deux (02) annexes. Nous précisons ici l'objectif de chacun d'entre eux, un bref descriptif de leur contenu et les points qui nous paraissent essentiels.

- **Chapitre 1 :** Présente une vue d'ensemble des concepts qui découlent du domaine des procédés logiciels, nous nous somme intéressé particulièrement à la modélisation des procédés, ses objectifs et ses méthodes illustrés par des exemples d'environnements existants.
- **Chapitre 2 :** Est consacré à la génération des environnements de production de logiciels à base de composants procédés. Il montre l'apport de la technique en énumérant quelques environnements déjà opérationnels.
- **Chapitre 3 :** Présente les concepts de base liés aux architectures logicielles, ainsi qu'un état de l'art des travaux sur les langages de description d'architecture ADL.
- **Chapitre 4 :** Introduit les notions de base de l'IDM et les techniques de transformations de modèles.
- **Chapitre 5 :** Détaille notre approche. Notre méthodologie y est décrite et les différentes étapes y sont présentées.
- **Chapitre 6 :** Concrétise notre démarche, en appliquant les différentes étapes proposées à un ADL et des PML spécifiques.
- **Chapitre 7:** Illustre les différentes étapes, de la démarche présentée dans les deux chapitres précédents, à travers un exemple concret.
- **Conclusion et perspectives :** Finalement nous concluons ce rapport en proposant une synthèse de notre étude ainsi qu'un ensemble de perspectives liées à la continuité du travail.
- **Annexes :** Deux (02) annexes termineront ce document, ils arborent essentiellement : une présentation de l'environnement GLOBE1.0 et la représentation XML de nos modèles et méta-modèles.

PARTIE I

ETAT DE L'ART

CHAPITRE 1

PROCEDES LOGICIELS

I- INTRODUCTION

Un procédé logiciel (*Software Process*) est souvent défini comme étant un ensemble intégré d'activités requises pour transformer les besoins de l'utilisateur en un logiciel fonctionnel y compris les activités nécessaires à la maintenance de ce logiciel**[AMI,99]**.

En effet, afin d'augmenter la productivité et d'accroître la qualité des produits, il est indispensable de suivre et de maitriser un procédé de fabrication de logiciel. D'où l'importance de l'ingénierie des procédés (*Software Process Engineering*), une discipline du génie logiciel dont l'objectif est de supporter les procédés logiciels en fournissant un moyen de modéliser, d'analyser, d'améliorer, de mesurer et d'automatiser les activités de développement.

De nombreux travaux se sont intéressés à ce domaine et les aspects qui lui sont liés. C'est pourquoi ce chapitre présente une vue d'ensemble des concepts qui découlent du domaine des procédés de fabrications de logiciels et de leur modélisation illustrée par des exemples d'environnements existants.

II - GENERATIONS DES SUPPORTS AUX PROCEDES LOGICIELS

Depuis l'apparition des procédés dans le monde de l'informatique, de grands efforts se sont consacrés à la formalisation des tâches de développements, afin d'offrir les moyens pour gérer la complexité des procédés et contribuer à l'amélioration de la qualité des produits. On distingue différentes catégories du support aux procédés logiciels, notamment : *Les modèles de cycle de vie, les environnements intégrés et les environnements centrés procédés logiciels*. La section suivante discute chacun des trois catégories.

1 - MODELES DE CYCLE DE VIE

Le cycle de vie d'un logiciel (*Software Life Cycle*) est la période de temps s'étalant de la proposition ou la décision de développer un logiciel, à sa mise hors service (*figure 1.1*) **[STR, 00].**

Un modèle de cycle de vie décrit les étapes globales à suivre pour développer ou maintenir un produit logiciel. Parmi les modèles les plus connus, nous citons le modèle en cascade, le modèle en V, le modèle en spirale…

Les modèles de cycle de vie ont permis de mieux comprendre le processus logiciel, en identifiant les étapes et l'ordre global d'exécution. Cependant, ces modèles sont trop généraux et incomplets pour faire ressortir des informations importantes permettant de contrôler et d'automatiser ce processus.

- Les conditions d'enchaînement des activités et la notion de temps ;
- Les rôles joués par les différents utilisateurs et les interactions entre eux ;
- Les outils utilisés pour supporter les opérations ;

Le plus souvent, ces informations sont décrites de façon informelle dans des documents peu ou mal exploités.

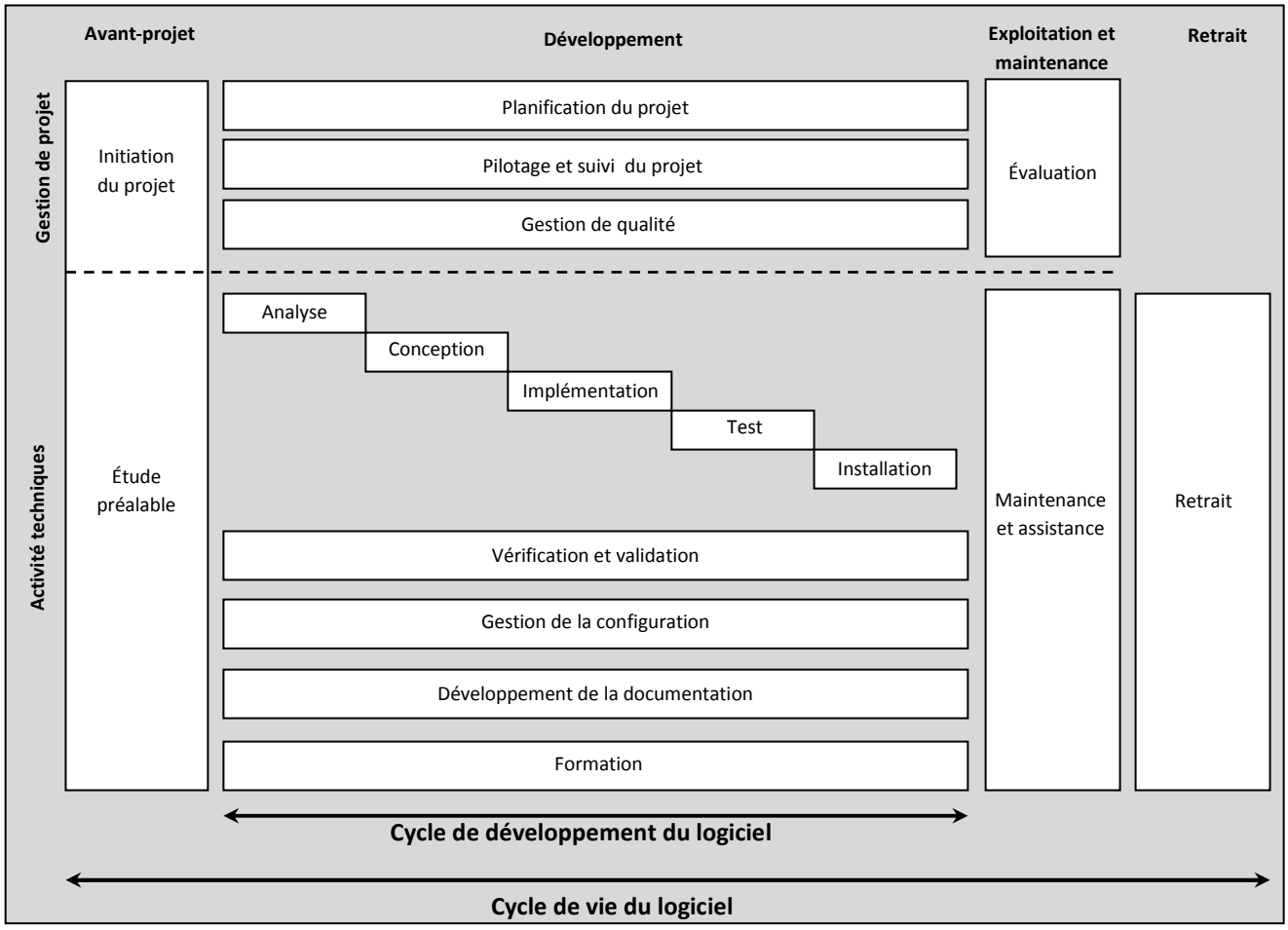


Figure 1.1 : Cycle de vie du logiciel [STR, 00].

2 - ENVIRONNEMENTS INTEGRES

Suite aux recherches concernant la conception et l'amélioration des modèles de cycle de vie, de multiples environnements intégrés regroupant des outils nécessaires au développement du logiciel ont été conçus.

EIPL (*Environnements Intégrés de Production de Logiciel*) ou **AGL** (*Atelier de Génie Logiciel*) ou encore **ateliers CASE** (*Computer Aided Software Engineering*) sont des ensembles d'outils permettant d'assister la production de logiciel **[MOS, 02] [LON, 92].**

Malgré le succès relatif de cette génération d'environnements, ces outils présentent une forte limitation due principalement à :

- Leur focalisation sur le produit en dépit des activités ;
- Leur aspect individuel, qui les rend difficilement adaptable et intégrable dans l'ensemble du procédé.

3- ENVIRONNEMENTS CENTRES PROCEDES LOGICIELS

Les approches présentés précédemment révèlent une limitation majeure : Soit ils proposent une méthodologie rigide (*modèles de cycle de vie*), soit ils intègrent des outils, des technologies logicielles sans méthodologie précise.

Pour remédier à ces limitations, des travaux ont été orientés vers le développement d'Environnements Centrés Procédé **ECP** (*Process Software Engineering Environment*), qui se définissent comme un système dans le quel le logiciel est fabriqué selon une description générique du procédé **[AMI, 99].**

L'objectif majeur des ECP est de fournir aux utilisateurs à la fois certaines des technologies de production du logiciel tout en intégrant une partie de la gestion de cette dernière dans un environnement intuitif.

Cette dernière catégorie de support de procédé logiciel s'énonce comme la plus prometteuse, en effet, un nombre important de projets industriels et de recherche se sont focalisés sur les ECP : Marvel **[FEI, 88]**, OZ **[SHA, 94]**, Arcadia **[TAY, 88]**, SPADE **[LAV, 94]**.

III– MODELISATION DES PROCEDES LOGICIELS

Pour pouvoir analyser, améliorer, mesurer et automatiser les procédés logiciels, il faut d'abord les définir explicitement. C'est l'objectif de l'axe de recherche principal de l'ingénierie des procédés, la modélisation de procédés logiciels (*Software Process Modeling*). En effet, modéliser un système avant sa réalisation permet de mieux comprendre son fonctionnement pour la bonne maitrise de sa complexité et d'en assurer sa cohérence.

La modélisation d'un procédé logiciel est l'élaboration d'une description de ce procédé en utilisant un (*ou plusieurs*) modèle*(s)* de procédé (*Process Model*). Une vue d'ensemble de la modélisation de procédé, est donnée ci-après.

Définition 1

> ***Procédé logiciel (Software process)*** *est en général défini comme un ensemble d'activités aussi bien techniques qu'administratives pour développer et pour maintenir un produit logiciel* **[HAN, 07].**

Définition 2

> ***Modèle de procédé (Process model)*** *est une description générique du procédé, spécifiant des types d'activités, des types de produits, des prescriptions et contraintes méthodologiques* **[AMI, 99].**

Définition 3

> ***Instance d'un procédé (Process instance)*** *est une adaptation d'un modèle à un projet spécifique en vue de gérer ses données et ses différentes étapes de réalisation* **[MOS, 02].**

1- OBJECTIF DE LA MODELISATION

L'objectif principal de la modélisation des procédés est de concevoir des pratiques de développement pour pouvoir les étudier, les améliorer et les utiliser de manière répétable, gérable et éventuellement automatique.

D'un point de vue général, les avantages de la modélisation peuvent être résumés en trois points essentiels :

- **Faciliter la compréhension des procédés** : Un modèle concrétise les connaissances du procédé et les rends plus compréhensive et communicable.
- **Faciliter la gestion et l'exécution du procédé** : En se basant sur une définition claire, un procédé peut bénéficier d'outils d'assistance et de contrôle, ainsi qu'une exécution éventuellement automatique.
- **Faciliter l'analyse et l'amélioration du procédé** : L'existence d'un modèle de procédé permet l'analyse, la simulation, ainsi que la réutilisation du procède, ce qui est important pour son amélioration.

2- ÉLEMENTS DE BASE DE LA MODELISATION DU PROCEDE

Divers éléments de procédé peuvent être représentés dans un modèle de procédé, ils se décomposent généralement, en éléments élémentaires et éléments secondaires. Dans ce qui suit, nous citons les plus répondus et les plus importants:

- **Activité (Activity):** Une opération atomique ou composite qui contribue à la réalisation du produit final.
- **Produit (Product):** Un artéfact (*code source, plan de test, …*) utilisé ou résultant d'une activité**.**
- **Rôle (Role)** : Concept abstrait, qui décrit les droits et les responsabilités d'un agent (*ou plusieurs agents*) pour réaliser certaines activités.
- **Agent (Agent)** : Un concept nécessaire pour la gestion du procédé, il représente toute personne qui participe au développement.
  Un agent peut avoir un ou plusieurs rôles en même temps, et peut également être membre de plusieurs groupes.
- **Ressource (Resource)** : Un élément (*matériel ou logiciel*) qui assiste et facilite l'exécution des activités, par exemple : les éditeurs, compilateur, …

3- FORMALISME DE MODELISATION

Afin de pallier la complexité de la description d'un procédé logiciel, plusieurs projets ont été lancés pour mieux étudier les aspects liés à la modélisation des procédés logiciels. De nombreuses approches tentent de supporter les processus logiciels aussi automatiquement que possible, le processus logiciel doit être décrit rigoureusement dans un formalisme qui permet de l'imposer, de le mesurer, de le simuler, de le réutiliser et de contrôler son évolution.

Définition 1

Un formalisme pour la modélisation de procédé logiciel est un langage textuel ou graphique qui permet de décrire des modèles de procédé **[KAB, 09].**

Langage de modélisation de procédé, PML (Process Modeling Language), est un formalisme de modélisation développé ou adapté pour décrire les procédés. Il fournit les concepts dédiés aux procédés, une syntaxe et un système de notation pour représenter les modèles de procédé en utilisant ces concepts.

Un PML peut être formel, semi-formel ou informel : Un PML formel est celui qui est prévue avec une syntaxe et une sémantique formelle. Tandis que, les langages semi-formels ont habituellement une notation graphique avec une syntaxe formelle, mais pas une sémantique formelle (*i.e. qu'ils ne sont pas exécutables*). Enfin, les PML informels sont ceux qui ne sont pas définis explicitement et formellement (*le langage naturel, peut être utilisé comme un PML informel*).

3.1- PROPRIETES D'UN PML

Plusieurs études ont discuté les propriétés attendues d'un PML (**[CUR, 90] [BEN, 07],…**), afin qu'il soit simple, uniforme et adapté pour la description du procédé. Certaines de ces propriétés sont :

- **Formalisation** : Cette propriété décrit le niveau de description de la syntaxe et de la sémantique d'un PML.
- **Expressivité** : C'est la capacité à représenter les procédés selon différentes perspectives.
- **Compréhensibilité** : Se mesure par la qualité de la notation et le degré de standardisation du langage.
- **Abstraction et Modularité** : Un PML supportant l'abstraction et la modularité facilite la réutilisation de procédé.
- **Exécutabilité** : Pour être exécutable, le langage doit offrir une représentation de modèle avec une sémantique opérationnelle.
- **Évolutivité** : Capacité de supporter l'évolution du modèle de procédé.

✍ *Il est a noté, cependant, qu'il est difficile de trouver un PML qui satisfait toutes ces propriétés, mais un bon compromis doit être trouvé.*

4- APPROCHES DE MODELISATION

Il existe plusieurs approches pour les formalismes de modélisation de procédés. Beaucoup d'entre eux, sont basées sur les approches existantes telles que les langages de programmation ou les réseaux de Pétri. Dont les plus répondus parmi eux, sont :

4.1- APPROCHE DECLARATIVE

L'approche déclarative utilise des déclarations logiques pour décrire le processus logiciel. Ceux-ci sont décrits en termes de résultats attendus par l'utilisateur sans détailler la manière dont ces résultats sont obtenus, de façons algorithmique.

Dans cette approche, les activités qui composent un modèle de processus logiciel sont décrites par des règles de production (*Pré-condition/Action/Post-condition*). Selon chaque type d'utilisation, une activité est enveloppée par une règle. Les pré-conditions de cette règle fournissent les conditions de déclenchement, tendis que les post-conditions fournissent les effets provoqués par son exécution.

Exemple :

Pré-condition	*Si le module X a déjà été compilé, tester le module X.*
Action	*Tester le module X*
Post-condition	*Si les testes se sont déroulés convenablement, alors modifier l'état du module X pour prendre en compte cette nouvelle information.*

Marvel **[FEI, 88]** et ALF **[CAN, 94]** sont deux systèmes qui se basent sur les règles de production.

L'exemple ci-dessous (*Listing 1.1*) montre la description d'une activité de compilation (*Compile*) dans Marvel (*Dans le langage MSL - Marvel Strategy Language*) :

```
1   Compile [?f.Cfile]:
2   ¤ Pré-conditions
3   ¤ Vérifié si le fichier C a été compiler après la dernière utilisation
4   (?f.compile_status =Not Compiled)
5
6   ¤ Active l'outil chargé de la compilation
7   {COMPILER Compile ? f.contents ?f.objet_code ? f.error_msg "-g"}
8
9   ¤Post-conditions
10   {or ( ? f.compile_status = Compiled )
11  (?f.compiled_status = Error);
```

Listing1.1: Approche déclarative – Marvel -

4.2- APPROCHE FONCTIONNELLE

L'approche fonctionnelle définit le procédé logiciel à travers un ensemble de fonctions mathématiques. Chaque fonction est décrite en termes de relations entre les données d'entrée et les données de sortie. Ces données sont des objets de génie logiciel (*fichier, programmes, etc.*). Par exemple, l'activité de compilation peut être modélisée par cette approche comme étant une fonction qui reçoit en entrée le code source et produit en sortie le code binaire correspondant. Parmi les systèmes utilisant cette approche, nous citons HFSP (*Hierarchical and Functional Software Process*) **[KAT, 89],** et PDL (*Process Description Language*) **[INO, 89].**

Par exemple (*Listing 1.2*), dans le HFSP, un procédé logiciel est décrit comme un ensemble de fonction de la forme :

```
1   A( x1, x2, ….,xn / y1,y2, ….., yn ) => A1,A2,……,Ak  Where  E ;
```

Listing 1.2 : Approche fonctionnelle – HFSP -

Avec : - A est une fonction qui correspond à une activité,
- $x_1, \ldots, x_n$ et $y_1, \ldots, y_n$ sont appelés les attributs de A, ils définissent ses entrées/sorties,
- $A_1, \ldots\ldots, A_k$ sont les sous activités qui composent A,
- E est une expression qui décrit comment se construit chacun des x_i et chacun des y_i.

4.3- L'APPROCHE PROCEDURALE

L'idée fondamentale est la spécification d'un modèle de processus logiciel sous la forme d'un programme. Ce programme décrit de manière détaillée comment le processus logiciel doit être réalisé. Par l'interprétation de ce langage le processus logiciel pourra être automatiquement exécuté, permettant ainsi de fournir une assistance aux différents usagers durant le déroulement de leurs activités.

Parmi les systèmes représentatifs de cette approche, on cite : Arcadia **[SUT,95]** et PWi **[BRU, 94]**, basés sur le langage Ada étendu.

Le listing 1.3 donne un exemple d'un programme Arcadia qui décrit la dérivation d'un module source vers un module objet et le comptage de mots dans un texte. La compilation est faite par l'opérateur ''compile'' décrit dans la relation ''Source_to_Object'' (*ligne 3*). Le comptage de mots est réalisé par l'opérateur ''WC'' dans la relation ''Word_Count'' (*ligne 18)* :

```
1   with compile; with code_defs; use
2   code_defs;
3   Relation Source_to_Object is
4   type src_to_obj_tuple is tuple
5   src : in source_code;
6   % paramétre d'entree
7   obj : out object_code;
8   % paramétre de sortie
9   end tuple;
10  entries
11  entry insert(src: source_code);
12  dependencies
13  determine obj by compile(src, obj);
14  end Source_to_Object
```

```
15  with WC;
16  % outil qui permet de compter les mots
17  with text_def; use text_def;
18  Relation Word_Count is
19  type wc_tuple is tuple
20  text: in text_type;
21  lines, words, characters: out integer;
22  end tuple;
23  entries
24  entry insert(text: in text_name);
25  dependencies
26  determine lines, words, characters
27  by wc(text, lines, words, characters);
28  end Word_Count
```

Listing 1.3 : Approche procédurale – Arcadia -

4.4- APPROCHE BASEE SUR LES GRAPHES

Cette approche décrit le procédé logiciel à travers un graphe qui se compose de nœuds et d'arcs.

Les réseaux de pétri (**Rdp**) **[DIA, 01]** sont souvent utilisés dans cette approche. Les concepts de base de ce dernier sont les transitions, les places et les jetons. Les transitions sont utilisées pour représenter les activités à conduire. Les places décrivent les pré et post conditions associées aux activités. La présence d'un jeton dans une place signifie que la condition est vérifiée, ils décrivent également les produits et les ressources manipulées par les activités. Parmi les systèmes basés sur les réseaux de pétri on peut citer SPADE **[LAV, 94]**.

La figure 1.2 illustre un exemple de modélisation dans SPADE, d'une activité de test, qui exige la présence d'une unité exécutable et des scénarios de test (*test cases*) pour commencer l'évaluation. Par la suite, tous les tests cases sont exécutés un par un, et leurs résultats sont ajoutés aux résultats de test obtenus précédemment (*Cumulative Test Results*). L'activité se termine une fois que tous les tests cases sont exécutés.

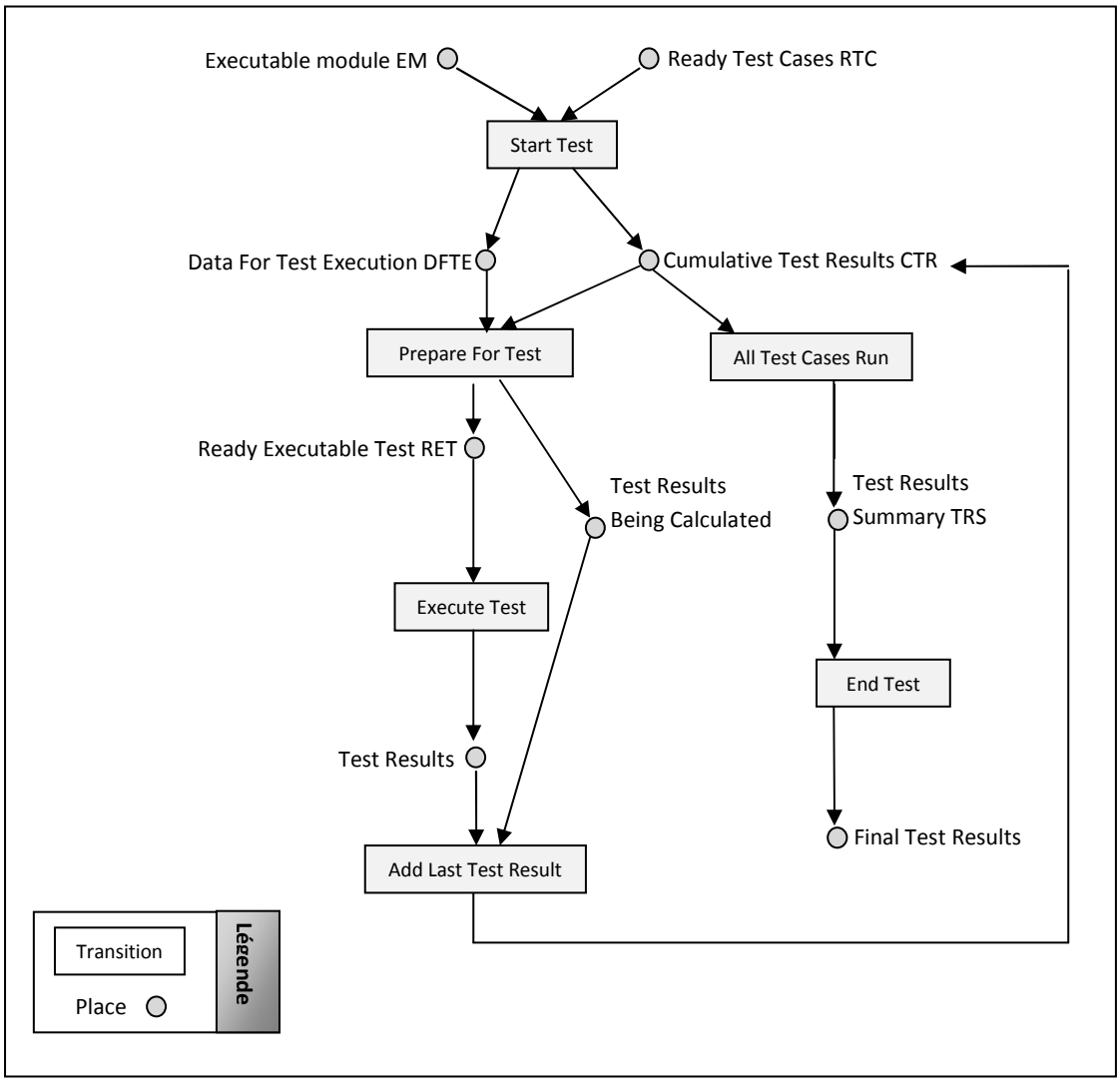


Figure 1.2 : Approche basée sur les réseaux de Petri - SPADE -

4.5- APPROCHE MULTI-PARADIGME

La modélisation multi-paradigme essaie de combiner plusieurs paradigmes, par exemple, le système JIL **[OST, 97]** combine l'utilisation du paradigme procédural et du paradigme déclaratif. Le premier est utilisé pour décrire l'exécution des activités du procédé, tandis que le deuxième paradigme, est utilisé pour décrire la réaction des activités à des événements qui se produisent durant l'exécution du procédé (*changement d'état des produits et des activités, les exceptions, etc.*).

La figure 1.3 montre l'exemple d'une activité de modélisation qui identifie les classes et les objets d'une application.

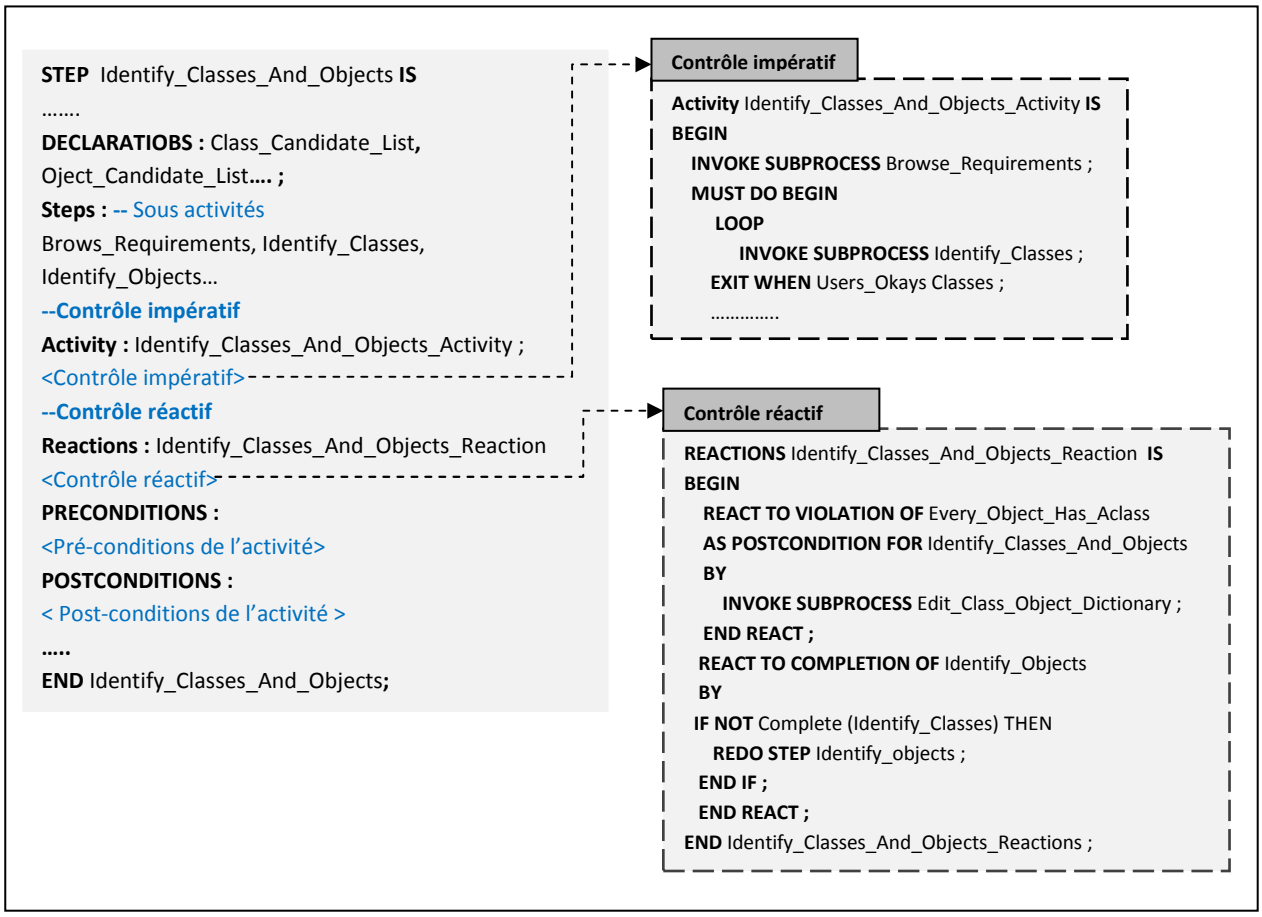


Figure 1.3- Approche multi-paradigme - JIL -

4.6- APPROCHE BASEE SUR UML

Cette approche utilise des diagrammes UML (*Unified Modeling Language*) pour représenter les concepts du procédé et renforce la sémantique de ces diagrammes avec un langage formel pour rendre les modèles de procédé à la fois compréhensifs et exécutables. Cette approche s'appuie sur les avantages d'UML (*notamment la richesse de ses concepts, de ses notations et de ses diagrammes*), sans négliger le fait que c'est un standard reconnu avec une position dominante dans les communautés académique et industrielle.

Plusieurs approches ont été développées avec cette perspective. Nous citons principalement : le standard de l'OMG SPEM **[SPE, O5] [SPE, O8],** UML4SPM **[BEN, 05] [BEN, 07] [BENa, 07],** et Executable PML from UML **[DIN, 02].**

La figure 1.4, extraite de **[BENa, 07],** décrit une partie d'un modèle de procédé en UML4SPM, qui modélise une abstraction du procédé de modification d'un plan de test.

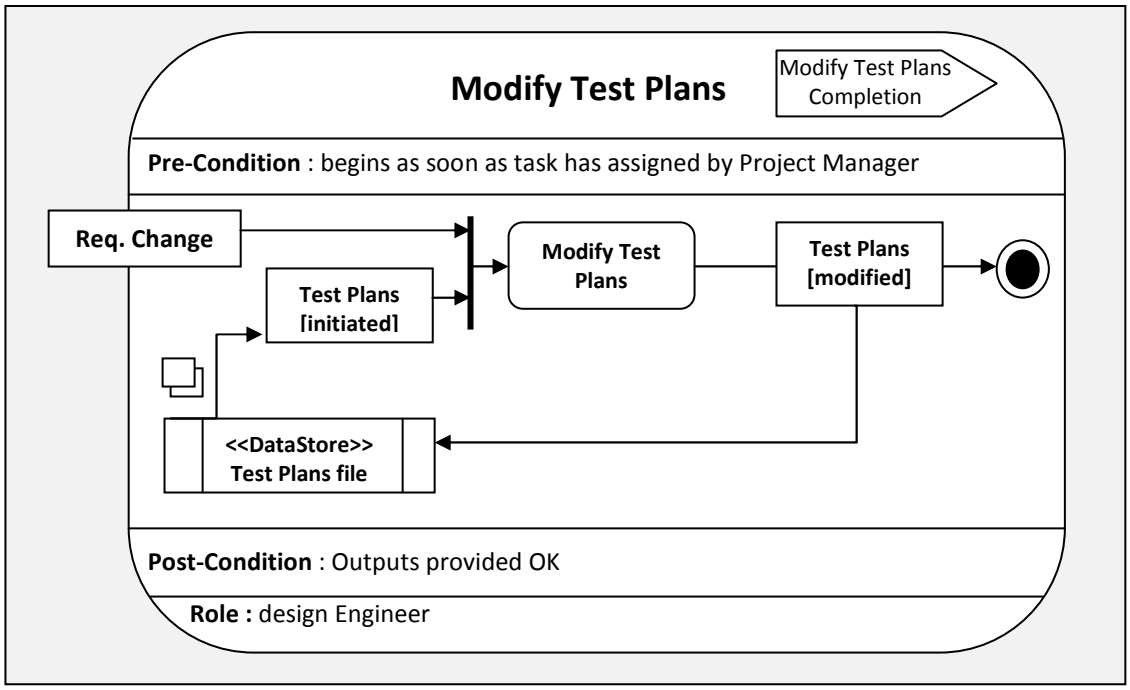


Figure 1.4- Approche basée sur UML - UML4SPM- [BENa, 07].

4.7- SYNTHESE

L'objectif principales des PML est de maintenir une connaissance répétitive des procédés, et de s'en servir à des fins de compréhension, d'analyse et d'exécution.

Cependant, les PML proposés par la littérature, n'ont pas réussi à s'imposer **[AMI, 99] [HAN, 07] [COM, 08] [KAB, 09]**. De plus, rares sont les langages pouvant concilier à la fois la compréhensibilité, la flexibilité et l'exécutabilité (*les plus importants critères pour la définition d'un PML*).

Ces PML sont le plus souvent basés sur des langages de programmation, d'autres sont plus similaires à des formalismes tels que les Réseaux de Pétri ou bien sont basés sur des règles d'inférences. Ces langages imposent de représenter un modèle de procédé logiciel sous forme d'un programme informatique. L'approche basée sur UML, quant-à-elle, profite des notations standardisées, mais elle a besoin d'être renforcée par une sémantique opérationnelle pour devenir formelle.

IV- REUTILISATION DES PROCEDES LOGICIELS

Le concept de réutilisation a été évoqué par M.MciLroy en 1968 dans une conférence de l'OTAN (*Organisation du Traité de l'Atlantique Nord*) consacré à la crise de logiciel ou il a proposé une solution basée sur la réutilisation de bibliothèques de composant (*article réédité en 1976* **[MCL, 76]).** Depuis, cette idée à évoluée et la réutilisation est devenue une discipline et un thème de recherche à part entière.

L'approche de réutilisation de procédé est parmi celle contribuant à ces objectifs. Elle se définit comme une approche ou la construction d'un procédé s'appuie sur des connaissances de procédé existantes, éprouvées et réutilisables.

Cette section, présente les principales techniques actuelles en matière de réutilisation.

1- PROBLEMATIQUES DE REUTILISATION DE PROCEDE

Dans le domaine de la réutilisation de procédé, deux types de problématique de recherche sont considérées : l'ingénierie pour la réutilisation (*Engineering for Reuse*), et l'ingénierie par la réutilisation (*Engineering by Reuse*) **[HAC, 06] [FRO, 05] [SER, 03] [HAN, 07]**.

1.1- INGENIERIE POUR LA REUTILISATION DE PROCEDE

L'objectif de l'ingénierie pour la réutilisation des procédés est de développer des méthodes supportant la production des connaissances de procédé réutilisables. Ses axes de recherche concernent principalement :

- L'identification des ressources candidates à la réutilisation,
- Leur représentation sous forme d'artefacts réutilisables,
- Leur organisation au sein de systèmes de réutilisation.

1.2- INGENIERIE PAR LA REUTILISATION DE PROCEDE

L'objectif de l'ingénierie par la réutilisation de procédés est de développer des méthodes pour réutiliser les entités réutilisables dans le but de construire de nouveaux procédés ou d'enrichir des procédés existants.

Trois facettes de la problématique de l'ingénierie par la réutilisation de procédés, sont généralement distinguées:

- Les opérateurs de manipulation d'entités réutilisables,
- Le guidage méthodologique,
- Les outils support qui peuvent supporter la modélisation de procédés par la réutilisation.

✍ *Notre contribution s'inscrit dans le cadre des travaux visant à offrir des moyens de conception d'entités réutilisables, et les méthodes de leur intégration au sein d'un procédé globale.*

2- FORMES DE REUTILISATION DANS L'INGENIERIE DES PROCEDES

Plusieurs techniques ont été proposées, toutes dans l'objectif d'offrir aux analyseurs, concepteurs et développeurs une meilleure infrastructure de réutilisation.

Quelques exemples, parmi les plus répondus actuellement, sont présentés dans cette section :

- **Composant de procédé** : un composant de procédé encapsule une connaissance de développement, Il fournit ainsi, un fragment de procédé réutilisable. Plusieurs travaux sur les composants de procédé ont été menés, comme : **[CREb, 97], [BEL, 96].**

- **Patron de procédé :** un patron de procédé capitalise la connaissance du procédé qui décrit explicitement la connaissance capturée ainsi que la manière de l'appliquer. Les patrons de procédé sont étudiés dans des travaux comme **[AMB, 98] [SEL, 01] [HAN, 07], [CREb, 97], [BEL, 96].**

V- CONCLUSION

Ce chapitre a traité plus particulièrement les formalismes de modélisation pour faire face à la complexité des procédés. De nombreuses recherches poursuivent leurs efforts pour promouvoir l'expressivité, la compréhensibilité, la flexibilité et l'exécutabilité comme critères essentiels aux langages de modélisation de procédés.

Ces critères répondent non seulement aux besoins des acteurs du procédé, mais aussi aux attentes du marché du logiciel, toujours plus exigeant en termes de fiabilité et de rapidité de production.

Ces besoins ont fait émerger des méthodes et des outils innovants, aujourd'hui largement adoptés et utilisés. Il est maintenant possible d'envisager une approche de développement basée sur la réutilisation de composants de procédés existants et éprouvés. Une telle approche doit permettre de réduire le temps de conception des procédés, d'en améliorer la qualité et d'en faciliter la maintenance.

En effet, l'approche « *composants de procédés* » est très prometteuse. Elle propose la description d'un procédé de développement en utilisant des composants réutilisables qui capitalisent l'expérience acquise lors des définitions précédentes des procédés.

Les parties réutilisables du procédé sont alors décrites dans des composants qui peuvent ensuite être réutilisés et instanciés pour faciliter la définition de nouveaux procédés. On obtient ainsi des solutions plus facilement adaptables aux changements et l'investissement intellectuel se trouve mieux préservé.

Cette approche répond en partie à la problématique de réutilisation des procédés traitée dans ce mémoire, c'est pourquoi on la détaille dans le prochain chapitre.

CHAPITRE 2

COMPOSANTS DE PROCEDE

I- INTRODUCTION

La réutilisation est un domaine important en génie logiciel, qui se définit comme une démarche ou la construction des systèmes s'appuie sur des connaissances éprouvées et réutilisables.

Plusieurs travaux se sont, orientés vers les technologies de réutilisation, afin de diminuer les efforts et les investissements requis pour le développement d'un nouveau projet.

L'ingénierie de conception des applications logicielles à base de composants CBSE (*Component-Based Software Engineering*) est parmi celles contribuant à ces objectifs. Cette technologie a donné naissance à différents axes de recherche, dont se trouvent, les travaux sur les composants de procédés, qui proposent la description de procédés de développement en utilisant des composants réutilisables. Ces composants, sont structurés par les éléments du procédé, qui peuvent être réutilisés.

Ce chapitre présente les concepts clés de l'approche composant logiciel, ensuite il détaille la notion de composant de procédé, et enfin, des environnements à base de composant de procédé sont présentés.

II- COMPOSANTS LOGICIELS

1- DEFINITION

Il existe actuellement plusieurs définitions du composant logiciel qui se distingue par l'accent mis sur des points différents. Elles ont tendance à converger vers une solution unique mais ce n'est pas encore aussi bien institué que la notion d'objet. En effet, Jusqu'à présent il n'existe toujours pas de consensus sur une définition commune du terme composant logiciel. Nous citons ici la définition de Clement Szyperski **[SZY, 98]**, la plus utilisée dans la littérature, qui couvre les principales notions :

Définition 1

> *"A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties."* **[SZY, 98]**

Un composant est une unité de composition qui spécifie par contractualisation ses interfaces, et qui explicite uniquement ses dépendances de contexte. Un composant logiciel peut être déployé indépendamment et est sujet à une composition par de tierces entités.
Cette définition est intéressante, car elle aborde presque toutes les caractéristiques d'un composant logiciel, en effet, elle implique l'existence des notions de base, telles que : l'encapsulation, la composition, l'interface et le déploiement.

2- REPRESENTATION D'UN COMPOSANT

Nombreux modèle de représentation existe actuellement dans la littérature, il n'existe toujours pas de modèle de représentation standard des composants. Ceci est dû au fait que les composants peuvent être relatives à des technologies différentes. Ou, chaque technologie possède son propre modèle, et sa propre représentation, qui met l'accent sur certaines spécificités du modèle et écarte d'autres.

Un composant est généralement représenté par les éléments suivants **[OUS, O5] [DEA, 04] [BEL, 08]**:

- **L'implémentation :** contient la mise en œuvre des aspects fonctionnels et non fonctionnels. L'implémentation du composant peut être fournie en code compilable, en forme binaire etc.
- **Les instances** : s'exécutent dans le système, chaque instance est définie par une référence unique, à savoir un type de composant et une implémentation particulière de ce type.
- **Les interfaces** : les interfaces d'un composant sont des points de communication qui lui permettent d'interagir avec son environnement.
- **Les propriétés** : les propriétés d'un composant peuvent concerner aussi bien sa structure, son comportement ou ses fonctionnalités. Elles peuvent être paramétrées et configurées selon le contexte d'exécution. Il existe également des propriétés non fonctionnelles qui ne font pas partie des services applicatifs (*sécurité, réseau, etc.*).
- **Les interactions** : les interactions expriment les différents échanges entre les composants d'un système.

- **L'architecture** : l'architecture logicielle d'un système à base de composants est une spécification des composants d'un système et de leurs interactions. Elle permet de fournir un plan précis approprié pour prédire le comportement d'un système avant de le construire et pour guider son développement.

3- DIFFERENTES ABSTRACTIONS D'UN COMPOSANT LOGICIEL

L'abstraction d'un composant correspond à la visibilité de son implémentation. On trouve généralement trois (03) sortes d'abstractions **[DEA, 04] [LEG, 05]** :

- **Boîte noire (Blackbox) :** Le client ne connaît aucun détail au-delà des interfaces et de leurs spécifications. La réutilisation d'une boîte noire se réfère au concept de réutilisation de l'implémentation en reliant les interfaces entre elles.

- **Boîte blanche (Whitebox) :** L'implémentation d'une boîte blanche est entièrement disponible et peut donc être étudiée afin d'augmenter sa compréhension. La réutilisation d'une boîte blanche s'apparente à l'utilisation d'un fragment de logiciel à travers ses interfaces, tout en tenant compte de la compréhension acquise par la connaissance de l'implémentation. On peut trouver dans la littérature, le terme de **Boîte transparente (Glassbox)**. Quand la distinction est faite cela signifie que la boîte blanche permet la manipulation de l'implémentation alors que la boîte transparente permet simplement l'étude de l'implémentation.

- **Boîte grise (Graybox) :** Elle offre la possibilité d'exposer certaines parties du composant et donc de modifier ou de configurer certaines parties internes du composant dans un cadre bien précis.

La figure 2.1 résume ces trois approches :

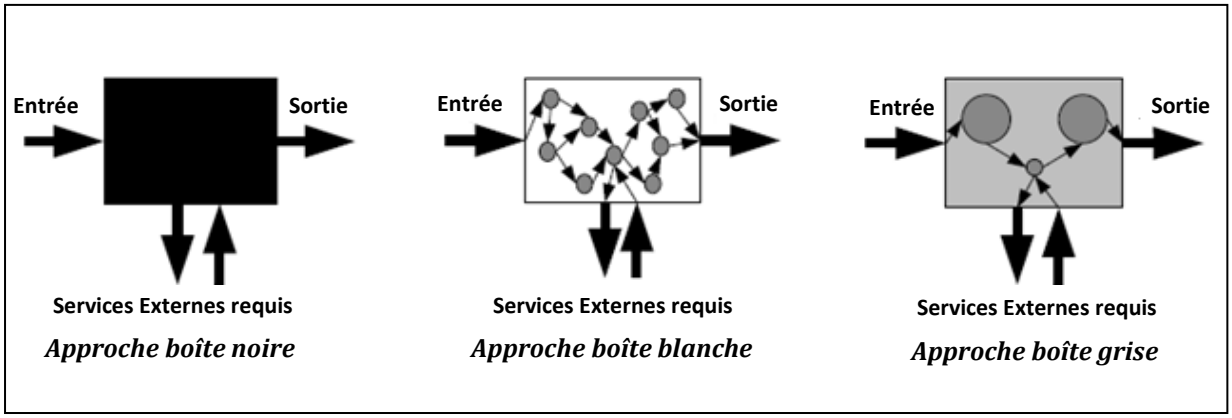


Figure 2.1 - Les différentes abstractions d'un composant [LEG, 05]

4 - COMPOSITION DES COMPOSANTS LOGICIELS

La composition est le mécanisme qui permet de lier un composant avec d'autres composants en satisfaisant les services requis d'un composant grâce aux services offerts d'un ou de

plusieurs autres composants. Elle permet de construire des applications complexes à partir de composants plus simples.
Deux types de composition sont distingués : la composition verticale et la composition horizontale.

4.1- COMPOSITION VERTICALE

La composition verticale, également appelée *composition hiérarchique*, permet de définir des composants de granularité supérieure par composition de composants de granularité inférieure.
Cet aspect consiste à voir l'implémentation des composants selon différents niveaux hiérarchiques. Ainsi, on distingue ces niveaux en définissant les composants *primitifs* et les *composants composites*.

- Un composant est dit *primitif* lorsqu'il implémente une fonctionnalité de granularité faible.
- Un composant est dit *composite* s'il est lui-même écrit à l'aide de composants primitifs et/ou composites.

4.2- COMPOSITION HORIZONTALE

La composition horizontale, également appelée *assemblages de composants*, permet l'interaction de ces derniers afin de déployer des applications.
La composition horizontale, est donc, capitale pour les composants logiciels dès lors que l'on veut fournir la possibilité de développer des applications à l'aide d'assemblages de composants. Elle fournit une vue compositionnelle des applications derrière laquelle se cache la notion de connexion.

La composition est assurée par différents moyens, citons par exemple :

- Les interactions simples entre composants à l'aide d'outils introduits par la programmation orientée objet : les interfaces (*fournies/requises*) et les événements (*source/puits*) qui consistent à faire coopérer simplement des composants par invocation de méthodes ou sur déclenchement d'événements.

- Introduction d'entité supplémentaire nommée *connecteur*. Les connecteurs sont des abstractions de communication de même niveau que l'abstraction composant, ce sont des éléments supplémentaires de l'architecture logicielle qui servent d'intermédiaires entre les composants. De façon générale, un connecteur définit l'ensemble des messages que les composants échangent, la façon dont ils se synchronisent et les contraintes qu'impose l'interaction. Cette entité encapsule donc un protocole d'interaction entre composants.

La figure 2.2, présente un exemple de compositions horizontale et verticale.

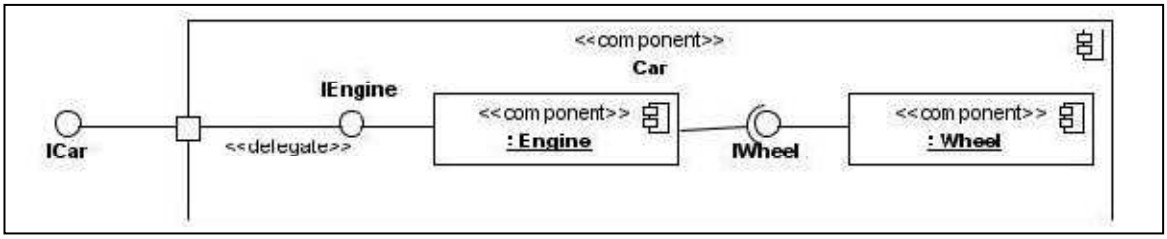


Figure 2.2 - Exemple de compositions horizontale et verticale

Le composant **Engine** est composé horizontalement avec le composant **Wheel** via l'interface **IWheel** ; le composant **Car** est un composite dont l'interface de service fourni **ICar** est réalisée par le composant **Engine**.

III - COMPOSANTS DE PROCEDE

Les approches de modélisation se concentrent souvent sur certains aspects de processus logiciels et les présentent dans une description prédéterminée, générale et monolithique. Ces approches échouent à fournir des moyens efficaces pour gérer des concepts comme l'évolution, l'intégration et la réutilisation. La modélisation des composants de procédé est une solution proposée afin de pailler ces manques.

1- DEFINITIONS

Définition 1

Process Component: *''A process component is an encapsulation of process information and behaviors at a given level of granularity.''* **[GARa, 97]**

Un composant de procédé est une encapsulation des informations du procédé et de ses comportements à un certain niveau de granularité.

Définition 2

Component-based Process Model *: ''A component-based process model is a collection of process components that interact in meaningful ways. A component-based process model is itself a process component. ''* **[GARa, 97]**

Un modèle de procédé à base de composants est une collection de composants de procédé qui interagissent de façon significative. Un modèle de procédés à base de composants est lui-même un composant de procédé.

2- OBJECTIFS VISES PAR LES COMPOSANTS DE PROCEDES

Les principaux objectifs d'un composant de procédé se résument comme suit **[BEL, 96]** :

- **Multi-paradigmes :** La plupart des modèles de processus existant ont été conçus pour prendre en charge un aspect particuliers (centré produits, centré activités ou centré rôles). L'objectif de cette approche est de couvrir au mieux toutes les activités d'un processus logiciel.

- **Réutilisation du processus :** La réutilisation est une activité stratégique dans la construction de processus logiciel. Une démarche de réutilisation devra donc, fournir des solutions afin d'exploiter les modèles existants et les réutiliser comme des composants de base pour des nouveaux modèles de processus.

- **Construction du processus :** Une approche de modélisation de processus basée composants, offre un paradigme puissant pour la réutilisation et la composition de processus logiciel. Déterminer des composants réutilisables qui compose un modèle de processus, et assurer leurs cohérences mutuelles est un but majeur de l'approche basé composant de procédé.

- **Contrôle du processus :** Le contrôle du procédé est une activité qui exige des moyens de spécification, de sélection et de présentation de l'état du processus avant et durant son exécution.

- **Évolution du processus :** Un procédé logiciel est amené à changer plusieurs fois durant son cycle de vie. Ainsi, un bon modèle de processus logiciel, doit être continuellement redéfini afin d'intégrer des changements permanents, par l'introduction de nouveau composant ou la mise à jour d'anciennes versions.

3– CARACTERISTIQUES SOUHAITÉES DANS UN COMPOSANT DE PROCEDE

Les caractéristiques des composants de procédé sont fortement inspirées de ceux des composants logiciels. Les caractéristiques présentées par TRAN Dan Thu **[THU, 01]**, sont repris dans cette section où il a adapté et complété le travail de synthèse de Sametinger **[SAM, 97],** par des points jugés nécessaires pour le contexte des composants de procédés.

- **La séparation entre spécification et implémentation** : c'est une caractéristique qui permet de développer séparément la partie implémentation et la partie spécification d'un composant.

- **L'auto-description** : chaque composant doit contenir des informations qui le décrivent. Ces informations sont nécessaires à la réutilisation du composant.

- **L'autonomie:** cette caractéristique permet de réutiliser un composant sans inclusion d'autres composants.

- **La possibilité d'évolution** : c'est l'évolution de chaque composant durant son cycle de vie.

- **La cohérence**: elle permet de décrire rigoureusement le processus de développement.

- **Les multiples vues de description**: elles ont pour but de passer progressivement d'une description informelle d'un composant de procédé à sa description formelle.

4- NIVEAUX DE COMPOSITION DES COMPOSANTS DE PROCEDES

Tous comme les composants logiciels, les composants de procédés sont organisés en deux niveaux : Les composants élémentaires et les composants complexes.

4.1- COMPOSANTS ELEMENTAIRES

Les composants élémentaires sont les entités de base pour décrire formellement et statiquement les procédés de développement. Ils correspondent aux constituants atomiques d'un procédé. Généralement, trois entités sont distinguées :

- **Activités :** Une activité décrit un travail à réaliser pour atteindre un certain objectif.

- **Produits :** ils sont utilisés pour décrire les entités manipulées. Chaque produit peut être considéré comme l'implémentation d'un type abstrait de donnée.

- **Rôle :** ils sont utilisés pour modéliser les compétences des développeurs participant au développement et caractériser les compétences requises pour réaliser une activité ou choisir une heuristique de développement.

4.2- COMPOSANTS COMPLEXES

Un composant complexe est une description d'une partie d'un procédé englobant un ensemble de composants élémentaires et les relations entre eux. Ils ont pour objectifs :

- De gérer et réutiliser plus efficacement les composants de procédés.
- D'aboutir à une cohésion forte de chaque composant, et à un couplage faible entre les composants.
- D'avoir si possible l'autonomie d'un composant.

IV- MODÈLE DE PROCÉDÉS À BASE DE COMPOSANTS

Cette section propose une brève description de quelques modèles de procédé à base de composants: RHODES **[CREa, 97]**, OPC **[GARb, 99],** PYNODE **[BEL, 96]** et SPEM2.0 **[SPE, 08].**

1- RHODES

L'environnement RHODES **[CREa,97] [CREb,97][THU, 01] [DAN, O5]** est un ECP qui a été développé à l'IRIT-ENSEEIHT. Il s'appuie sur une description formelle du procédé qu'il exécute pour contrôler le développement et assister les développeurs.

La figure 2.3 illustre l'architecture de l'environnement RHODES avec cinq constituants principaux : le méta-modèle de composant, la base de composants de procédés, le noyau d'exécution, les outils pour les concepteurs de procédés, et les instances de RHODES.

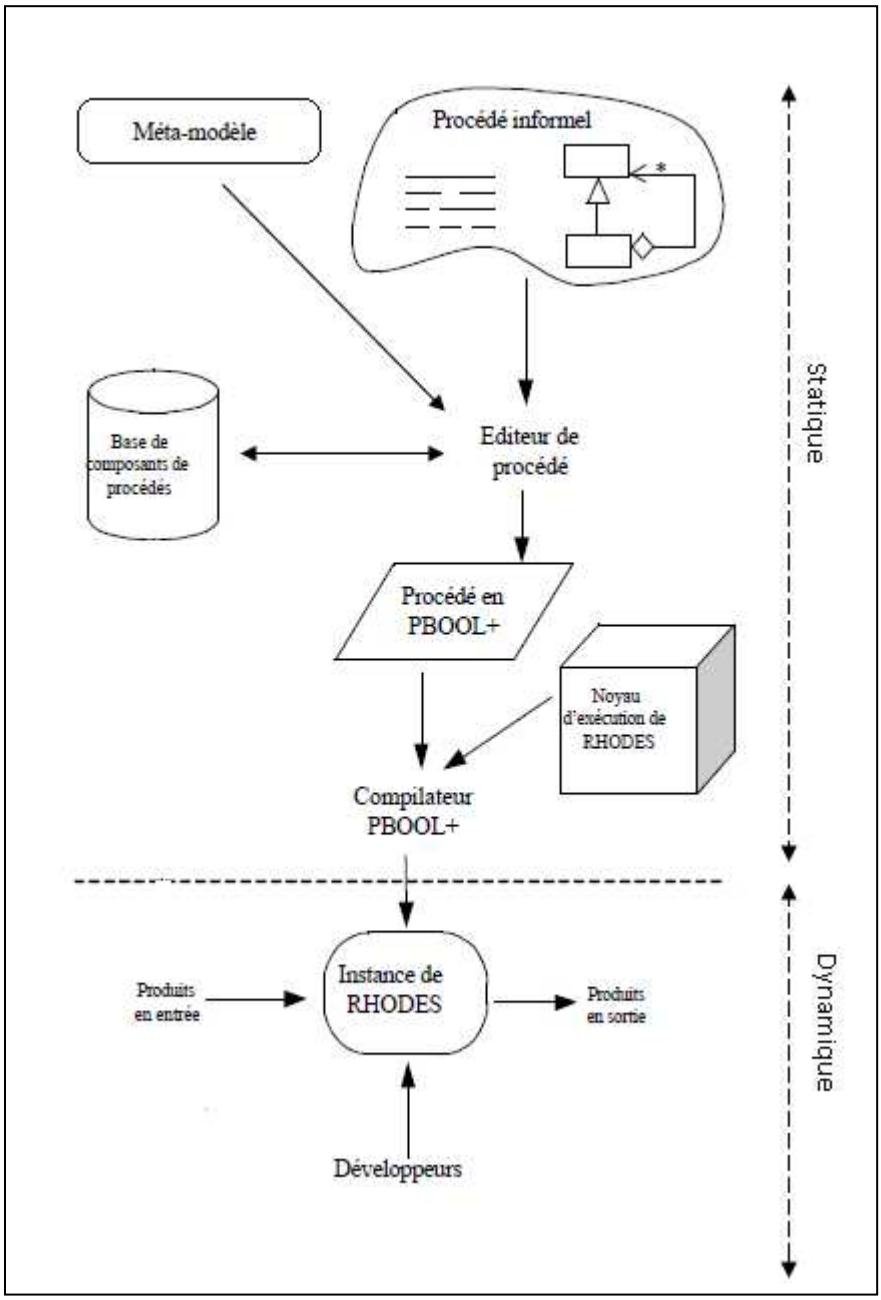


Figure 2.3 : Architecture de l'environnement RHODES [CREb, 97]

- **Le méta-modèle :** Le méta-procédé de RHODES permet de passer progressivement d'un procédé informel à un procédé décrit sous forme de composants PBOOL. Il prend en compte à la fois la gestion globale d'un procédé et la description de composants de procédés de granularité fine.

- **Le langage PBOOL+ :** Le langage PBOOL+ **[CREa,97] [DAN, O5]** est une extension du langage de procédés PBOOL auquel il ajoute la notion de composants complexes.
  Le langage PBOOL, permet de modéliser un procédé de développement. Il privilégie les activités et les heuristiques de développement mais décrit également les produits, les rôles et les stratégies dans un modèle complémentaire.

- **La base de composants de procédés :** La base de composant de procédés est utilisée pour le stockage et la réutilisation des composants de procédés

- **Le noyau d'exécution :** L'objectif du noyau d'exécution est d'exécuter des procédés, de contrôler rigoureusement le développement, d'adapter une assistance organisationnelle aux développeurs, et de permettre l'évolution dynamique de procédés.

- **Les outils :** Les concepteurs de procédés utilisent des outils (*navigateur, éditeur, compilateur*) pour décrire les composants de procédés, les stocker dans une base de composants, les assembler et les compiler en une instance de RHODES.

- **Les instances de RHODES :** Les développeurs utilisent des instances de RHODES pour développer des logiciels. Ils bénéficient ainsi des connaissances méthodologiques intégrées dans cette instance.
  Une instance de RHODES supporte un procédé particulier. Elle est obtenue par compilation d'un procédé PBOOL+.

2 - OPC (*Open Process Components*)

OPC **[GARb, 99] [GAL, 00]** propose un environnement pour la production de procédés à base de composants. Dans cette approche, les modèles de procédés sont construits comme un ensemble de composants qui interagissent à travers des interfaces bien définies.

Il propose aussi une infrastructure, pour la réutilisation de composants procédés hétérogènes, construit selon trois niveaux d'abstraction (*figure 2.4*).

- Le niveau méta–modèle *(Schema)* qui identifie les entités fondamentales et leurs relations.
- La Machine d'états–finis *(State)***,** qui représente le comportement des entités du procédé.
- Un framework orienté objet (*Implementation*) lié au formalisme de représentation du procédé et à son domaine d'application.

Le schéma d'OPC inclut des ''*meta-role*'', qui permettent d'identifier différents rôles pour chaque entité (*définition, observation….)* et qui agissent réciproquement avec les autres composants. Ainsi, chaque composant OPC, possède des vues distincte ''*meta-views*'' associer à un type de rôle.

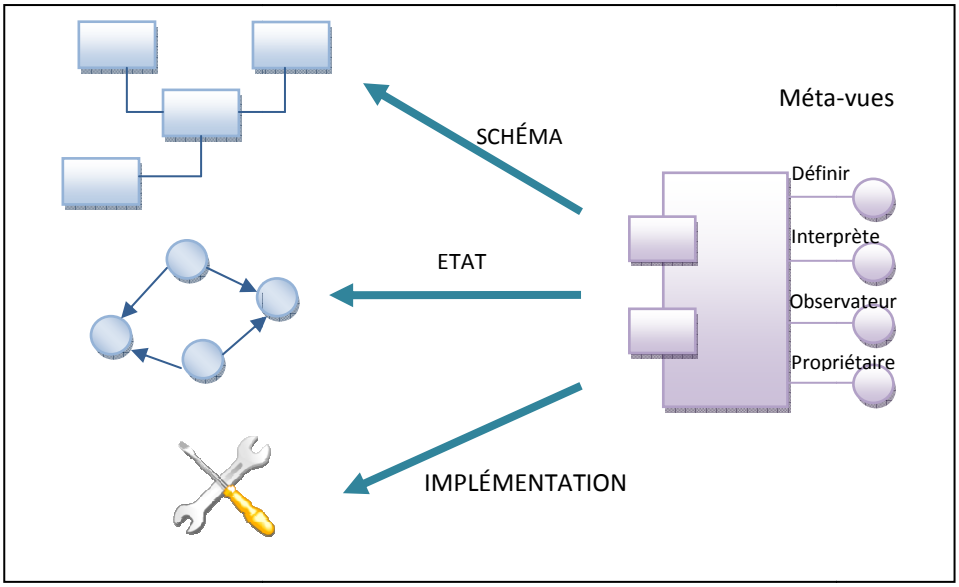


Figure 2.4 : Composant de procédé OPC [GARa, 97]

OPC supporte trois mécanismes d'interaction : *méthodes, messages et événements*.

- **Méthodes :** OPC supporte un ensemble standard de méthodes, dérivé de son méta-modèle et de son graphe de transition d'état.
- **Messages :** Un composant procédé peut envoyer un message afin de requérir un changement d'état.
- **Evénement :** Un composant de processus produit et reçoit des événements. Un composant produit un événement quand son état ou l'une de ses entités fondamentales changent.

3- PYNODE

La principale motivation de Pynode **[BEL, 96] [GAL, 00]**, est de développer un environnement interactif et intégré, supportant des procédés. L'idée consiste à éliminer les aspects monolithiques des formalismes de procédés, et ainsi fournir des possibilités de réutilisation et d'intégration de représentations de procédés hétérogènes.

Pynode utilise la notion de vue. Une vue permet la décomposition d'un processus selon une caractéristique bien définie (*rôle, activité ou produit*) .Deux sortes de vues sont distinguées (*figure 2.5*) :

- **Vues d'exécution** : Ces vues concernent les parties du modèle où toutes les actions modifiant l'état du processus sont décrites.
- **Vues d'observation** : Ces vues reposent sur des données rassemblées des vues d'exécution pour donner des informations synthétiques sur l'état des éléments du procédé.

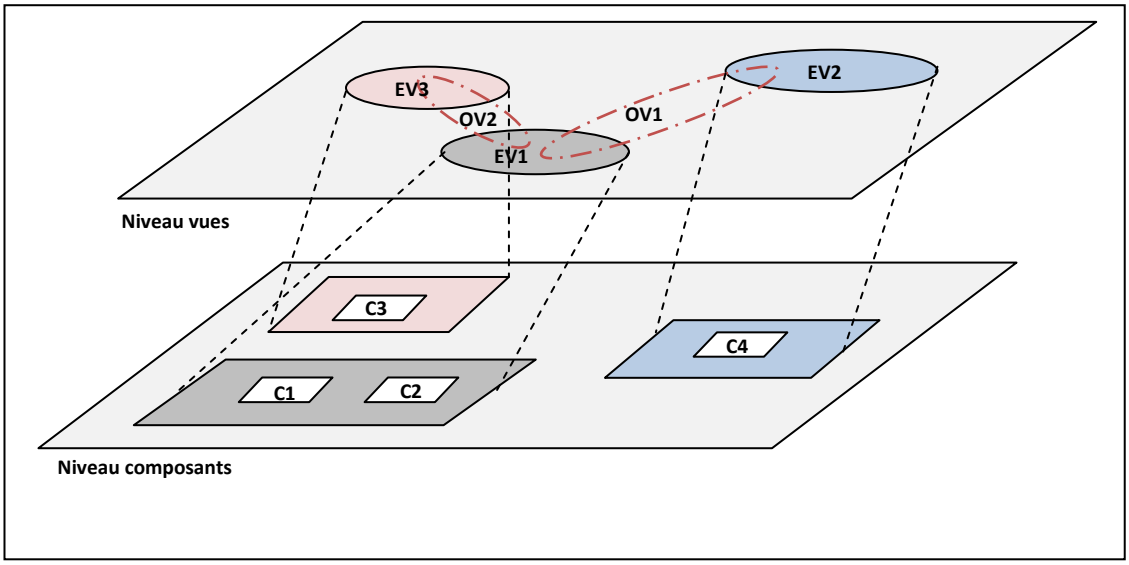


Figure 2.5 : les vues d'exécution et d'observation [BEL, 96]

L'exécution du procédé est supportée via les vues d'exécution. Une vue d'exécution définit une collection de composants et de relations de composition. La composition dynamique de composants via les vues d'exécution des composants définit l'exécution du procédé.

Chaque composant Pynode est caractérisé par un triplet (*rôle, produit, activité*), Les possibilités de composition sont représentées dans la figure 2.6. En construisant un modèle, il faut se décider quelle perspective est prioritaire et laquelle est secondaire. Et suivre ensuite le chemin correspondant dans le graphe (*figure 2.6*). La composition le long d'un axe spécifique peut être partielle.

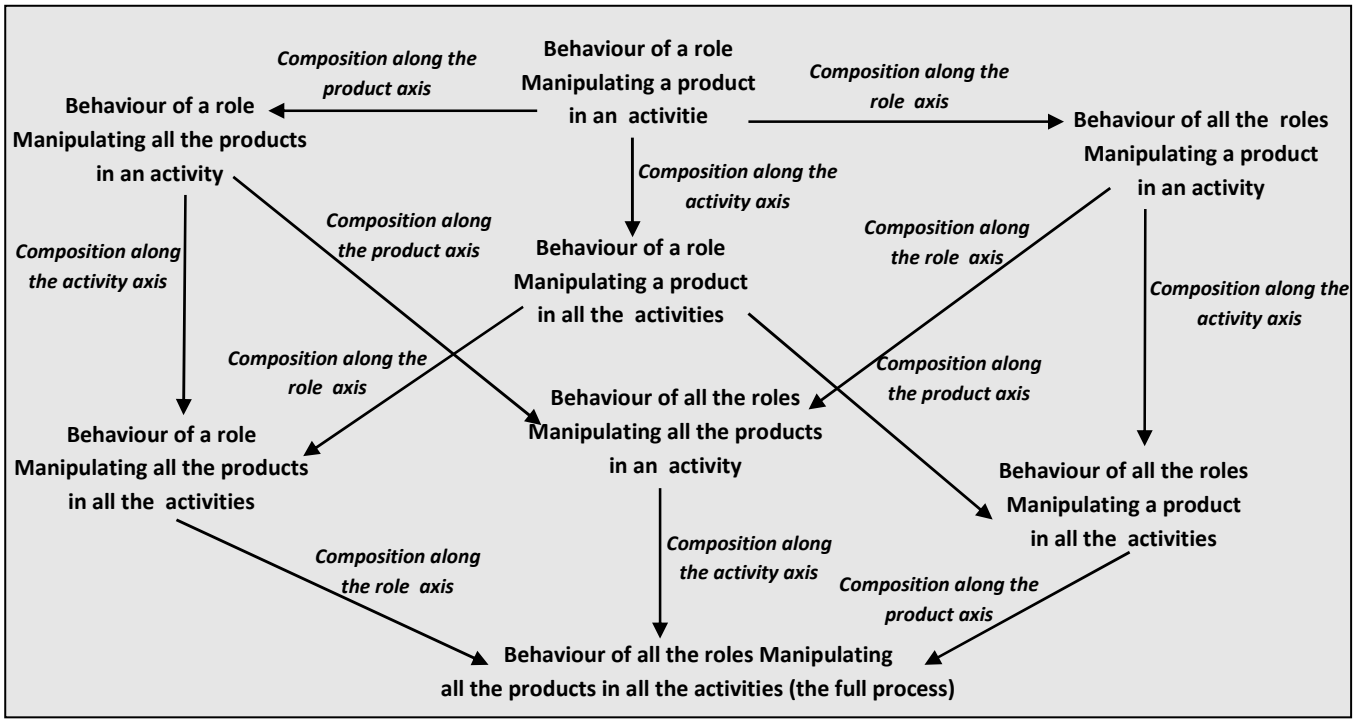


Figure 2.6: schéma de composition [BEL, 96]

4- SPEM 2.0

SPEM (*Software & Systems Process Engineering Metamodel*) est un méta-modèle développé par l'Object Management Group (*OMG*). Cette section est principalement basée sur la spécification SPEM2.0 **[SPE, 08].**

Depuis les années 80, il y a eu divers méta-modèle de procédé, et enfin l'OMG a décidé de standardiser un méta-modèle de procédé. Ainsi, la spécification de SPEM 1.0 est apparue en 2002. En 2005, SPEM1.1 **[SPE, 05]** a été libéré avec une mise à jour mineure. SPEM 1.1 avait plusieurs lacunes, par exemple, la sémantique été ambiguë et difficile à comprendre. Par la suite, SPEM 2.0 a été publié en 2007 fixant les failles dans les versions précédentes.

Le méta-modèle SPEM a changé en profondeur entre la version 1.1 et la version 2.0. En effet, SPEM 2.0 différencie l'aspect opération de l'aspect temporel.
Le méta-modèle de SPEM 2.0 est composé de plusieurs packages (*Figure 2.7*).

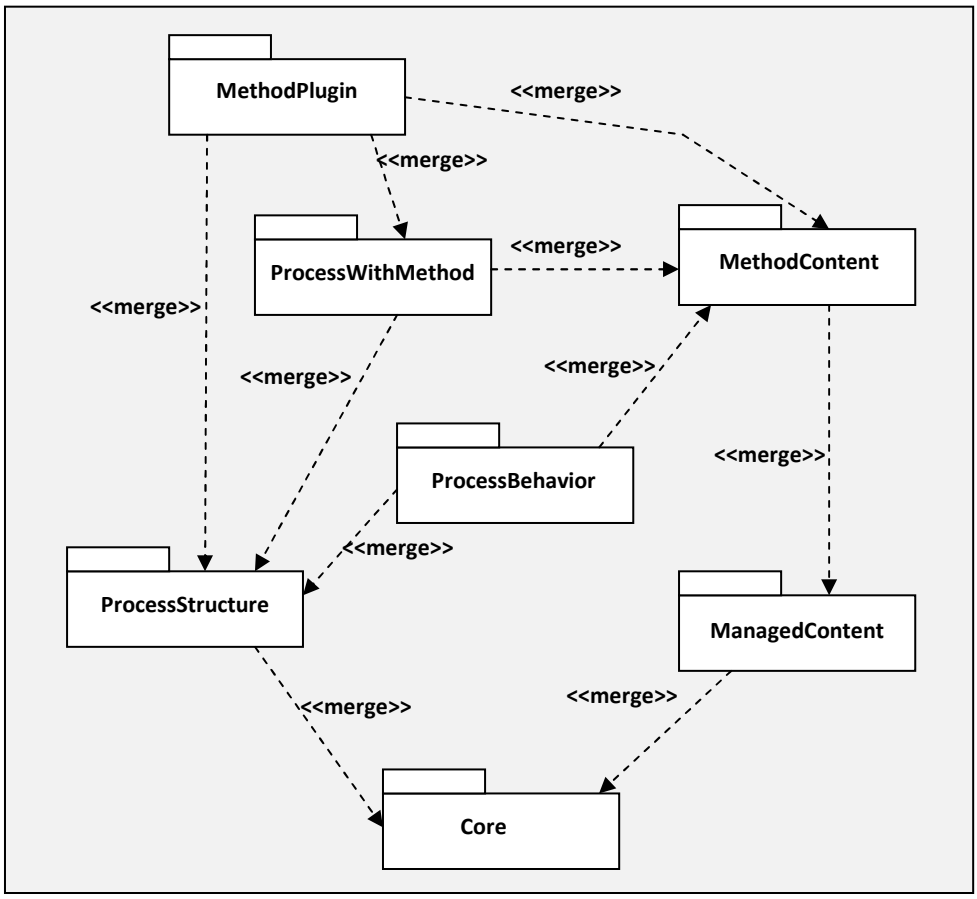


Figure 2.7- Structure du méta-modèle de SPEM 2.0 [SPE, 08].

- *MethodContent* décrit des rôles qui exécutent des tâches pour réaliser des produits, sans préciser l'enchainement des tâches et leur place précise dans un processus.

- *ProcessStructure* définit la structure des processus, c'est-à-dire l'ordonnancement des activités et la composition des structures des processus.

- *Core* contient les classes communes définies dans les différents packages. Ce package contient plusieurs éléments importants.

- *ManagedContent* permet aux utilisateurs de SPEM de décrire leurs processus en langage naturelle, en gardant un lien entre le processus modélisé et la description.

- *ProcessBehavior* offre le moyen de lier un élément de procédé SPEM2.0 avec un comportement externe comme des diagrammes d'activités UML2.0 ou des modèles BPMN (*Business Process Modeling Notation*).

- *ProcessWithMethod* spécifie les liens entre les éléments des packages Method Content et Process Structure, pour réutiliser les opérations à réaliser dans les structures temporelles spécifiques au processus défini.

- *MethodPlugin* permet d'étendre et de personnaliser des instances de Method Content et Process Structure sans les modifier directement, mais grâce à un système d'extension, le plugin en lui-même.

La figure 2.8 présente un exemple de modèle de processus instancié à partir du méta-modèle SPEM 2.0. Il présente les rôles et produits impliqués dans la tâche ''*Prioritize Use Cases*'' Cette tâche est exécutée par un rôle principal, ''*Architect*" et des rôles secondaires : ''*Analyst*'' et ''*Manager*''. Le produit en entrée est : ''*Risk List*'', et le produit en sortie est : ''*Requirements* Attributes''.

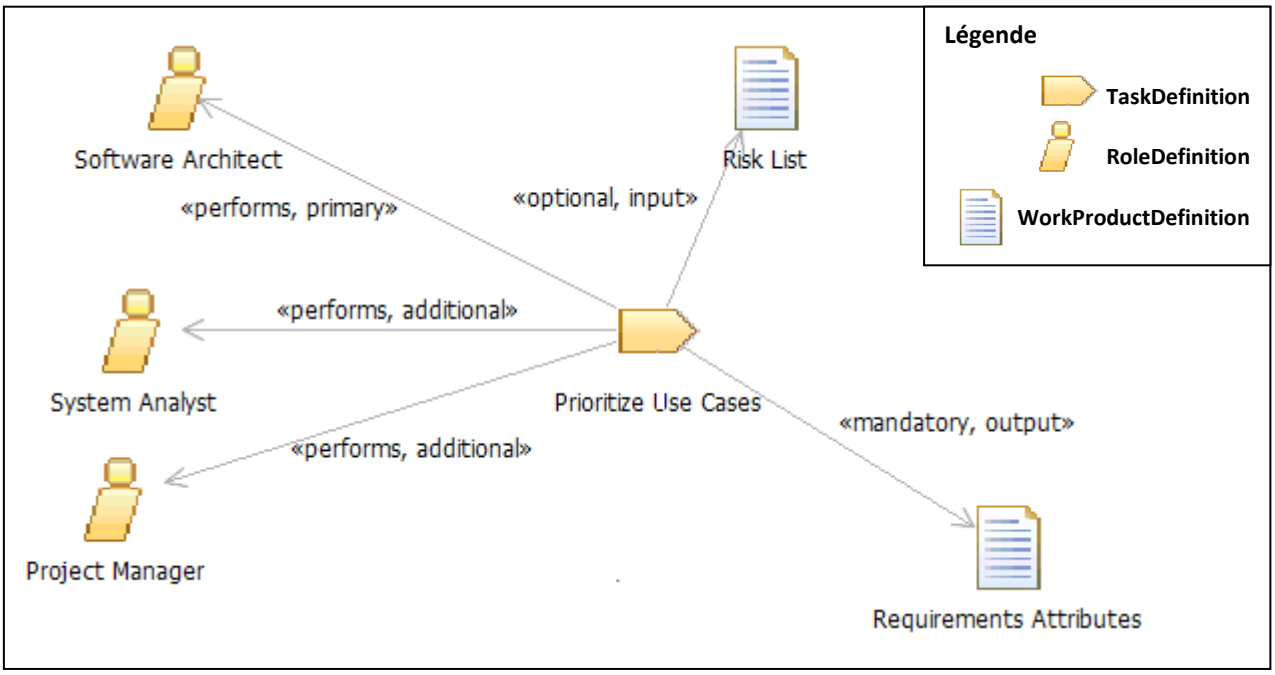


Figure 2.8-Exemple de modèle de procédé (SPEM 2.0)

5- SYNTHESE

Les différents modèles présentés, mettent l'accent sur un certain nombre de concepts et en mettent d'autres de côté, notamment :

- La cohérence : Définit des mécanismes qui permettent de contrôler la cohérence d'un processus, avant et pendant son exécution.
- Le mécanisme d'interaction : Fournit des mécanismes d'interaction fiables entre composants, pour que les connaissances fournisses et requises soit associées de façon cohérente.
- La réutilisation : Capitalise l'expérience acquise lors de la définition d'un procédé, en l'encapsulant dans un composant doté d'interfaces assemblables.
- L'hétérogénéité : Masque la diversité des procédés logiciels soit sur l'aspect :
    - Sémantique : due à la diversité des concepts utilisés par les composants. Chaque composant peut offrir des concepts différents pour gérer l'aspect du procédé auquel il est dédié.
    - Syntaxique : les composants peuvent offrir un formalisme différent pour décrire l'aspect de procédé qui leur correspond.

Le tableau 2.1 synthétise le support des ces différents concepts par les modèles de composants procédés présentés dans ce chapitre.

	PYNODE	OPC	RHODES	SPEM 2.0
Réutilisation	Interne au système			Comportement externe.
Cohérence	interfaces de composants	Le schéma permet de décrire les entités de base et les interactions entre eux.	- Assertions (*pré/post condition*) - Règles de guidage méthodologique	Pris en charge par divers mécanismes : assertions (*prés/post condition*), règle de guidage, des contraint,…
Mécanismes d'interaction	vues d'observation et les vues exécution	messages, évènements et les méthodes	les méthodes et les schémas de réalisation (*heuristique de développement*)	Connecteurs implicites
Hétérogénéité	Syntaxique	Syntaxique	Homogène	Syntaxique

Tableau 2.1 -Synthèse des concepts présentés dans les ECP étudiés

A partir du tableau 2.1, on a pu conclure certaines remarques, particulièrement :

1- Réutilisation :

- les approches PYNODE, OPC, et RHODES, ne considèrent que les composants développés dans leur système, en d'autre terme, les composants sont réutilisés dans le même environnement pour le quel ils ont été conçus. Par exemple, il n'est pas possible dans un environnement RHODES d'utiliser des composants conçus à partir de l'environnement PYNODE et vice-versa.

- SPEM fournit, à travers le package Process Behavior, un moyen de relier les éléments de procédé avec un comportement extérieur, dans l'objectif d'offrir aux procédés la possibilité de choisir une méthode d'exécution qui correspond le mieux à leurs besoins, dans ce cas, cette approche revendique une plus grande flexibilité.

2- Mécanismes d'interaction se limitent à :

- Des échanges de messages, d'événements, ou par une simple fonction (exemple : la fonction *call* dans *RHODES*). En conséquence, la modification de la topologie d'un procédé nécessite des modifications des composants qui en dépendent.

- Par le biais des connecteurs implicites, comme dans le cas de SPEM 2.0, qui ne jouent aucun rôle pour faciliter la connexion entre les composants, leurs rôles se limite simplement à s'assurer de la communication entre les composants .En outre, certaines propriétés des connecteurs (*sémantique, évolution...*) (*chapitre 3*) ne sont pas prises en compte.

3- Hétérogénéité : les ECP étudiés, ne permettent pas l'interopérabilité sémantique, et ceci, même si certains d'autres eux, peuvent offrir plus de flexibilité, ils restent généralement focalisés sur la syntaxe des informations échangées, qui doit être respectée par les différents composants qui interopèrent. Cependant, la cohérence dans le sens où les informations échangées doivent avoir la même signification au sein de ces systèmes reste peu explorée.

V- CONCLUSION

Ce chapitre présente, les générations d'ECP à base de composants comme une solution très prometteuse, car elle pallie plusieurs problèmes de modélisation classique, présentés dans le chapitre 1.

En effet, les approches « *composants de procédés* » fournissent des environnements qui permettent de gérer la complexité, en divisant un procédé en des sous procédés de moindre

complexité tout en étant suffisamment pertinents pour être réutilisables, offrant ainsi d'importants gains de productivité et de qualité.

Toutefois, Une des lacunes des ECP étudiés, est l'absence d'un modèle qui gouverne les interactions entre les différentes entités du procédé. Une autre lacune de ces ECP, est qu'ils ne permettent pas l'interopérabilité sémantique des composants (*par exemple : un composant RHODES ne peut être utilisé avec un composant PYNODE ou OPC, dans un même procédé*).

Les langages de description d'architecture logicielle (ADL) répondent en partie à ces reproches, en effet, une des clés de l'initiative ADL est la possibilité de pouvoir réutiliser les composants/connecteurs dans des applications et environnements multiples, sans avoir à se soucier des outils utilisés pour les créer. Ceci implique, entre autres, que le contenu soit séparé des contraintes liées au contexte et aux spécificités du logiciel d'exécution de telle sorte qu'il puisse être inclus dans d'autres applications.

Ainsi, d'un coté, le besoin de fournir des outils et des techniques qui aident à construire et/ou à contrôler le procédé tout au long de son cycle de vie, et d'un autre coté les progrès des travaux liés aux architectures logicielles, nous ont amenés à tenter d'allier ces deux approches pour une meilleure efficacité.

Le chapitre suivant, va donc, porter sur l'étude des architectures logicielles, plus précisément, nous allons voir : En quoi l'utilisation des techniques/outils offerts par les ADL peut-elle améliorer le travail des différents acteurs de processus logiciels.

CHAPITRE 3

ARCHITECTURES LOGICIELLES À BASE DE COMPOSANTS

I- INTRODUCTION

Depuis l'émergence du domaine des architectures logicielles, il y a plus d'une quinzaine d'années, de nombreux travaux ont été entrepris pour l'utilisation formelle des architectures, avec tout d'abord une forte concentration aux États-Unis puis partout dans le monde.

Parmi ces travaux, une variété de langages a été crées pour fournir un vocabulaire précis et commun pour les différents acteurs *(architectes, concepteurs, développeurs, intégrateurs et testeurs*) avec des notations pour spécifier les modèles et pour pouvoir raisonner dessus.
Une sémantique formelle s'est substituée aux simples conventions de représentation. Est alors apparue, au cours des années 90, la dénomination de *langage de description d'architecture* ou **ADL**.

Définition

*Un langage de description d'architecture (ou **ADL** pour Architecture Description Language) fournit une syntaxe et une sémantique formelle pour modéliser l'architecture conceptuelle d'un système. Il permet de spécifier abstraitement les éléments d'une architecture sans définir leur implémentation* **[GHI, 06].**

En fait, les ADL sont un support pour la description de la structure de l'application basée sur les composants. Ils offrent des facilités de réutilisation des composants et des moyens de description de la composition par description des dépendances entre composants et des règles de communication à respecter.
Les langages de description d'architectures permettent de décrire formellement, sans ambiguïté, les architectures logicielles.

Actuellement les ADL, représentent le meilleur support à la définition d'architecture logicielle. Ils ont en commun une capacité à décrire l'architecture des systèmes mais

diffèrent largement dans leurs motivations et leurs formalismes d'écriture. Ils sont en général graphiques et textuels et possèdent des outils associés.

Ce chapitre présente, des notions et des concepts relatifs aux architectures logicielles à base de composants ainsi qu'à leurs langages de description, ensuite il étudie les différentes approches proposées par les principaux ADL, pour faire face à la problématique de réutilisation des composants de procédés.

II- NOTION D'ARCHITECTURE LOGICIELLE

1- DEFINITION

Plusieurs définitions du terme « ***architectures logicielles*** » sont proposées dans la littérature, il n'y a pas de définition standard et universellement acceptée, bien qu'on en trouve un grand nombre. Nous en proposons ici celle donnée par l'IEEE (IEEE Std 1471-2000):

Définition

> *Une **architecture logicielle** est l'organisation fondamentale d'un système incarné dans ses composants, leurs relations avec chacun des autres et avec l'environnement, et les principes guidant sa conception et son évolution.*

2- PROPRIETES DEFINIES PAR UNE ARCHITECTURE LOGICIELLE

Une description architecturale est concernée avant tout par les aspects suivants **[LEY, 04]** :

- **La structure** : *composants et interactions*,
  Une architecture caractérise la structure d'un système en termes d'éléments fonctionnels et de leurs interactions. Ainsi, une architecture est conçue comme une configuration de composants interagissant. Il n'est question ni du cahier des charges, ni des détails d'implémentation (*tels des algorithmes ou des structures de données*). La description d'une architecture doit être assez simple pour permettre de raisonner sur un système et de pouvoir faire des prévisions.

- **Le comportement** : *fonctionnalités et protocoles de communication, dynamisme, évolution,*
  Les interactions entre les composants architecturaux fournissent un vocabulaire riche pour les concepteurs. Bien que les interactions puissent être aussi simples que des appels de procédure ou des variables de données partagées, elles représentent souvent des formes de communication plus complexe. On cite par exemples : les interactions client-serveur (*avec des règles sur l'initialisation, la terminaison, et la gestion des exceptions*), les évènements (*avec de multiples destinataires*), et les

protocoles d'accès aux bases de données (*avec des protocoles pour l'invocation des transactions*).
Les interactions sont définies au sein du comportement définissant l'ordonnancement des communications d'un élément architectural.

- **Les propriétés globales** *: propriétés fonctionnelles ou non fonctionnelles,*
Une architecture décrit typiquement les propriétés globales d'un système. Ainsi, les problèmes auxquels elle répond sont habituellement au niveau système, tel que les temps de traitement, la latence, ou la résilience d'une part du système à une panne.

III- STYLES ARCHITECTURAUX

1- DEFINITION ET BENEFICES GENERAUX

Définition

> *Un style architectural caractérise une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques* **[RAT, 05].**

Le but principal des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, par exemple : de la structure, de l'utilisation des ressources des composants et des connecteurs dans un système….

Un style architectural fournit **[RAT, 05]** :

- Un vocabulaire : pour spécifier les types spécifiques de composants utilisables ;
- Des règles de conception : pour spécifier les compositions d'éléments qui sont permises ;
- Une interprétation sémantique : pour définir la signification des compositions d'éléments contraintes par les règles de conception ;
- Les analyses qui peuvent être appliquées sur les systèmes construits dans ce style.

D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires.

En outre, l'utilisation des styles architecturaux comporte des intérêts précis **[RAT, 05]**:

- Elle promeut la réutilisation au niveau de la conception (*et non seulement au niveau du code, comme c'est le cas de la plupart des techniques de réutilisation*) ;

- Elle peut donc aussi mener à une réutilisation significative de code ;
- Elle permet de normaliser une famille d'architectures, ce qui améliore la compréhension de l'organisation d'un système ;

- Elle permet l'utilisation d'analyses spécifiques au style concerné.

Les analyses spécifiques au style fixent des contraintes de construction, de trois types **[REV, 05]** :

- Les contraintes topologiques *:* précisent le type des éléments de construction, le nombre d'occurrences possibles au sein de l'architecture et les règles de configuration contraignant leurs interactions (*par exemple deux composants sont toujours liés par l'intermédiaire d'un connecteur*).

- Les contraintes comportementales *:* définissent d'une part, le comportement des éléments de l'architecture et, d'autre part, le comportement globale de l'architecture (*par exemple il n'y a pas d'interblocage entre les différents composants*).

- Les contraintes d'attributs *:* concernent les aspects non structurels et non fonctionnels d'une architecture. Les attributs apportent des informations complémentaires sur les éléments architecturaux en contraignant leur type, leur nom et leur plage de valeurs (*par exemple la durée de traitement réalisée par un composant ne doit pas dépasser 0.5 seconde*). Elles peuvent également spécifier la manière dont les valeurs des attributs sont liées avec d'autres aspects architecturaux (*topologie et comportement*).

IV- PRINCIPALES CARACTERISTIQUES DES ADL

Afin de classifier et de mener des études approfondies sur les ADL, de nombreux travaux **[MED, 00]**, **[ACC, 02]**, **[ELB, 06]**… se sont focalisés sur la définition des caractéristiques et des critères que doit prendre en compte un langage pour être considéré comme étant un langage de description d'architecture logicielle.
En effet, les ADL sont une famille de langage plus ou moins spécialisés, leur rôle est la modélisation de l'architecture, ils peuvent être très différents mais possèdent tous en générale la capacité de modéliser les composants, l'interface des composants, les connecteurs et la configuration du système, les autres caractéristiques sont essentiellement dues aux besoins auxquels ils répondent et à l'exploitation voulue des descriptions produites.

Cette section présente brièvement les éléments fondamentaux sous-jacents aux ADL. La classification présentée dans cette partie est inspirée des études antérieure rencontrés dans la littérature, citons par exemple : **[MED, 00]**, **[MED, 97]**, **[ACC, 02]**, **[ELB, 06]**, et **[MAR, 02]**.

- Composant (*Component*) : Un composant, au sens architectural, est un lieu de traitement ou de stockage de données plus ou moins complexe. Par définition tous les ADL décrivent les composants, comme un couple spécification-code : la spécification donne les interfaces, les propriétés du composant ; le code correspond à l a mise en œuvre de la spécification par le composant ;

- Connecteur (*Connector*) : Le connecteur, est un concept architectural dont la fonction est la modélisation des interactions entre les composants et des lois qui régissent ces interactions. A la différence des composants, les connecteurs n'ont pas forcément d'implémentations correspondantes dans le système.

- Configuration ou la topologie (*Architectural Configuration)* **:** la configuration, est le graphe de connexion qui permet de décrire la structure de l'architecture. Sa description permet de décrire le système, ses propriétés et ses qualités.

Plusieurs caractéristiques agrémentent ces trois concepts, qui sont développés dans les sections suivantes.

Un quatrième concept est toutefois abordé dans **[MED, 00] [MED, 97],** il s'agit des outils associés aux ADL (*Tool Support*). Ce concept n'est pas détaillé dans ce mémoire. Cependant, une brève indication peut être citée : les outils de support permettent au concepteur d'application de travailler sur la base des trois concepts précédents. Ils fournissent donc des aides à la conception, des vues multiples de l'architecture, et des fonctionnalités d'analyse, de raffinement ou de compilation.

1- CARACTERISTIQUES DES COMPOSANTS ET DES CONNECTEURS

Composants et connecteurs partagent six (06) caractéristiques globales : l'interface, le type, la sémantique, les contraintes, l'évolution et les propriétés non fonctionnelles. Ces caractéristiques sont définies dans le tableau 4.1, avec la signification dans le cas précis des composants ou des connecteurs.

	SIGNIFICATION DES CARACTERISTIQUES POUR	
CARACTERISTIQUES	**COMPOSANT**	**CONNECTEUR**
Interface (*Interface***) :** *ensemble des points d'interaction avec l'extérieur.*	Services fonctionnels du composant : les services offerts et requis par celui-ci	Mécanismes de connexion entre composants. Certains ADL décrivent ces interactions comme des rôles

Type (*Types***) :** *abstraction des fonctionnalités, en vue de leur réutilisation.*	Abstraction des fonctionnalités fournies par le composant.	Abstraction des mécanismes de communication, de coordination ou de médiation entre composants.
Sémantique (*Semantics***) :** *abstraction du comportement, qui permet de spécifier les aspects dynamiques et les contraintes de l'architecture. Le modèle sémantique assure aussi des projections cohérentes entre les niveaux d'abstraction de l'architecture.*	Abstraction des fonctionnalités du composant (et qui seront utilisées par l'application)	Abstraction des protocoles d'interaction
Contraintes (*Constraint***) :** *propriétés ou assertions qui doivent être vérifiées sous peine d'incohérence du système (qui devient alors inacceptable).*	Limites d'utilisation du composant et des dépendances intra composants	Limites d'utilisation du protocole d'interaction mis en œuvre par le connecteur et des dépendances intra connecteurs
Évolution (*Evolution***) :** *un ADL doit permettre l'évolution des éléments de l'architecture, donc permettre la modification de leurs propriétés. En général l'évolution est assurée par des techniques de sous-typage ou de raffinement.*	Evolution de l'interface, du comportement, de l'implémentation du composant	Evolution de l'interface et du comportement du connecteur ; Permet notamment de faire évoluer les protocoles d'interaction
Propriétés non fonctionnelles (*Non Functional Properties***) :** *les propriétés non fonctionnelles permettent de spécifier les aspects liés à la sécurité, la performance, la portabilité, etc. La signification ne diffère pas entre les propriétés non fonctionnelles du composant et celles du connecteur.*		

Tableau 4.1 : Caractéristiques des composants et des connecteurs.

2- CARACTERISTIQUES DES CONFIGURATIONS

Un ADL doit fournir les possibilités de configuration suivantes :

- **La compréhensibilité (***Understandability***):** Une configuration doit permettre de fournir une syntaxe simple et une sémantique permettant de faciliter la communication entre les différents partenaires d'un projet et de rendre compréhensible la structure d'une application.
- **Le mécanisme de composition-décomposition (***Compositionability***)** : La définition de la configuration d'une application doit permettre la modélisation et la représentation de la composition à différents niveaux de détail (*composition hiérarchique*).

- **Le raffinement et la traçabilité (***Refinement and traceability***)**: De la spécification à la réalisation du système exécutable, l'architecture se raffine en passant par différents niveaux d'abstraction. Pour maintenir l'exactitude et la consistance de l'architecture à travers ces niveaux, les ADL's doivent pouvoir spécifier et tracer ces évolutions.

- **La qualité d'hétérogénéité (***Heterogeneity***)** : Le développement de systèmes de grande taille s'appuie sur l'utilisation et la réutilisation d'éléments hétérogènes par leurs niveaux de granularité, les formats de spécification, les langages de programmation utilisés, les plates formes d'exécution ou les protocoles d'interaction. A ce niveau, il est souhaitable que les langages soient "ouverts", et facilitent la spécification et le développement avec des composants et des connecteurs hétérogènes.

- **La capacité de croissance '' Le passage à l'échelle '' (***Scalability***)** : permet la réalisation d'applications complexes et dont la taille peut devenir importante.

- **La capacité d'évolution (***Evolvability***)** : La conception de nouveaux systèmes s'effectue souvent à partir de versions antérieures dont on adapte ou améliore les fonctionnalités. Cela se traduira essentiellement par la possibilité d'ajouter, de retirer ou de remplacer des composants ou des connecteurs.

- **Le dynamisme architectural (***Dynamism***)**: Certains systèmes peuvent nécessiter des modifications de leur configuration ou le changement d'éléments en cours d'exécution. C'est le cas de systèmes critiques ou des systèmes que l'on ne peut arrêter. Les ADL peuvent fournir des moyens de modélisation et d'évaluation des changements dynamiques de l'architecture et des moyens techniques pour les rendre effectifs.

- **Les contraintes de configuration (***Constraints***)** : Les contraintes liées à la configuration viennent en complément des contraintes définies pour chaque composant et chaque connecteur. Elles décrivent les dépendances entre les composants et les connecteurs et concernent des caractéristiques liées à l'assemblage de composants que l'on qualifie de contraintes inter composants (*contraintes globales*).

- **Les propriétés non fonctionnelles (***Non Functional Properties***)** : Certaines propriétés non fonctionnelles ne concernant ni les connecteurs et ni les composants doivent être exprimées au niveau de la configuration. Ces contraintes sont généralement liées à l'environnement d'exécution.

V- CLASSIFICATION DES ADL ADOPTEE

Plusieurs critères ont été proposés dans la littérature pour classifier et comparer les ADL **[MED, 96] [ACC, 02] [ELB, 06] [MAR, 02]**, tels que :

- L'aspect du système auquel ils s'intéressent : structurel et/ou comportemental ;
- Leurs formalismes de base : formels ou semi formels ;
- Leur origine : écoles américaines ou écoles européennes.

Pour classer les ADL, nous avons opté pour une classification basée sur le critère de *spécificité* ou non d'un ADL à un domaine particulier. Ainsi, En se basant sur ce critère, cette classification distingue les ADL spécifiques au domaine des procédés logiciels et les ADL qui peuvent adresser plusieurs domaines (*les ADL générique*).

1- ADL SPECIFIQUES AUX PROCEDES LOGICIELS

Les ADL spécifiques à un domaine particulier, se focalisent généralement sur des propriétés particulières d'un système et proposent souvent une syntaxe propre à ces propriétés. Une panoplie d'ADL spécifiques a été rencontré dans la littérature : Darwin **[DAR]** vise la reconfiguration dynamique des architectures, Rapide **[RAP]** a pour objectif la description architecturale de systèmes événementiels, Aesop **[AES]** se focalise sur la définition des styles architecturaux, etc. Cependant, aucun ADL conçus pour décrire les procédés logiciels, et définissant clairement une syntaxe propre aux concepts de procédé, n'a été trouvé dans nos recherches.

Toutefois, des travaux qui se basent sur la modélisation architecturale des procédés ont été rencontrés, citons :

- Boehm et al. **[BOE, 96],** proposent un modèle architectural ''*Toaster Model*'' qui se base sur l'identification des éléments architecturaux et les spécifications de leurs interfaces en terme de post et prés conditions entre les différents éléments du procédé et guide le développement en se basant sur des points d'ancrage.

- Fei Dai et al. **[DAI, 08],** définissent un langage de description de l'évolution des procédés (*EPDL, Evolution Process Description Language*). Dans le but de décrire la structure générale d'un processus d'évolution du procédé. Son approche peut être vu comme un modèle de composition des composants d'évolution des procédés (*EPC, Evolution Process Component*).

- Borsoi et al. **[BOR, 08],** définissent l'architecture des processus logiciels comme un ensemble de vues représentées par des modèles et des relations entre eux. Ces modèles sont à des niveaux d'abstraction distincts. La méthode se base sur la définition des phases séquentielles (*partie principale de la méthode*) qui visent à définir les modèles de l'architecture des procédés dans des niveaux d'abstractions différentes. Pour représenter ces modèles, des concepts de base et éléments de notation ont été définis.

- Aoussat et al. **[AOU, 10],** proposent une modélisation architecturale des procédés, dans le but d'accroître leur réutilisation. Son approche est basée sur deux étapes

principales : l'utilisation d'une ontologie de domaine pour capitaliser la connaissance des procédés et l'inférence de nouveaux modèles de procédés et de leur déploiement, à partir de l'ontologie définie précédemment.

2 - ADL GENERIQUE

Les ADL génériques tentent d'offrir une base qui regroupe l'ensemble des concepts communément admis par les ADL, tout en offrant les moyens à l'utilisateur de définir d'autres ou d'intégrer des spécifications établies avec d'autres ADL. Ce type d'ADL peut, servir par conséquent comme une passerelle entre les différents ADL existants. Les ADL les plus représentatifs de cette catégorie sont ACME **[ACM]** et xADL2.0 **[XAD]**.

- **ACME** a pour but principal de prendre en compte les caractéristiques communes de l'ensemble des ADL. Comme le langage ACME se focalise sur l'aspect structurel des architectures logicielles, il offre un mécanisme d'annotation pour favoriser l'intégration d'autres informations spécifiques ne concernant pas la structure et apportées par certains ADL (*telle que la spécification du comportement par exemple*).

- **xADL2.0** est défini comme un langage de description flexible et extensible de schémas XML. Ses auteurs ont largement tiré profit d'XML pour apporter cette souplesse et bénéficier des outils supports d'XML. Le langage offre une base composée des concepts communément admis par les ADL et permet ensuite au concepteur de définir d'autres concepts qu'il juge adaptés pour son système.

VI- SYNTHESE ET CONCLUSION

Nous avons vu dans ce chapitre qu'un ADL doit, en principe, pouvoir décrire une architecture logicielle sous une forme dite des trois C : Composant, Connecteur et Configuration **[MED, 00]**.

Les composants représentent les unités de calcul et de stockage de données dans un système logiciel. L'interaction entre ces composants est encapsulée par les connecteurs. Dans la configuration, un nombre de composants et un nombre de connecteurs sont instanciés. Ils sont liés entre eux afin de former le système complet. Une partie des ADL répond à cette description. Cependant, une autre partie diverge (*Certains ADL comme Rapide ou Darwin, n'explicitent pas la notion de connecteur*).

Nous avons présenté dans la section V.1 une classification des ADL selon la spécificité ou non d'un ADL à un domaine particulier. Nous nous sommes intéressés au domaine des procédés logiciels. Mais vu le manque de ce type de langages, on s'est alors, intéressé aux langages génériques qui peuvent exprimer les concepts de base communs aux différents ADL, afin de les correspondre aux concepts de base liés aux procédés de développement.

Nous concluons notre étude, par le fait que les ADL apportent un gain significatif pour le développement d'un processus logiciel. En effet, les ADL offrent de larges possibilités de modélisation. Ils ont permis un nouveau mode de développement axé sur la phase de conception plutôt que sur la phase d'implémentation. L'utilisation de ces langages pour la modélisation des systèmes actuels est de plus en plus explorée. Il est donc tout à fait justifié d'axer nos travaux sur la possibilité de bénéficier des atouts de cette approche pour améliorer les procédés de développement de logiciel.

Toutefois, afin de rapprocher ces deux axes de recherche, à savoir l'ingénierie des procédés et les architectures logicielles, nous devons avoir des modèles abstraits qui apportent une simplification dans la compréhension et la manipulation de la connaissance des différentes technologies utilisées. C'est pourquoi, nous avons eu recours aux méthodes définies par l'IDM (*Ingénierie Dirigée par les Modèles*), qu'on détaille dans le chapitre prochain.

CHAPITRE 4

INGENIERIE DIRIGEE PAR LES MODELES

I- INTRODUCTION

L'Ingénierie Dirigée par les Modèles (**IDM**), ou Model Driven Engineering (**MDE**) en anglais, ou aussi (**MDD)** Model-Driven Development, 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. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles.

Ce chapitre présente les principes clés de l'IDM. Il introduit dans un premier temps la notion de modèle, détaille ensuite l'un des axes principaux de l'IDM, la transformation de modèle. Enfin, une synthèse résume l'approche IDM et met en avant ses principaux avantages qui justifient notre choix.

II- DEFINITIONS ET CONCEPTS FONDAMENTAUX DE L'IDM

Le concept central de l'IDM est la notion de modèle pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, des initiatives académiques et industrielles, ont permit de clarifier les concepts de cette discipline.

Cette section définit les concepts de base de l'IDM et les relations qui existent entre eux. Elle vise avant tout à préciser la terminologie employée dans la suite du document et est adaptée à partir des travaux de : **[BLA, 05] [THO, 08] [JOU, 05] [NGU, 08] [MAR, 02] [BEZ, 04]** et **[COM, 08].**

1- DEFINITIONS

- Système

> ***Un système*** *est un ensemble d'éléments interdépendants formant un tout complexe. Il représente l'entité à modéliser.*

- Modèle

> ***Un modèle*** *est une abstraction d'un système, il représente une spécification formelle des fonctions, des propriétés, de la structure et/ou du comportement du système dans son environnement. Construit dans une intention particulière.*

La notion de modèle est l'élément central de toute l'approche. Un modèle est définit comme une abstraction dans la mesure où il contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis dans la mesure où les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui sera faite du modèle.

- Méta-modèle

> ***Un méta-modèle*** *est la spécification d'une abstraction d'un ou plusieurs modèles. Cette spécification définit un ensemble de concepts important pour exprimer des modèles, ainsi que les relations entre ces concepts. Le méta-modèle définit la terminologie à utiliser pour définir des modèles.*

Le méta-modèle décrit le lien existant entre un modèle et le système qu'il représente. On dit qu'un modèle est *conforme* à un méta-modèle si l'ensemble des éléments du modèle sont définis par le méta-modèle. Cette notion de conformité est essentielle à l'ingénierie des modèles mais n'est pas nouvelle : un texte est conforme à une orthographe et une grammaire, un programme JAVA est conforme au langage JAVA et un document XML est conforme à sa DTD…

- Méta-méta-modèle

> ***Un méta-méta-modèle*** *est un modèle d'un langage de modélisation. C'est une abstraction d'un langage permettant de décrire des modèles. Généralement il a la capacité de se décrire lui-même.*

Le méta-méta-modèle représente la vue la plus abstraite et correspond à la couche d'abstraction la plus haute. Afin d'éviter une infinité de niveaux d'abstraction, le méta-méta-modèle est, en effet, auto-descriptif, c'est-à-dire qu'il se définit lui-même.
Le méta-méta-modèle permet de spécifier les éléments qui composent un méta-modèle et la manière de les manipuler.

2 - RELATION ENTRE LES DIFFERENTS CONCEPTS

La figure 4.1, extraite de la thèse de Fleurey **[FLE, 06]** et complétée par des notions de la thèse de Richard Lemesle **[LEM, 00]** résume les différents niveaux de modélisation (*M1 : modèle, M2 : méta-modèle et M3 : méta-méta-modèle*), ainsi que les relations qui existent entre les différents concepts. Cela permet de mettre en évidence les relations qui existent entre :

- **Modèle et méta-modèle** : un modèle est défini à partir d'un méta-modèle, on dit généralement qu'un modèle est conforme à son méta-modèle.
- **Entité et méta-Entité** : Toute entité est explicitement définie dans un modèle (*figure 4.1*). De plus, toute entité doit être liée explicitement à la méta-entité (*relation méta*) représentant son type et définie dans un méta-modèle.
- **Modèles et langages** : un modèle est exprimé dans un langage de modélisation (*c'est-à-dire qu'il appartient à ce langage*) et est conforme au modèle qui représente ce langage. Cela permet également de préciser ce que l'on entend par autodéfinition (*ou boot-strap*) pour le méta-méta-modèle : le méta-méta-modèle représente le langage dans lequel il est exprimé, il est donc conforme à lui même.

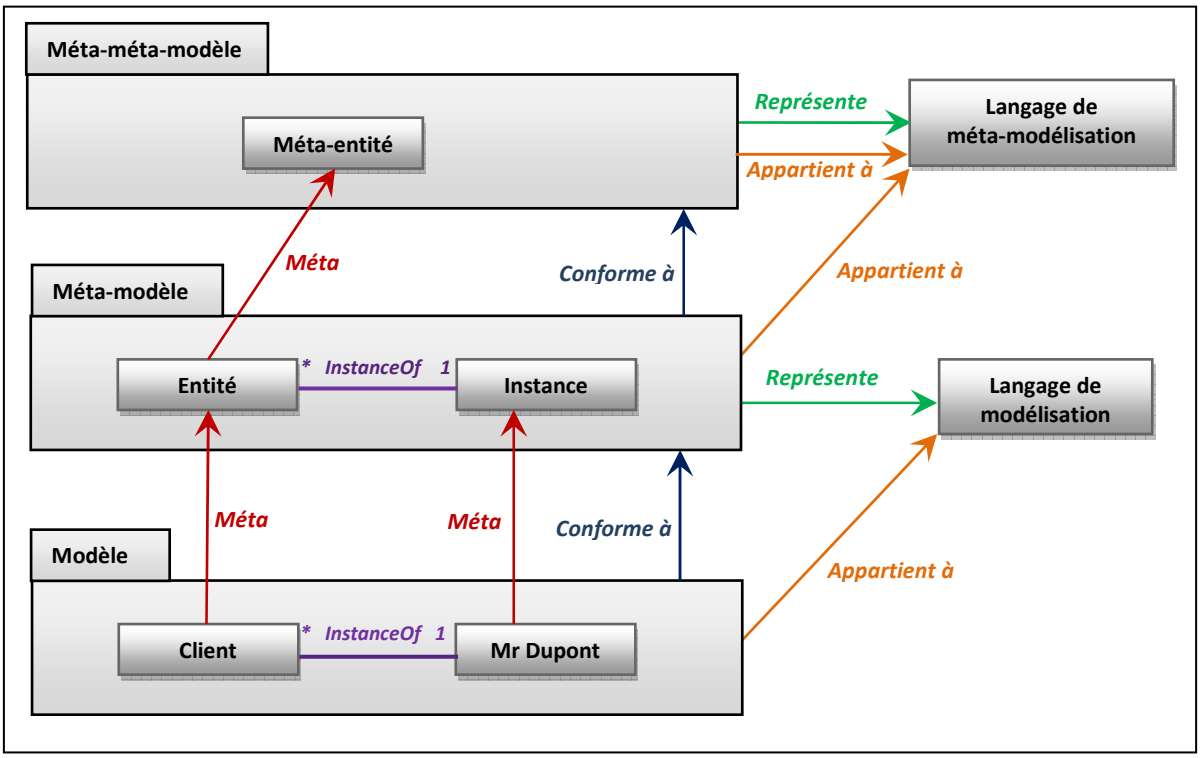


Figure 4.1 : niveaux de modélisation

3- SYNTHESE

Dans cette section, il a été présenté les concepts et relations essentielles dédiés à la méta-modélisation qui peuvent être regroupés dans le méta-modèle réflexif (ou le noyau réflexif), défini par Lemesle **[LEM, 00],** et illustré dans la figure 4.2.

Dans ce noyau tout est considéré comme une entité. Ainsi, un méta-modèle est un modèle (*un sous-type de modèle : un méta-modèle n'est autre qu'un modèle de modèle*) qui lui même est une entité. De même, une méta-entité est une entité (*un sous-type d'entité : une méta-entité n'est autre qu'une entité qui va permettre de décrire un type d'entité*).

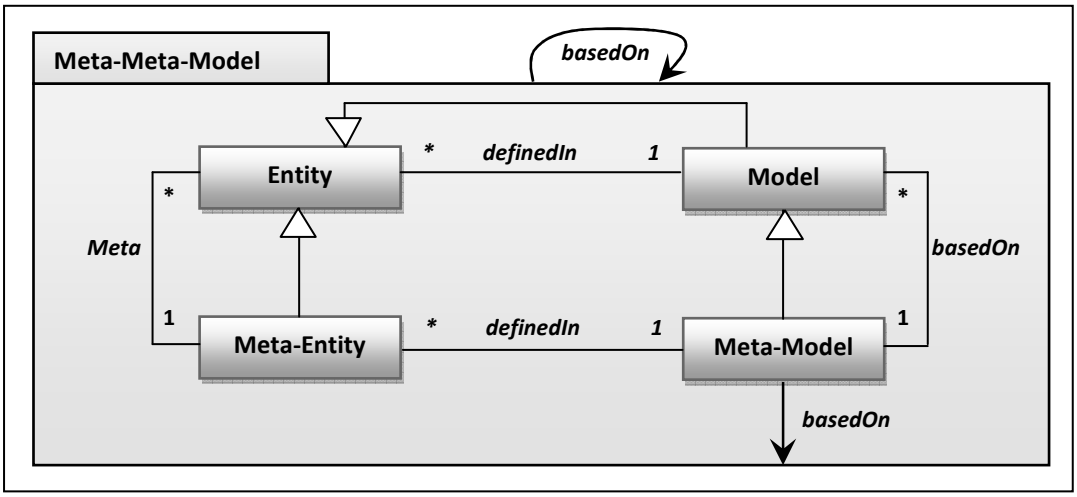


Figure 4.2 : Le noyau réflexif de LEMESLE

La Figure 4.2 définit les concepts et les relations suivantes :

- Le concept d'entité,
- Le concept de méta-entité,
- La relation Meta entre une entité et sa méta-entité,
- Le concept de modèle,
- Le concept de méta-modèle,
- La relation basedOn (conforme *à*) entre un modèle et son méta-modèle,
- La relation definedIn entre une entité et le modèle dans lequel elle est définie,
- La relation definedIn entre une méta-entité et le méta-modèle dans lequel elle est définie.

III– MODELES ET L'ESPACE TECHNIQUE

Cette section détaille la notion d'espace technique afin de structurer notre proposition par rapport aux différentes technologies utilisées. Ceci est en effet nécessaire afin de pouvoir traduire des données exprimées à l'aide d'une technique vers un format défini par une autre technique.

1 – DEFINITION

> *A technical space is a model management framework accompanied by a set of tools that operate on the models definable within the framework.* **[BEZ, 05]**

Cette définition est celle donnée dans **[BEZ, 05],** elle présente la notion d'espace technique (*ou Technical Space abrégé en TS*) comme un cadre qui fournit un système de représentation de modèles ainsi que des outils pour les manipuler. La figure 4.3 illustre ce concept.

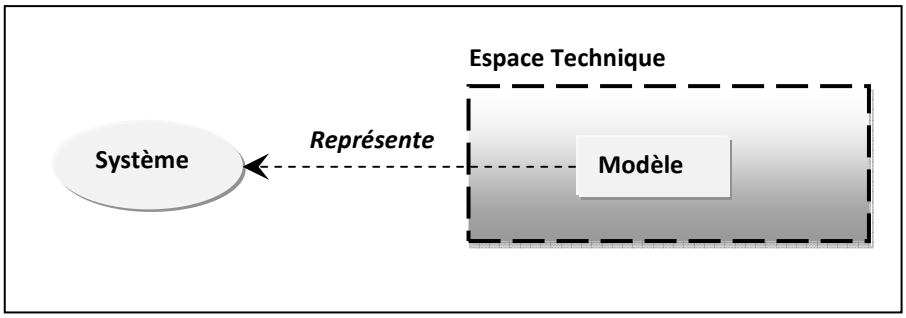


Figure 4.3 - Notion d'espace technique par rapport aux notions de modèle et de système [JOU, 06]

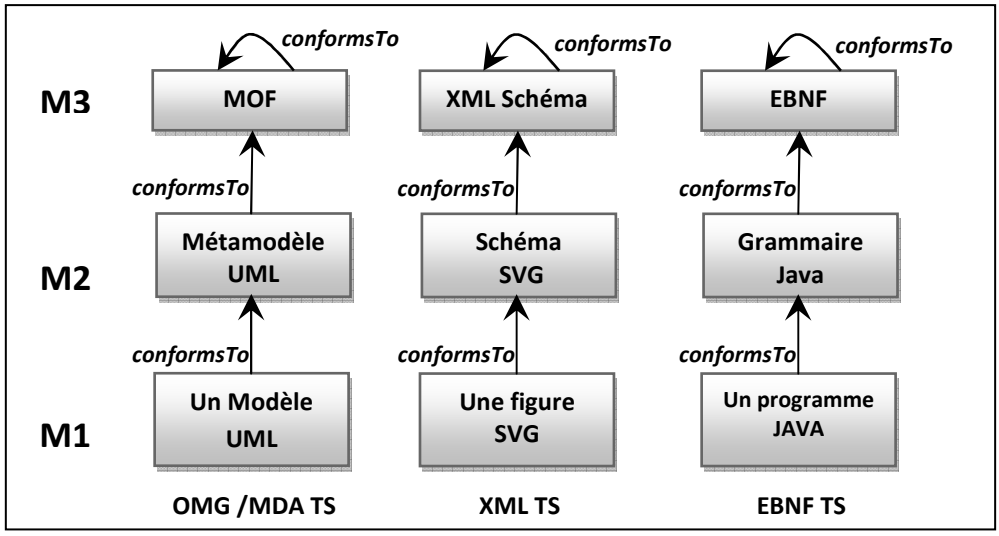
2- ESPACE TECHNIQUE ET LA META-MODELISATION

Dans **[BEZ, 05]** l'auteur suppose que plusieurs espaces techniques sont organisés comme celui de l'ingénierie des modèles (*figure 4.4*). En effet, si certains artéfacts manipulés dans un espace technique donné sont des modèles, alors d'autres artéfacts sont des méta-modèles, et il existe un unique méta-méta-modèle auquel tous les méta-modèles sont conformes.

En pratique, un méta-méta-modèle détermine l'*espace technique* de ses méta-modèles et de leurs modèles. Ainsi, Un espace technique est l'ensemble des outils et techniques issus d'une pyramide de méta-modèles dont le sommet est occupé par une famille de (méta) méta-modèles similaires **[FAV, 06]**. A titre d'exemple (*figure 4.4*):

- EBNF (*Extended Backus Naur Form*) appartient à l'espace technique des grammaires. Il permet de définir des grammaires qui jouent le rôle de méta-modèles et dont les mots jouent le rôle de modèles. La compilation permet de passer d'un niveau d'abstraction à l'autre.
- XML appartient à l'espace technique des documents pour lequel la DTD joue le rôle de méta-modèle pour des documents XML pouvant alors être considérés comme des modèles.
- UML appartient à l'espace technique des modèles représentant des systèmes logiciels.

De ce fait, le choix du méta-méta-modèle est déterminant, car il est la base de toute la chaîne de modélisation et définit les concepts sous-jacents à la représentation de tous les autres niveaux d'abstraction. Cette section présente différents méta-méta-modèles correspondant aux deux espaces techniques évoqués dans ce document : les modèles de procédés et les architectures logicielles.



4.4- Exemples d'espaces techniques à trois niveaux [JOU, 06].

2.1- PROCEDES LOGICIELS

Les modèles de procédés appartiennent à l'espace technique des modèles (*Modelware*). Les principaux langages de méta-modélisation existants dans ce domaine sont MOF, Ecore et Karmeta.

- **MOF**

  Le Meta Object Facility (MOF) **[MOF, 11]** est normalisé par OMG dans sa version courante 2.4. Il est le langage utilisé pour définir UML, CWM (*Common Warehouse Metamodel*), SPEM … Le MOF est un méta-méta-modèle ouvert et il doit pouvoir supporter un grand nombre d'utilisations, de même qu'il doit pouvoir aussi être supporté par un grand nombre d'applications. Le MOF s'auto-définit en termes de classes, d'associations, de paquetages et de types de données. Cela lui permet aussi de pouvoir définir d'autres méta-modèles.

- **Ecore**

  Le méta-méta-modèle Ecore **[VOJ, 10]**, ressemble fortement au méta-méta-modèle EMOF (Essential Meta Objet Facility), car il ne supporte que la notion de méta-classe sans méta-association. Ecore est légèrement différent du standard EMOF en ce qu'il est entièrement intégré à la plate-forme Eclipse.

  Ecore est le méta-méta-modèle défini dans le cadre du framework EMF (Eclipse Modeling Framework) **[EMF] [STE, 08]**. Il permet de définir des modèles de classes simples à l'aide de paquetages, classes, attributs, méthodes, paramètres, énumérations et types primitifs.

- **Karmeta**

  Kermeta **[KER]** est défini comme un langage de méta-modélisation exécutable : il permet de décrire des méta-modèles dont les modèles sont exécutables. Kermeta est basé sur le langage de méta-modélisation EMOF et est intégré à l'IDE Eclipse sous la forme d'un plugin. Il a été conçu comme un tissage entre un méta-modèle comportemental et EMOF. Le méta-modèle de Kermeta est donc composé de deux packages, Core et Behavior.

  Le premier correspond à EMOF et le second est une hiérarchie de méta-classes représentant des expressions impératives. Ces expressions sont utilisées pour décrire la sémantique comportementale des méta-modèles.

2.2- ARCHITECTURE LOGICIELLE

Très peu de travaux ont été conduits dans la technique de méta-modélisation architecturale. Cependant, quelques travaux correspondants à ce domaine peuvent être cités :

- **AML**

  AML (*Architecture Meta-Language*) **[WIL, 99]** est une première tentative de proposition d'un socle pour fournir aux ADL une solide base sémantique. C'est un langage très primitif, ayant des déclarations pour seulement trois concepts : les éléments, les types et les relations. Chacune de ces constructions peut être contrainte par des prédicats de la logique temporelle. Même si ce langage est destiné à être un méta-ADL, sa syntaxe est très détaillé, il est plus proche d'un ADL qu'à un méta-ADL. Toutefois, AML peut être considéré comme un ADL ayant la possibilité de définir de nouvelles entités.

- **MADL**

  MADL (*Meta Architectural Description Language*) **[SME, 05]** est une seconde tentative qui a introduit les concepts de méta-components, méta-connecteurs et les méta-architectures nécessaires pour manipuler et pour définir des concepts architecturaux (*structurelles et comportementales*). Ces méta-concepts sont organisés en trois packages : package méta-méta-architecture, package méta-architecture et le package architecture.

- **CoDex**

  CODeX **[MAR, 02]**, fournit un cadre de travail pour méta-modéliser des ADL. Le méta-méta-modèle précise la manière dont le MOF doit être utilisé pour structurer la définition de méta-modèles en respectant la séparation des préoccupations. Il est défini au travers d'un certain nombre de packages (*ArchitecturalBase, AnnotationBase, Integration, Mapping…*). Ce méta-méta-modèle représente le moyen à utiliser pour la définition des différents plans d'un ADL.

IV - TRANSFORMATIONS DE MODELE

La transformation des modèles est l'un des aspects les plus importants de l'IDM. La section suivante, présente cet aspect, ainsi que ses principes de base.

1- DEFINITIONS ET CONCEPTS FONDAMENTAUX

- Transformation

La définition suivante est celle proposée dans **[KLE, 03]**.

> *A Transformation is the automatic generation of a target model from a source model, according to a transformation definition* **[KLE, 03]**.

Cette définition, décrit la transformation de modèle comme étant la génération automatique d'un modèle cible depuis un modèle source, selon une définition de transformation.

✍ Mens et al. **[MEN, 06]** étendent cette définition en permettant la transformation depuis un ou plusieurs modèles sources vers un ou plusieurs modèles cibles.

> *Automatic generation of one or multiple target models from one or multiple source models, according to a transformation description* **[MEN, 06].**

- Définition de transformation

> *A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target Language* **[KLE, 03].**

Cette définition relate qu'une définition de transformation est un ensemble de règles de transformation qui décrivent comment un ou plusieurs modèles source peuvent être transformés dans un ou plusieurs modèles cible.

- Règles de transformation

> *A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language* **[KLE, 03].**

Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible.

Une règle de transformation contient deux parties : une partie gauche (*left-hand side «LHS»*) et une partie droite (*right-hand side «RHS»*). La LHS exprime des accès aux modèles sources, alors que la RHS indique les expansions (*création, modification, suppression*) dans les modèles cibles.

2- PRINCIPE DE LA TRANSFORMATION DE MODELES

Il existe différents types de transformations utilisées de différentes manières. Le processus général d'une transformation consiste à transformer un modèle Ma en un modèle Mb qui sont conformes à leurs méta-modèles respectifs MMa et MMb (*cf .figure 4.5*).

La transformation est *endogène* si MMa = MMb, sinon elle est *exogène* **[COM, 08]**. Une classification des différents types de transformation peut aussi être faite selon le changement de niveau d'abstraction. Une transformation *horizontale* produit un modèle du même niveau d'abstraction alors qu'une transformation verticale produit un modèle de niveau d'abstraction différent du modèle d'entrée **[COM, 08].** La figure 4.5 classe différentes catégories de transformation de l'IDM.

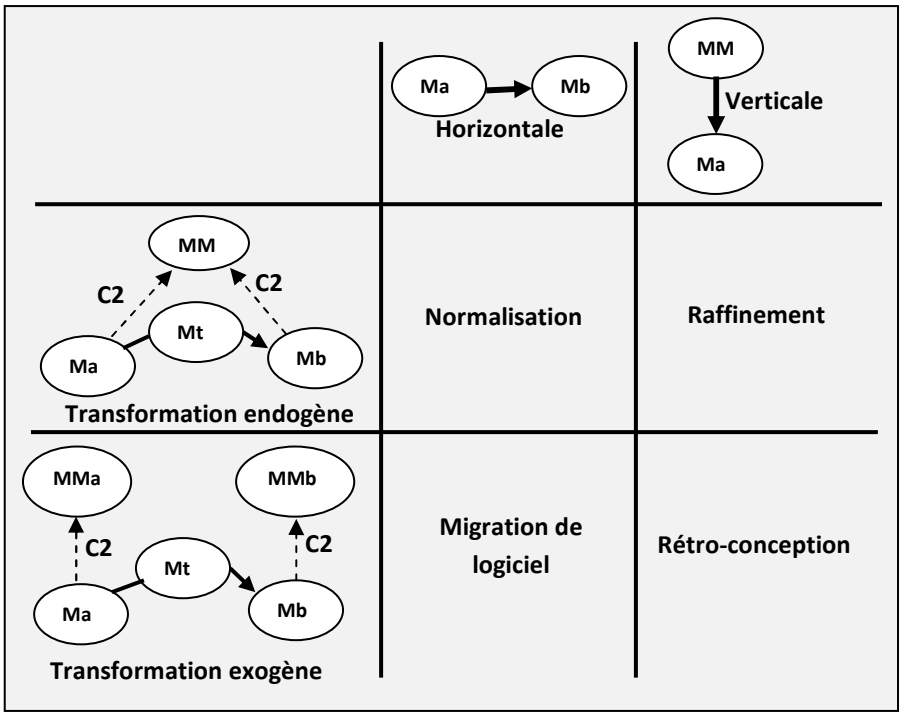


Figure 4.5 : Types de transformation et leurs principales utilisations [COM, 08].

3- PROCESSUS DE TRANSFORMATION

De manière générale, le processus de transformation est composé de trois étapes :

3.1- DEFINITION DES REGLES DE TRANSFORMATION

Il s'agit dans cette étape d'élaborer une mise en correspondance des concepts de modèle source à ceux de modèle cible. Dès lors, on a recours à la technique de méta-modélisation pour mettre en place une base de règles exhaustive et générique.

Les règles de transformation sont établies entre le méta-modèle source et le méta-modèle cible, c'est-à-dire entre l'ensemble des concepts du modèle source et celui du modèle cible. Le processus de transformation prend en entrée un ou plusieurs modèles conformes à des méta-modèles source et produit en sortie un ou plusieurs autres modèles conformes à un ou plusieurs méta-modèles cibles en utilisant les règles préalablement établies.

3.2- EXPRESSION DES REGLES DE TRANSFORMATION

Pour exprimer les règles de transformation un langage de spécification de règles est nécessaire.

Un langage de transformation peut être déclaratif, impératif ou hybride. Dans la programmation déclarative, on décrit d'une part les données du problème à traiter et d'autre part les contraintes sur ces données. Le programme s'exécute à partir de la situation courante décrite par les données tout en respectant les contraintes. Le langage déclaratif décrit ce qu'on devrait avoir à l'issue d'un certain nombre de données initiales. XSLT (*Extensible Stylesheet Language Transformation*) **[XSL]**, est un exemple de langage déclaratif de transformation. Par opposition à un programme déclaratif, un programme impératif décrit comment le résultat devrait être obtenu en imposant une suite d'actions que la machine doit effectuer. Un langage hybride regroupe à la fois les paradigmes de programmation déclarative et impérative.

3.3- EXECUTION DES REGLES DE TRANSFORMATION

Une fois spécifiées et exprimées, les règles requièrent un moteur d'exécution pour être exécutées. Ce moteur prend comme entrée le modèle et le méta-modèle source, le méta-modèle cible, ainsi que le modèle de transformation (*les règles de transformation écrites dans le langage de transformation, basées sur les correspondances entre les deux méta-modèles source et cible*) et son méta-modèle (*représentant la grammaire du langage de transformation*). Il produit en sortie le modèle cible.

V – SYNTHESE ET CONCLUSION

Ce chapitre a introduit l'idée fondatrice du développement dirigé par les modèles consistant à élever le niveau d'abstraction tout en cachant toute la complexité des technologies, des procédés de développement, etc.

Bien évidemment, du fait qu'un modèle augmente le niveau d'abstraction, des informations sont omises, ce qui rend le système plus simple et plus facile à comprendre. Ainsi, les

modèles permettent de capturer la complexité d'un système et de raisonner sur quelques-unes de ses propriétés, en ignorant certains détails de manière à ne considérer que ceux qui peuvent être pertinents, utiles et intéressants, vis-à-vis de l'utilisation qui en sera faite.

De ce fait, nous pensons que l'ingénierie des modèles permet de faire le lien entre les différents espaces techniques avec lesquels nous devons opérer (les *architectures logicielles et les composants de procédés logiciels*) et ceci, en fournissant des modèles abstraits qui apportent une simplification dans la compréhension et la manipulation de la connaissance des différentes technologies utilisées.

D'autre part, nous avons vu que les techniques de transformation de modèle, s'expriment directement entre les syntaxes abstraites des différents modèles, permettant ainsi, de se concentrer sur les concepts et d'aboutir, par la suite à une migration technologique cohérente et fiable.

Dans le chapitre suivant, on présente notre contribution qui utilise des principes de l'IDM pour représenter les modèles d'architecture logicielle et ceux des composants procédés, ainsi que le scénario de transition entre ces deux approches.

PARTIE II

CONTRIBUTION

CHAPITRE 5

CONCEPTION DE MAPL

I- INTRODUCTION

Pour faire face à la complexité des procédés, la communauté génie logiciel propose leur construction en faisant coopérer plusieurs équipes (*peut-être géographiquement distribuées*). En considérant que chaque équipe peut avoir ses propres environnements de développement (*formalismes de représentation, moteur procédé, etc.*), les modèles fournis par ces équipes seront hétérogènes. Et malheureusement, les formalismes de représentation actuels liés à l'infrastructure qui les supporte ne facilitent ni l'interopérabilité, ni la réutilisation des modèles de procédé.

Comme nous l'avons présenté dans l'état de l'art, les travaux cherchant à résoudre le problème de l'interopérabilité et de la réutilisation sont nombreux. Cependant, force est de reconnaitre que ces travaux offrent des solutions souvent liées au formalisme de représentation. Tandis que l'hétérogénéité des concepts reste un domaine peu exploré.

Un autre manque des ECP étudiés est l'absence d'un modèle qui gouverne les interactions entre les différentes entités du procédé, qui se limitent généralement à des échanges de messages ou d'événements, sans expliciter une définition des mécanismes d'interactions utilisés.

Ainsi, afin de répondre aux lacunes des ECP (Chapitre 1, section II.3), ce travail tente de fournir :

- Des mécanismes qui permettent la réutilisation et l'interopérabilité entre des modèles de procédé hétérogènes au sein d'un environnement commun.
- Des moyens de description explicites des mécanismes d'interaction et des protocoles de communication utilisés.

La solution proposée considère un modèle de procédé particulier comme la composition d'un ensemble hétérogène de composants de procédé existants, testés, utilisés et dont l'efficacité a déjà été prouvée. Chaque composant de procédé est donc considéré comme une vue partielle du procédé global.

Nous considérons la conception d'un procédé comme l'élaboration d'un plan de construction d'une architecture logicielle, et nous adoptons les techniques de cette approche pour modéliser les interactions entre les différents éléments qui composent un procédé initialement décrit avec un PML, sous forme architecturale à l'aide d'un ADL. Assurant ainsi, la séparation entre les entités logicielles et leurs mises en œuvre, ce qui permet de concevoir la structure de l'application indépendamment de toute plate-forme d'exécution.

En conséquence, MAPL (Modèle Architectural des Procédés Logiciels) se place à un niveau d'abstraction supérieur. Il définit les concepts de base d'une description architecturale, sur lesquels s'accordent l'ensemble des travaux portant sur les ADL, à savoir : *Composant*, *Connecteur*, *Configuration* et *Interface*. Définissant ainsi, un modèle architectural qui peut être utilisé par différents ADL.

Au cours de ce chapitre, nous présentons les concepts de base de notre proposition. Ensuite, nous détaillons nos principales étapes de conception. Finalement, nous exposons quelques éléments fournis par la solution.

II- PRESENTATION DE MAPL

Conformément à la définition basique des architecture logicielles, dites de trois C (*Chapitre 3, section IV*), un modèle de procédé se définit sous MAPL par :

1- COMPOSANT

Un composant donne une définition abstraite aux entités logicielles, il est considéré comme une boite noire ayant une partie externe et un corps. La partie externe est le moyen à travers lequel l'entité peut échanger les données avec l'extérieur. Le corps écrit l'implémentation et la structure interne de l'entité.

La figure 5.1 donne une idée sur la représentation graphique du composant procédé sous MAPL. La partie externe est constituée des propriétés globales, (*propriétés fonctionnelles ou non fonctionnelles*) et des points d'interaction (*l'interface*). Le corps est représenté par sa partie interne.

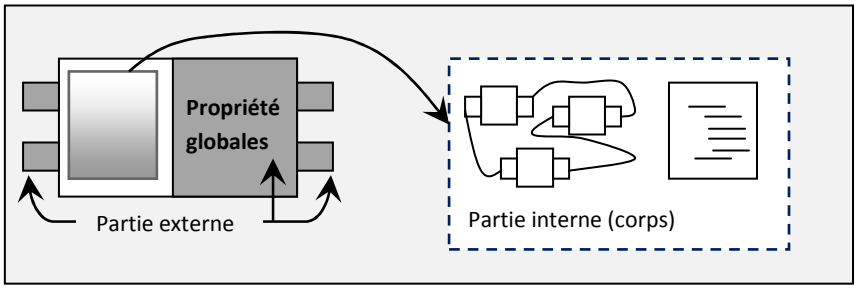


Figure 5.1 - Structure du composant MAPL

2- CONNECTEUR

Le connecteur correspond à un élément d'architecture logicielle qui représente un moyen d'interaction entre les composants. Il permet également d'assembler des composants en utilisant leurs interfaces (*fournies et requises*).
Plusieurs types de connecteurs peuvent existés, qui dépendent des types de relations qui unissent les différentes entités. MAPL définit différents types de connecteurs de bases, qui seront détaillés plus loin (*section V*).

3- INTERFACE

Spécifient des points de communication qui permettent à un composant/connecteur d'interagir avec l'environnement. On a définit deux types d'interfaces : *Provided_interface* et *Required_interface*.

- Provided_interface: Représente les interfaces fournies, qui décrivent les services proposés par le composant.
- Required_interface : Représente les interfaces requises, qui décrivent les services que les autres composants doivent fournir pour le bon fonctionnement du composant.

4- CONTRAINTES DE CONFIGURATION

Les contraintes définissent les limites d'utilisation d'un composant/connecteur et les dépendances intra-composants. Une contrainte est une propriété devant être obligatoirement vérifiée par un procédé ou une de ses parties. Si celle-ci est violée, le procédé est considéré comme un procédé incohérent.
Les règles de cohérence sont modélisées par des propriétés. De telles règles de cohérence permettent de vérifier des propriétés structurelles telles que :

- Tout composant a au moins une interface offerte.
- Toute interface requise est reliée à une interface offerte.

III- DEMARCHE DE CONCEPTION DE MAPL

Nous présentons, dans cette section, la démarche de conception de MAPL, qui se base sur trois (03) étapes essentielles illustrées dans la figure 5.2 :

- **L'unification** : comprend principalement la transformation des modèles de procédés en des architectures logicielles unifiées par un modèle architectural commun (*l'ADL pivot*).
- **Assemblage** : Une fois les modèles de procédé unifiés, l'architecte peut les grouper en un modèle de procédé global. En conséquence, plusieurs types de connecteurs ont été proposés afin de simplifier l'assemblage, ils dépendent principalement des types de relations qui unissent les différentes entités d'un procédé.

- **Raffinement et implémentation** : Nous rappelons qu'une architecture logicielle est la description abstraite d'un système logiciel. Le raffinement d'une architecture logicielle permet d'obtenir une description plus concrète du système, c'est-à-dire une architecture de plus bas niveau pouvant être proche d'une implémentation. Le raffinement d'architectures logicielles constitue une base de développement des systèmes logiciels corrects. Pour cela, le raffinement d'une architecture logicielle doit garantir que l'architecture logicielle concrète est valide au regard de l'architecture logicielle initiale **[MEG, 04].**

✍ *Vu les objectifs visés par notre travail, on ne s'intéresse pas actuellement aux problématiques de niveau code et du raffinement d'une architecture logicielle. Cependant, ceci fait l'objet d'une des plus importantes perspectives de notre travail.*

IV- UNIFICATION

C'est l'étape la plus importante de la démarche, dans la quelle, des modèles de procédé issus d'environnements hétérogènes, vont être unifiés sous un modèle architectural. Permettant ainsi, leur intégration et leur interaction au sein d'un modèle commun.
Dans ce qui suit, nous allons définir dans un premier lieu le concept de l'ADL pivot, puis nous détaillons les étapes d'unification.

1- ADL PIVOT

C'est le modèle commun dans le quel, on va migrer tous les modèles de procédé afin de masquer leur hétérogénéité. Il doit aussi, permettre leur réutilisation avec d'autres ADL.
Ainsi, nous avons définit certain critères pour le choix de l'ADL pivot, qui se résume principalement :

- Bien connu et largement utilisé (*un modèle standard si il existe*),
- Représente les concepts communs aux ADL,
- Combinant les fonctionnalités et les constructions d'une grande variété d'ADL,
- Fournit des moyens de raffinement, d'analyse et d'évaluation, tel que la vérification de la cohérence d'un système, …
- ADL extensible, qui peut être étendu vers d'autre modèle.

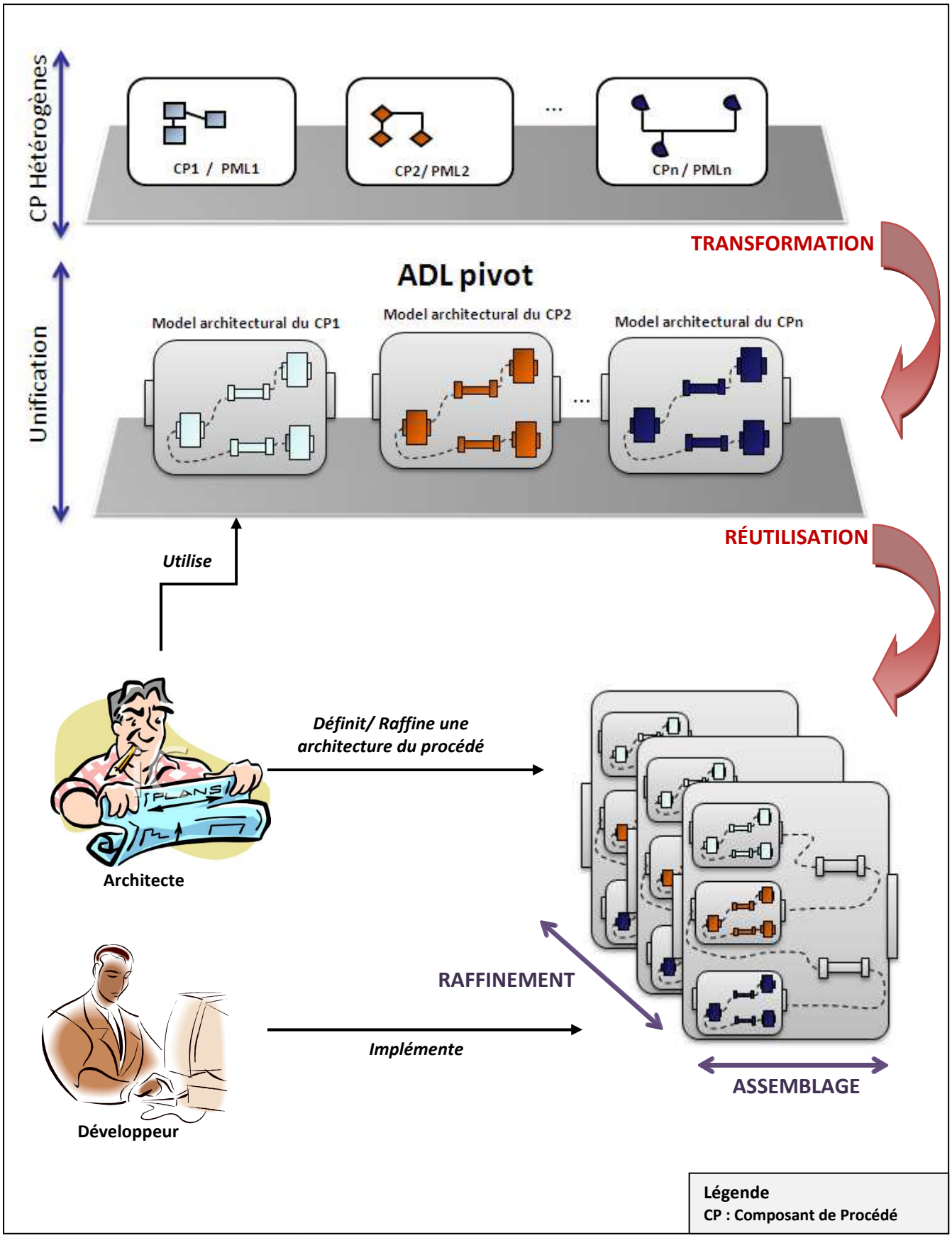


Figure 5.2- Architecture générale de la démarche proposée.

2 -TRANSFORMATION

La procédure de transformation commence par établir les relations d'associations entre un méta-méta-modèle de procédé et celui des architectures logicielles, en se basant sur les techniques de méta-modélisation. Définissant ainsi, un méta-modèle de transformation.

Par la suite, le méta-modèle de transformation obtenu, sera exploité pour établir les mises en correspondance de l'ensemble des méta-modèles sources conforme au méta-méta-modèle de procédé avec l'ensemble des méta-modèles cibles conforme au méta-méta-modèle architectural. Des règles de transformations seront ainsi, spécifiées et utilisées pour effectuer les transformations requises dans le niveau concret : Spécification dans un niveau plus abstrait et transformation dans le niveau plus bas. La procédure de transformation sera donc, appliquée sur différents niveaux d'abstractions (*figure 5.3*).

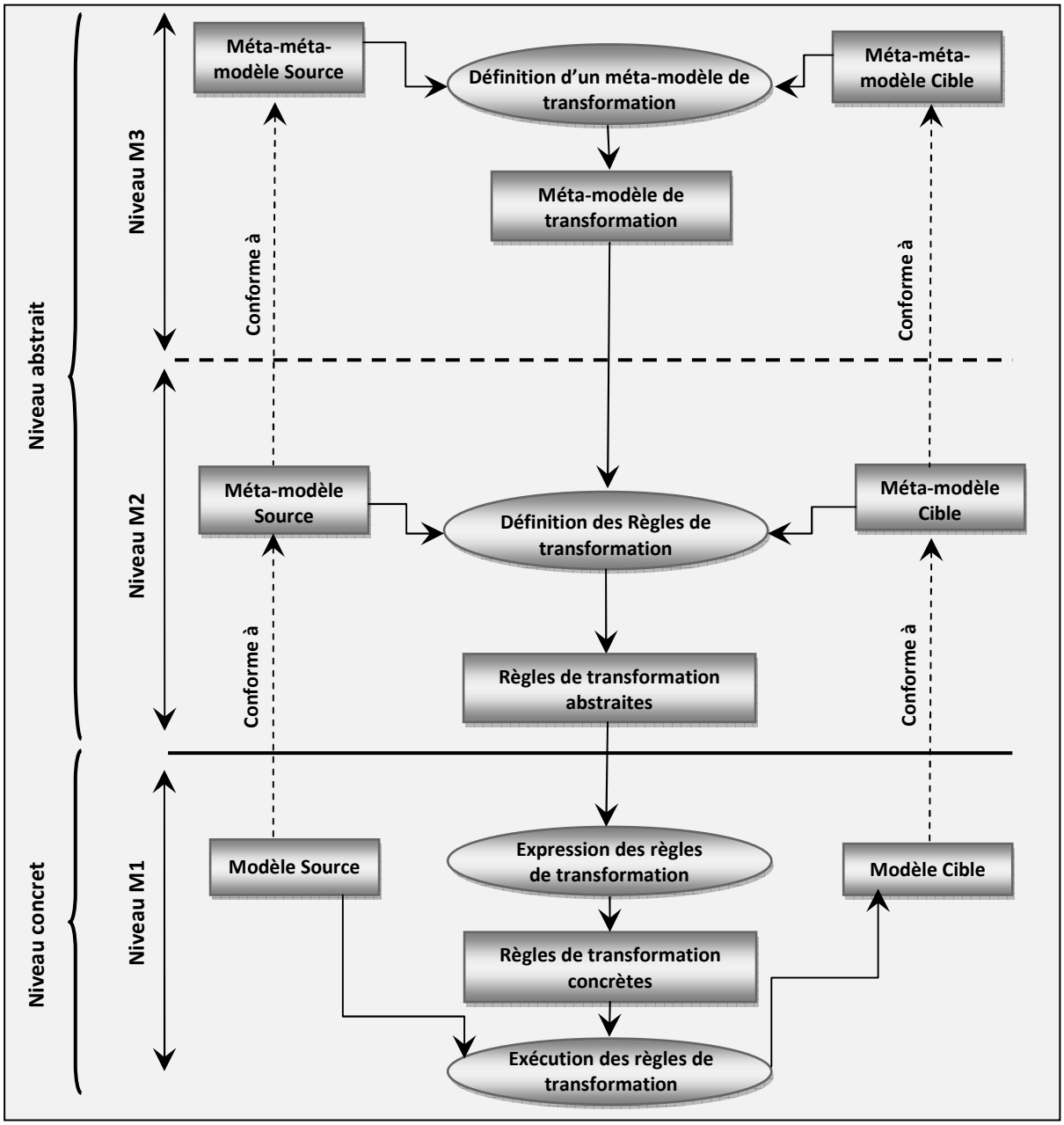


Figure 5.3 - La structure de transformation sur différents niveau d'abstraction

1.2- NIVEAU META-META-MODELE (NIVEAU M3)

C'est la première étape de la transformation qui consiste à établir les correspondances entre deux méta-méta-modèles correspondants aux deux paradigmes (*les procédés et les architectures*) décrivant ainsi un méta-modèle de transformation. Cette étape se base essentiellement sur les techniques de méta-modélisation.

1.3 - NIVEAU META-MODELE (NIVEAU M2)

Dans ce niveau, la mise en correspondance s'effectue entre un méta-modèle de procédé spécifique et le méta-modèle architectural unifié (l'ADL pivot), et ceci en exploitant le méta-modèle de transformation crée précédemment.

La transformation dans ce niveau passe par quatre (04) étapes principales.

Pour chaque entité du méta-modèle de procédé (*figure 5.4*) :

1. On identifie la méta-entité correspondante.
2. La méta-entité identifiée sera comparée aux méta-entités du modèle cible correspondantes dans le méta-modèle de transformation.
3. Chaque méta-entité cible identifiée dans l'étape précédente sera instancie, on obtiendra ainsi, une liste d'entités architecturales.
4. On examine la liste des entités architecturales résultante de la troisième étape précédente, et on choisit l'entité la plus adéquate.

Cette procédure sera appliquée pour toutes les entités du méta-modèle de procédé, et on aboutira ainsi, à un modèle de transformation du méta-modèle de procédé vers l'ADL Pivot, conforme à notre méta-modèle de transformation.

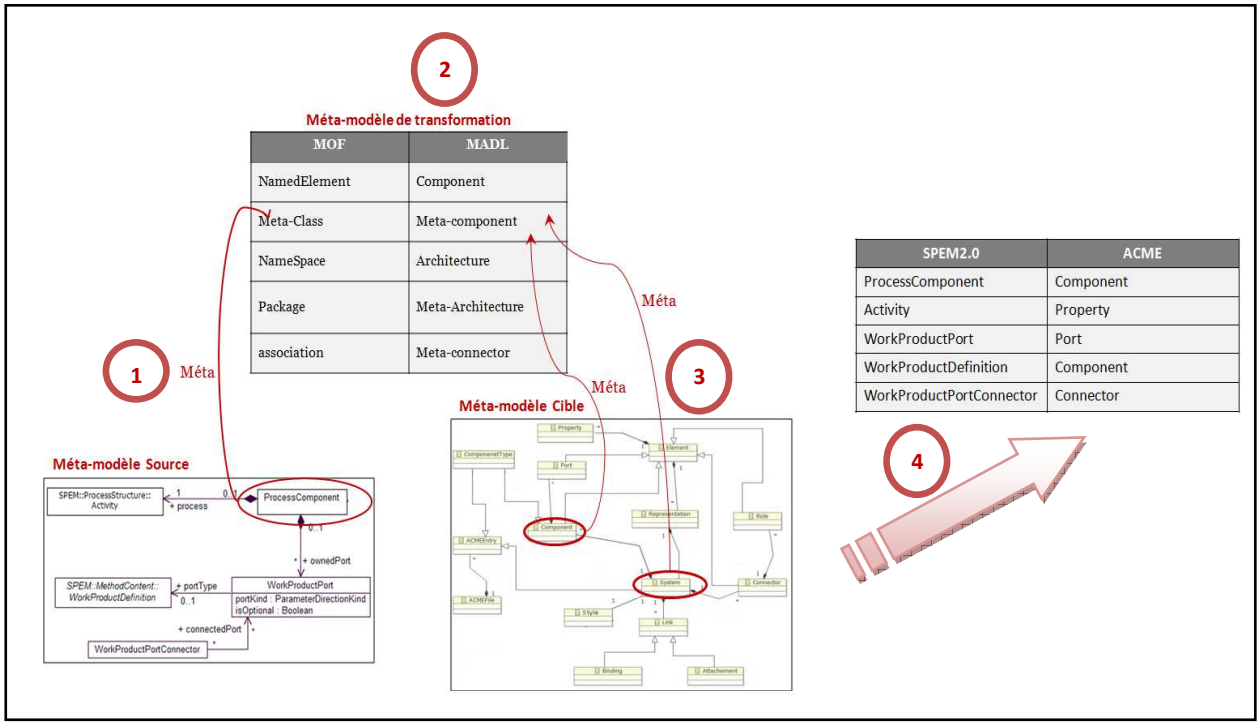


Figure 5.4- Les quatre principales étapes de la transformation.

1.3- NIVEAU MODELE (NIVEAU M1)

La principale tâche de ce niveau, est d'écrire et d'exécuter les règles qui vont créer les éléments du modèle cible à partir des éléments du modèle source.

Chaque lien de correspondance entre les éléments des modèles source et cible se traduit par une règle de transformation. Une fois vérifiée, l'exécution des règles de transformation peut commencer. En associant pour chaque élément reconnu du modèle de procédé, un (*ou plusieurs*) élément créé du modèle architectural.

Ensuite, on raffine petit à petit le modèle architectural obtenu, en modèle de plus en plus concret jusqu'à l'obtention du procédé final. Cette étape sera détaillée dans le chapitre 06.

V – TRANSFORMATION AU NIVEAU MÉTA-MÉTA

L'objectif de cette transformation est d'aboutir à un méta-modèle de transformation, qui à partir d'un méta-modèle conforme au méta-méta-modèle source, génère des règles de transformation vers un méta-modèle conforme au méta-méta-modèle cible. Dans le cadre de ce mémoire on a choisi de travailler avec :

Méta-méta-modèle MOF : Parmi les langages de méta-modélisation existants, nous avons retenu MOF2.0 pour les travaux présentés dans ce document. Les deux raisons de ce choix sont, d'une part, le fait que ce langage est normalisé et d'autre part le fait qu'il est utilisé dans plusieurs technologies standardisées par l'OMG comme : UML, SPEM, XMI…

Méta-méta-architecture MADL : Afin de pouvoir aligner une méta-méta-architecture avec le MOF, nous optons pour une méta-méta-architecture minimale, réflexible et générique qui permet de définir les concepts de base d'une méta-architecture. Ainsi nous avons choisis MADL. En effet, MADL définit trois éléments de base qui permettent de définir les différentes méta-architectures (*composant, connecteur et configuration*) et il est conforme à lui-même.

1- MAPPING MOF-MADL

Il consiste à établir les liens de correspondances entre MOF et MADL dans le but de fournir un méta-modèle de transformation. Pour ce faire, on s'est basé sur les techniques de méta-modélisation, où on a identifié et mis en correspondances pour chaque méta-méta-modèle les éléments essentiels pour décrire le noyau réflexif minimal (*Chapitre 4 section II.3*).
Les autres éléments peuvent être obtenus par simple déduction, ainsi, ils ne seront pas abordés dans ce chapitre.

1.1- MADL

MADL **[SME, 05]** est un méta-méta-modèle minimal et générique qui introduit les concepts de bases nécessaires à la manipulation et à la définition des entités architecturales. Il est

organisé en trois packages : package *Meta-meta-architecture*, package *Meta-architecture* et le package *Architecture* (*figure 5.5*).

- **Package Meta-meta-architecture :** S'auto-définit, il est composé de deux packages, le package *Meta-architecture* et le package *Architecture*. il englobe tous les concepts nécessaires à la définition des architectures et des méta-architectures.
- **Package Meta-architecture :** Une instance de la méta-méta-architecture, qui permet de définir et de classer des architectures. Il se compose des éléments suivants :
  - **Meta-component** : Il s'agit d'un élément méta-architectural qui classe et définit des lieux de traitement ou de stockage au niveau méta-architectural. Il hérite de la classe *Component*, et c'est une instance de lui même.
  - **Meta-connector** : C'est un élément méta-architectural qui classe et définit des interactions entre composants au niveau méta-architectural. Il hérite de la classe *Component*.
  - **Meta-interface** : Il s'agit d'un élément méta-architectural qui classe et définit des interfaces pour les composants, les connecteurs et les architectures au niveau méta-architectural.

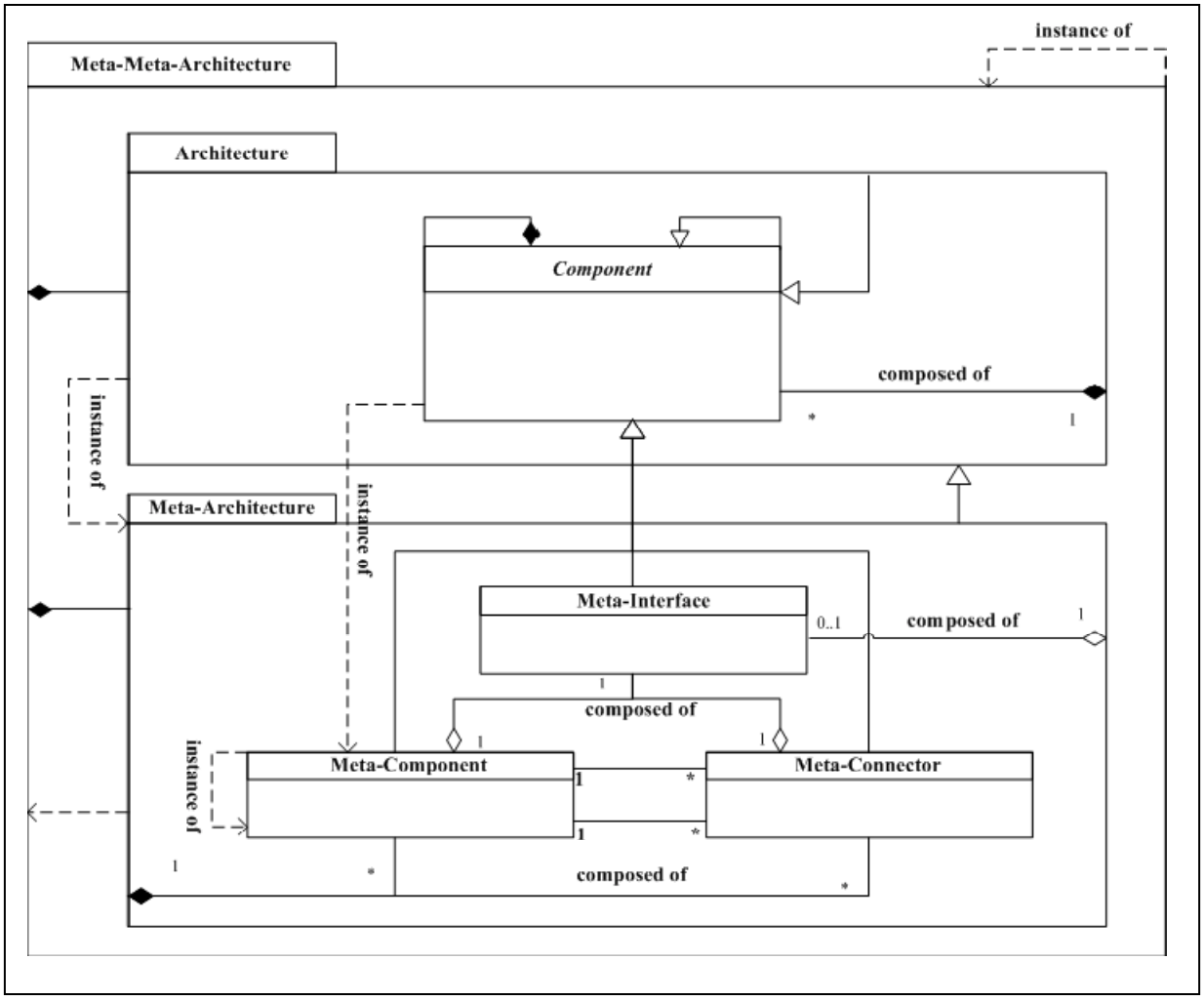


Figure 5.5- Le méta-méta-modèle MADL [SME, 05]

- **Package Architecture** : Une instance de la méta-architecture qui hérite de la classe *Component*, et qui respecte ainsi, le principe de MADL '' tout est un composant '' .Par conséquent, les architectures au niveau méta-architectural se comportent comme des composants, c'est-à-dire, peuvent interagir les uns avec les autres, avoir une relation de composition et d'héritage. Il se compose de le classe Component.
    - **Component** *:* Une instance de méta-composant, c'est une classe abstraite qui classe et définit tous les éléments MADL. En conséquence, tous les éléments de MADL héritent de component.

1.2- NOYAU REFLEXIF DE MADL

MADL respect parfaitement le principe du noyau réflexif définit par LEMESLE **[LEM, 00]** : L'architecture est une instance de méta-architecture, composant est une instance de méta-composant, et un méta-composant est une instance de lui même, de sorte que tous les rapports d'instanciation se termine dans le méta-composant. (*figure 5.6*).

D'autre part, on voit clairement que la relation *defineIn* est modélisée entre la meta-architecture et meta-component, et entre l'architecture et le component.

Ainsi, en comparant le noyau réflexif de MADL et celui définit par LEMESLE, on a pu déduire les mises en correspondance décrites dans le tableau 5.1 :

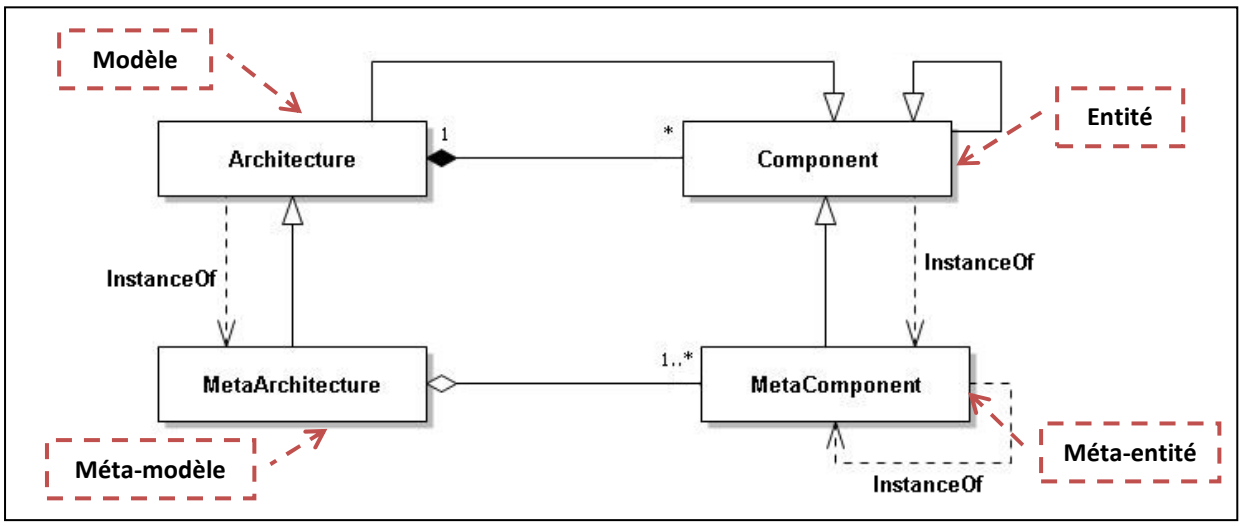


Figure 5.6 - Le noyau réflexif de MADL

LEMESLE	*MADL*
Entité	*Component*
Méta-entité	*Meta-component*
Modèle	*Architecture*
Méta-modèle (méta-méta-modèle)	*Meta-Architecture*
Méta-relation	*Meta-connector*

Tableau 5.1 : Mapping LEMESLE-MADL

1.3- MOF2.0

MOF (*Meta Object Facility*) dans sa version actuelle 2.4.1 **[MOF, 11]** a pour but de définir un langage de modélisation unique pour représenter des modèles et méta-modèles. MOF est utilisé pour sa propre définition, et aussi utilisé dans plusieurs technologies standardisées par OMG comme UML, CWM, XMI. Contrairement aux versions précédentes, et pour des raisons d'efficacité, il a été convenu qu'un méta-modèle instance de MOF2.0 (et les versions postérieur) pouvait ou non contenir des méta-associations. Voilà pourquoi, pour faire la différence entre ces deux types de méta-modèles, MOF2.0 a été décomposé en deux parties : EMOF (*Essential MOF*) pour l'élaboration de méta-modèles sans association, et CMOF (*Complete MOF*) pour celle de méta-modèles avec association.
MOF2.0 intègre le méta-modèle UML2.0 Infrastructure **[UMLb, 11]**. EMOF intègre le package Basic de l'infrastructure et CMOF le package Constructs.

Dans ce mémoire, on a adopté CMOF (*figure 5.7*) du fait qu'il englobe l'ensemble du modèle MOF. Ainsi, le CMOF, et MOF vont être utilisés indifféremment dans le reste de ce mémoire.

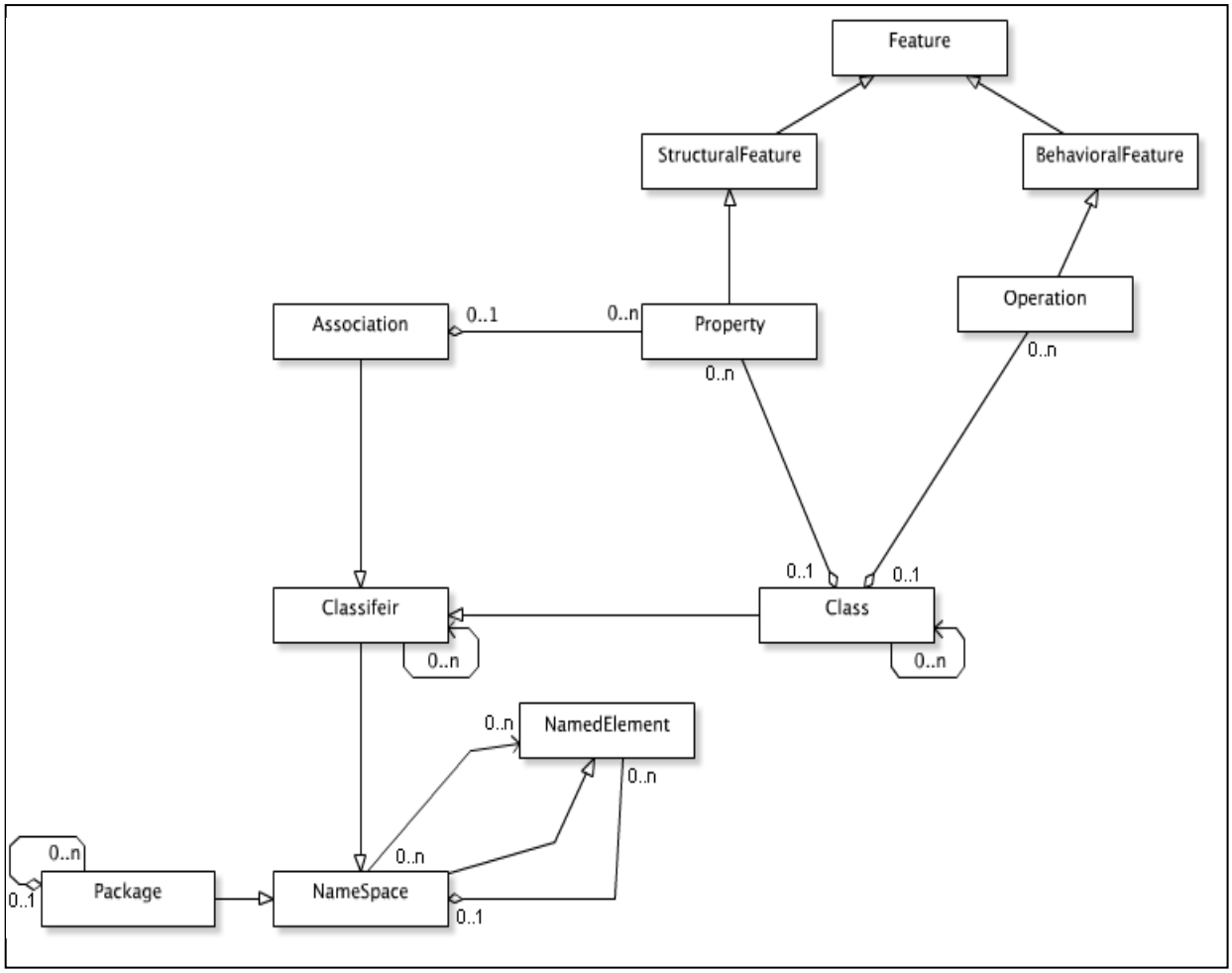


Figure 5.7 - Un extrait du méta-méta-modèle MOF [MOF, 11]

Dans ce qui suit, on se contente de présenter que les éléments MOF, qui constitue le noyau réflexif :

- *NamedElement* : Un élément nommé représente des éléments qui peuvent avoir un nom. Le nom est utilisé pour l'identification de l'élément nommé au sein de l'espace de nom dans lequel il est défini. *NamedElement* est une méta-classe abstraite.
- *NameSpace* : Un espace de nom est un élément nommé qui peut posséder d'autres éléments nommés. Chaque élément nommé peut être la propriété d'au plus un espace de nom. Espace de nom est une méta-classe abstraite.
- *Class* : Une classe décrit un ensemble d'objets qui partagent les mêmes spécifications, caractéristiques, contraintes, et sémantique.
- *Package* : Un package fournit un moyen de regrouper des types et d'autres package, ce qui peut être utile pour comprendre et gérer un modèle.

1.4- NOYAU REFLEXIF DU MOF

Une méta-entité MOF (*figure 5.8*) est définie par une entité de type Class et un méta-modèle est généralement constitué d'une entité de type Package. La relation entre une classe et son paquetage (*l'équivalent de la relation definedIn entre une méta-entité et son méta-modèle*) est définie par l'association MOF appelée *contains*. Ces trois éléments (*la méta-entité Class, le méta-modèle Package et l'association Contains*) sont les seuls éléments du MOF qui ont été mis en correspondance avec les entités du noyau réflexif proposé par LEMESLE.

Cependant, le MOF définit, en plus de ce méta-méta-modèle, un paquetage appelé ''*Reflective*'' qui inclut une classe ''*RefObject*'', considéré comme l'entité du noyau, et qui définit des méthodes équivalentes aux relations meta entre une entité et sa méta-entité.
D'un autre coté, comme l'entité racine du MOF, NamedElement, est sous-type de RefObject, elle est considérée comme l'entité dans le noyau réflexif de MOF.

Pour la notion de modèle, on a considéré que la classe NameSpace, était la plus adéquate, en effet, un espace de nom contient des éléments nommé, (*l'équivalent de la relation definedIn entre une entité et son modèle*), et un NameSpace est un sous-type de NamedElement (*un modèle est une entité*).

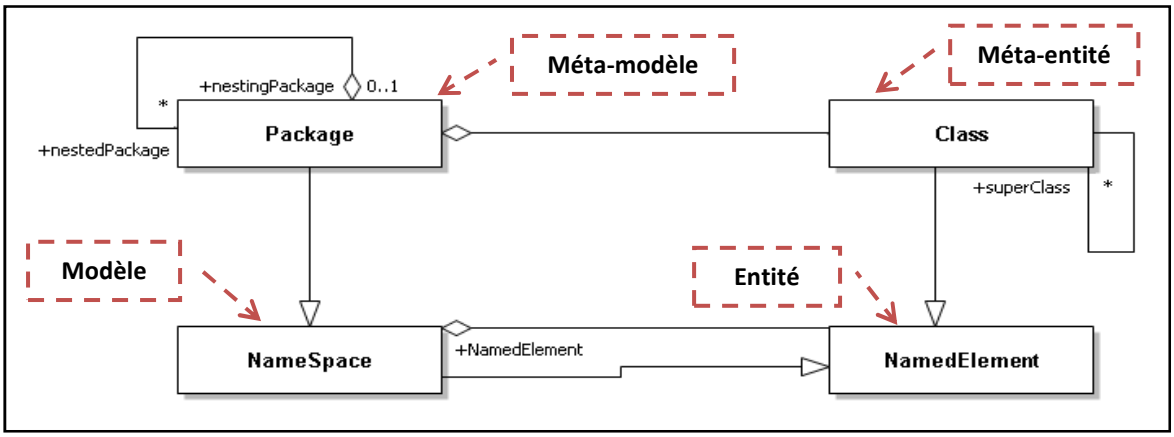


Figure 5.8 - Le noyau réflexif du MOF

Malgré l'absence de la relation entre un modèle et son méta-modèle. Le noyau identifié est suffisant pour nous permettre d'établir les mises en correspondances exposées dans le tableau 5.2 :

LEMESLE	MOF
Entité	NamedElement
Méta-entité	Class
Modèle	NameSpace
Méta-modèle (*méta-méta-modèle*)	Package
Méta-relation	association

Tableau 5.2 : Mapping LEMESLE-MOF

1.5- MISE EN CORRESPONDANCE MOF-MADL

La mise en correspondance MOF-MADL, décrite sur la figure 5.9, se déduit directement de l'étude menée précédemment, sur les noyaux réflexifs de MADL et MOF.
On définit ainsi, notre méta-modèle de transformation.

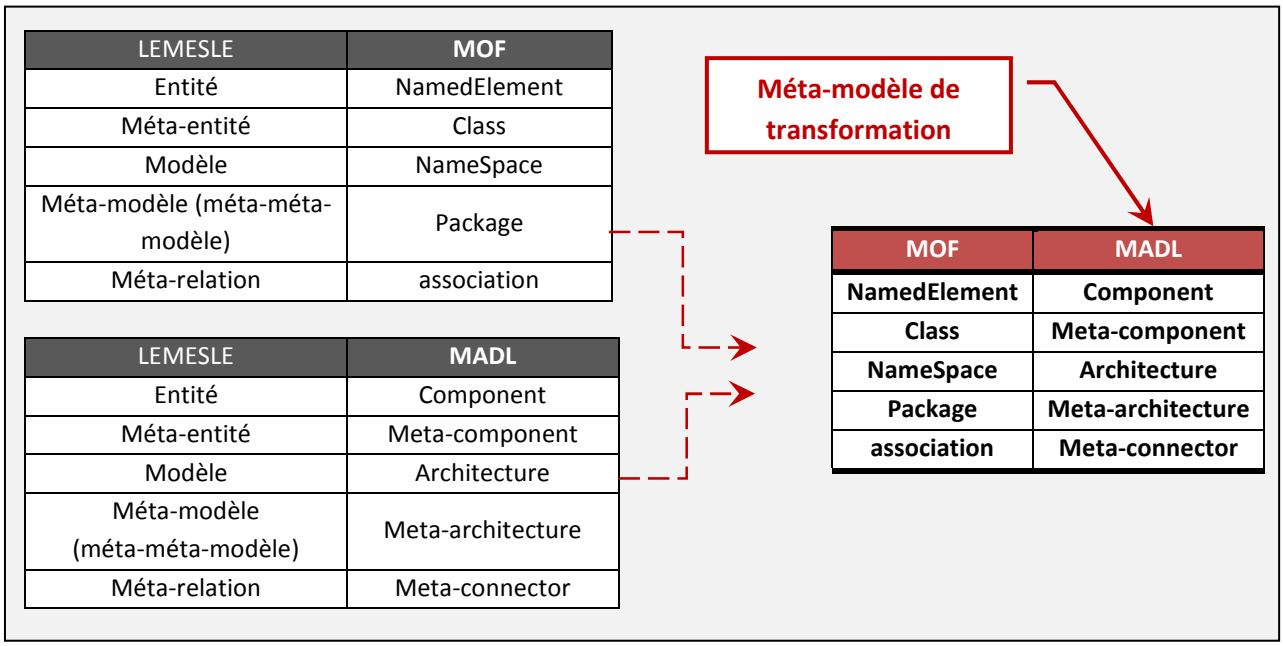


Figure 5.9- Mapping MADL-MOF

VI- TRANSFORMATION AU NIVEAU MÉTA

Dans ce qui suit, nous illustrons les quatre (04) principales étapes de la transformation du niveau méta, décrites dans la section IV.1.3, à travers un méta-modèle architectural qui représente l'ADL pivot et deux différents méta-modèles de procédé.

1- PRESENTATION DE L'ADL PIVOT

Le choix pour l'ADL pivot s'est porté sur ACME pour les raisons suivantes :

- ACME est un ADL largement utilisé, supporté par divers plateformes et outils, et sur le quel, divers travaux se basent,
- ACME définit la majorité des concepts utilisés dans une modélisation structurelle d'architectures logicielles et représente explicitement leurs caractéristiques et les relations qui existent entre eux,
- ACME a été proposé dans le but principal de fournir un format d'échange pour les outils et environnements du développement architectural. Ainsi, le langage fournit l'intégration de plusieurs outils développés pour supporter des ADL différents, fournissant un mécanisme pour l'échange d'informations sur l'architecture. Ceci est bénéfique du fait qu'on peut pivoter notre architecture vers divers autres ADL.
- ACME confère aux connecteurs une importance et un niveau de granularité et de réutilisation semblable à celui des composants. Un connecteur ACME est défini explicitement et considéré comme une entité de première classe. Nous pouvons ainsi définir des extensions du concept connecteur d'ACME pour décrire les connecteurs MAPL proposés (section V),
- ACME dispose d'un mécanisme d'annotation pour l'intégration d'information spécifique ne concernant pas la structure.

1.1- META-MODELE ACME

En se basant sur le BNF d'ACME **[ACMa]** on a pu exprimer en UML, un extrait du méta-modèle ACME, qui se résume dans la figure 5.10.

Une spécification d'ACME est représentée par des types de base: composant, connecteur, attachement, représentation, propriété, port, rôle, et de la carte de représentation (*rep-map*) (*figure 5.10*).

- **Composant :** Le composant représente l'unité de traitement ou de donnée d'une application. Il est spécifié par une interface composée de plusieurs types de ports.
- **Port :** Chaque type de port identifie un point d'interaction entre le composant et son environnement.
- **Connecteur :** Le connecteur représente l'interaction entre composants. Il s'agit d'un médiateur de communication qui coordonne les connexions entre composants.
- **Rôle :** Chaque type de rôle d'un connecteur définit un participant à une interaction.
- **Système :** Le système représente la configuration d'une application, c'est-à-dire l'assemblage structural entre les composants et les connecteurs.

- **Représentation :** La représentation et la carte de représentation permettent à ACME 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.
- **Carte de représentation :** 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.
- **Propriétés :** Une propriété a un nom qui l'identifie, un type optionnel et une valeur. Chaque élément du design décrit ci-dessus peut avoir une ou plusieurs propriétés.

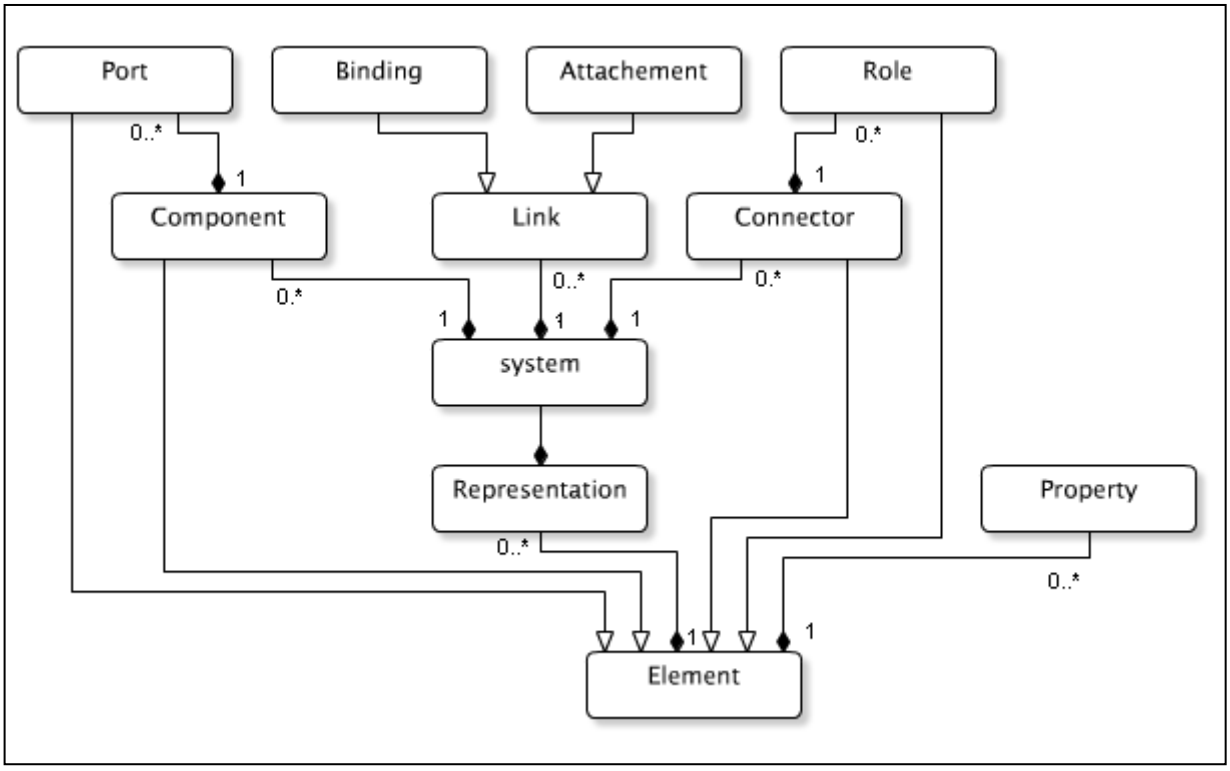


Figure 5.10- Extrait du méta-modèle ACME

1.2- INSTANCIATION DE MADL PAR ACME

De l'étude menée sur le méta-modèle d'ACME, on a pu conclure les déductions suivantes :

- Le système représente une configuration qui assemble des composants et des connecteurs, donc c'est une instance de méta-composant.
- Les composants représentent une unité de traitement ou contenant des données, ainsi, ils sont des instances de méta-composant.
- Les connecteurs, les attachements et les binding représentent les interactions possibles entre les types de composants, en conséquence, ils sont des instances de méta-connecteurs.
- Les interfaces telles que les ports et les rôles sont des instances de la méta-interface.

L'instanciation de MADL par ACME se résume dans le tableau 5.3 :

ACME	MADL
Composant	Méta-composant
Connecteur	Méta-connecteur
Système	Méta-composant
Attachement	Méta-connecteur
Binding	Méta-connecteur
Port	Méta-interface
Rôle	Méta-interface
Propriété	Méta-interface

Tableau5.3 - instanciation de MADL par ACME [SME, 05]

2- PRÉSENTATION DES MÉTA-MODÈLES DE PROCÉDÉ

Afin de concrétiser notre solution nous avons choisis deux méta-modèles de procédé hétérogènes et dans l'assemblage sous un procédé globale, n'était pas envisageable.

Ainsi, nous avons fait le choix d'utiliser :

- SPEM 2.0 principalement, du-fait, qu'il est :
  - Le standard de l'OMG,
  - Largement utilisé,
  - Défini sous la forme d'un méta-modèle MOF,
  - Dédié à l'ingénierie des procédés et intégrant l'ensemble de ses aspects.

- RHODES, particulièrement pour le fait qu'il est :
  - Conforme au MOF,
  - Définit clairement le concept du composant de procédé,
  - Méta-modèle homogène au niveau syntaxique et sémantique.

2.1- META-MODELE SPEM2.0

SPEM 2.0 propose un méta-modèle générique et flexible qui couvre divers types de procédés. Il est composé de sept paquetages liés avec le mécanisme « merge » (*chapitre2, section II.4*). Chaque paquetage traite un aspect particulier.

Dans le cadre de nos travaux, on s'est intéressé aux mécanismes qui gèrent et réutilisent des bibliothèques de procédé, et plus particulièrement au concept de composant de procédé. En effet, SPEM 2.0 propose la notion de composant de procédé (*ProcessComponent*) remplaçable et réutilisable selon le principe de l'encapsulation (*composant boite noire*).

La figure 5.11 expose le méta-modèle de composant de procédé SPEM2.0 **[SPE, 08],** une description plus détaillée de ses composants est donnée dans le tableau 5.4.

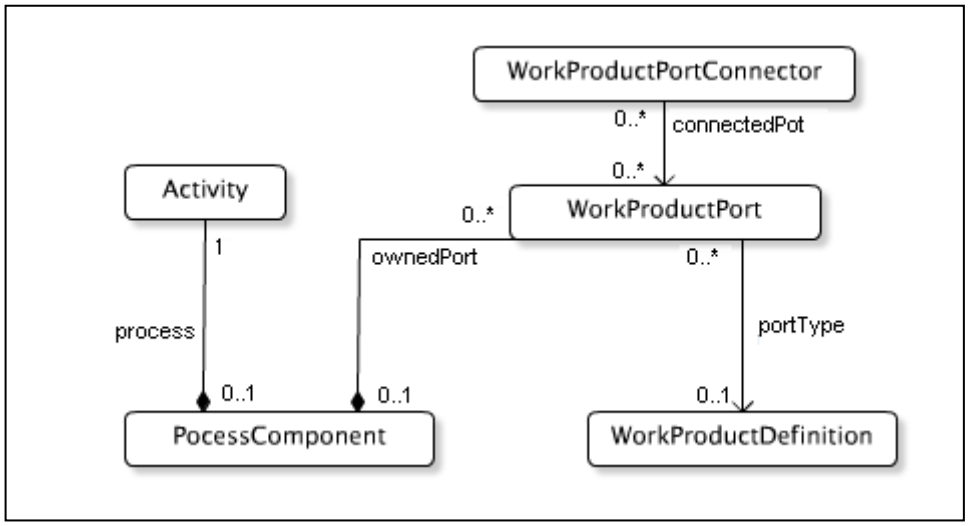


Figure5.11 - Méta-modèle simplifié du composant de procédé SPEM2.0 [SPE, 08].

CONCEPTS SPEM2.0	DESCRIPTION
ProcessComponent	Un composant de procédé contient exactement un processus représenté par une activité, et définit un ensemble de *WorkProductPort*.
Activity	L'activité définit une unité de travail basique dans un processus. SPEM ne limite pas la granularité de l'activité. En effet ce regroupement logique peut être constitué d'une tâche ou d'un sous processus
WorkProductPort	Définit les produits in/out pour un *ProcessComponent.*
WorkProductDefinition	Un constituant informatif ou une entité physique qui représente tout ce qui est produit, utilisé ou modifié par un processus.
WorkProductPortConnector	Connecte les *WorkProductPort* pour assembler un *ProcessComponent*

Tableau5.4 – Concepts SPEM2.0

2.2- INSTANCIATION DU MOF PAR SPEM2.0

En se basant sur la spécification SPEM2.0 de l'OMG, on a conclu les résultats présentés dans le tableau 5.5 :

- ProcessComponent est un regroupement cohérent d'éléments de modèle de processus, c'est une instance de la méta-classe *Class*.
- Une activité est un regroupement logique de tâches, de rôles et de produits, c'est une instance de la méta-classe *Class*.

- WorkProductPort définit une spécification du processComponent, c'est une instance de la méta-classe *Feature*.
- WorkProductDefinition définit un élément du procédé (*un artéfact ou un résultat*), c'est une instance de la méta-classe Class.
- WorkProductPortConnector définit un assemblage de *ProcessComponent,* c'est une instance de la méta-classe Association.

SPEM2.0	MOF
ProcessComponent	Class
Activity	Class
WorkProductPort	Feature
WorkProductDefinition	Class
WorkProductPortConnector	Association

Tableau 5.5- instanciation du MOF par SPEM2.0

2.3- META-MODELE DE RHODES

Le méta-modèle des composants de procédés RHODES **[THU, 01]** est présenté en UML dans la figure 5.12. Il peut être classé en deux (02) principales classes :

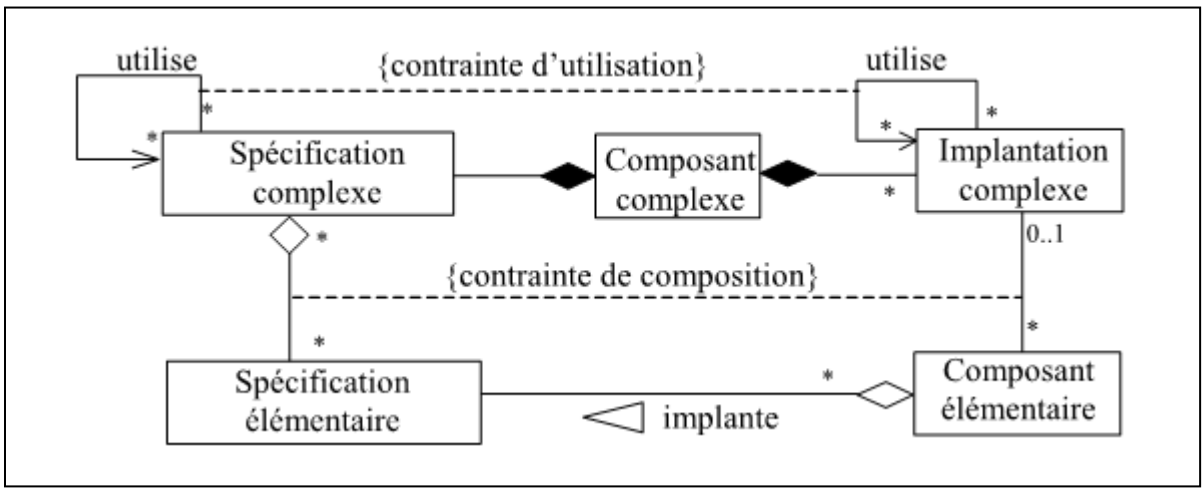


Figure5.12 : Méta-modèle RHODES [THU, 01].

- **Composant Complexe** : Chaque composant complexe possède deux parties : une partie *spécification* et une partie *implémentation*. Un composant complexe peut posséder plusieurs implémentations. On dit aussi que chaque implémentation du composant complexe implante ce composant ou implémente sa spécification.

  La spécification du composant complexe dite *''spécification complexe'' est* composée d'un ensemble de spécifications élémentaires. Une implémentation du composant complexe dite *''implémentation complexe''* si elle existe, correspond à un ensemble de composants

élémentaires dont chacun implante une spécification élémentaire inclus dans la spécification complexe.

- **Composant élémentaire** : Les composants élémentaires sont les entités de base pour décrire formellement et statiquement les procédés de développement. Ils correspondent aux constituants atomiques d'un procédé : les activités, les produits, les rôles et les stratégies. Chaque composant élémentaire possède une unique spécification.

2.4- INSTANCIATION DU MOF PAR RHODES

Un composant RHODES (*complexe ou élémentaire*) se compose de :

- Une partie spécification qui correspond à son interface, c'est une instance de la méta-classe *Feature*.
- Une partie implémentation qui détaille son comportement, c'est une instance de la méta-classe *BreavioralFeature*.
- Assertions qui décrivent des propriétés du composant, c'est des instances de la méta-classe *Property (StructalFeaure)*

Le tableau 5.6 donne un récapitulatif de l'instanciation du MOF par RHODES.

RHODES	MOF
Implémentation complexe	Class
Implémentation élémentaire	Class
Spécification élémentaire	Feature
Assertion	Feature
Composant complexe	Class

Tableau 5.6- instanciation du MOF par RHODES

3- DEFINITION DES TRANSFORMATIONS VERS ACME

Cette étape consiste à établir les mises en correspondances des méta-modèles de procédé et l'ADL pivot, en appliquant les étapes décrite dans la section IV.1.3 et en exploitant les résultats présentés dans la figure 5.9 et les tableaux 5.3,5.5 et 5.6. .

Le schéma de la figure 5.13 donne un exemple de mise en correspondance d'une entité SPEM2.0 avec une entité ACME, les numérotations correspondent aux étapes présentées dans la section IV.1.3.

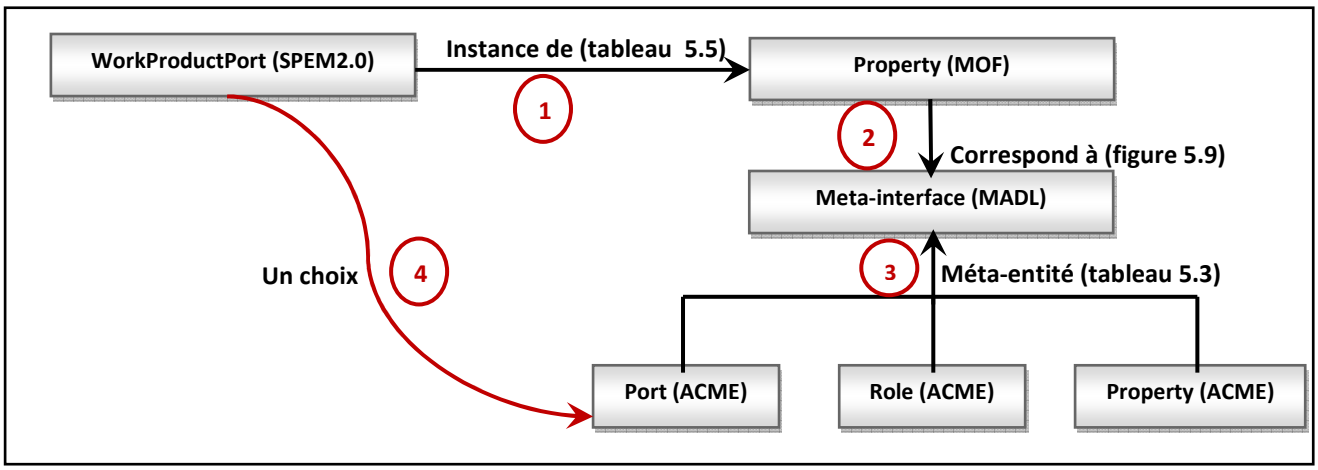


Figure 5.13- Exemple de mise en correspondance SPEM-ACME

En appliquant le processus schématisé dans la figure 5.13, sur toutes les entités SPEM2.0 et RHODES, nous obtenons les mises en correspondance RHODES-ACME et SPEM2.0-ACME décrites respectivement dans les tableaux 5.7 et 5.8.

RHODES	ACME
Implémentation complexe	Representation
Implémentation élémentaire	Component
Spécification élémentaire	Port
Assertion	property
Composant complexe	Component

Tableau5.7- Mapping RHODES-ACME

SPEM2.0	ACME
ProcessComponent	Component
Activity	Representation
WorkProductPort	Port
WorkProductDefinition	Component
WorkProductPortConnector	Connector

Tableau5.8- Mapping SPEM2.0-ACME

VII- ASSEMBLAGE

Dans cette étape, un architecte peut concevoir de nouveaux modèles de procédés par une simple intégration des composants de procédés unifiés dans les étapes précédentes.
Plusieurs types de connecteurs peuvent existés, qui dépendent des types de relations qui unissent les différents fragments du procédé (*dans ce qui suit, un fragment de procédé est mentionné par entité*). MAPL définit deux (02) principaux types de connecteurs de base : Binary_Connector et Generalized_Connector.

1- BINARY_CONNECTOR

Un connecteur binaire, qui relie deux entités d'un procédé. Ce type de connecteur peut exprimer, par exemple, les dépendances entre un modèle d'activité et un modèle de produit, un modèle d'activité et un autre modèle d'activité …

Le listing 5.1 décrit le connecteur binaire en ACME, qui se compose de deux types de rôles : Offert et Requis, attachés à des ports de type PortIn ou PortOut. Il vérifie la conformité entre les types de ports et des rôles attachés.

Description ACME

```
1   Connector type BinaryNode {
2   Role Offert {
3   rule un_port_attache = invariant  size (self.attachedPorts)== 1;
4   rule  type_interface = invariant forall p:Port in self.attachedPorts |
5   declaresType (p, PortIn);
6   }
7   Role Requis {
8   rule un_port_attache = invariant  size (self.attachedPorts)== 1;
9   rule  type_interface = invariant forall p:Port in self.attachedPorts |
10  declaresType (p, PortOut);
11  } }
```

Listing 5.1 - Description ACME du connecteur Binary_Connector

2- GENERALIZED_CONNECTOR

Ce type de connecteur est généralement utilisé pour exprimer des relations entre plusieurs entités. Sous MAPL, nous avons essentiellement modélisé des connecteurs qui expriment le dataflow d'un procédé, Ainsi, en se basant sur le diagramme d'activité du standard UML **[UMLa, 11]**. On a définit quatre (04) sous types du connecteur Generalized_Connector, décrites ci-après (*figure 5.14*).

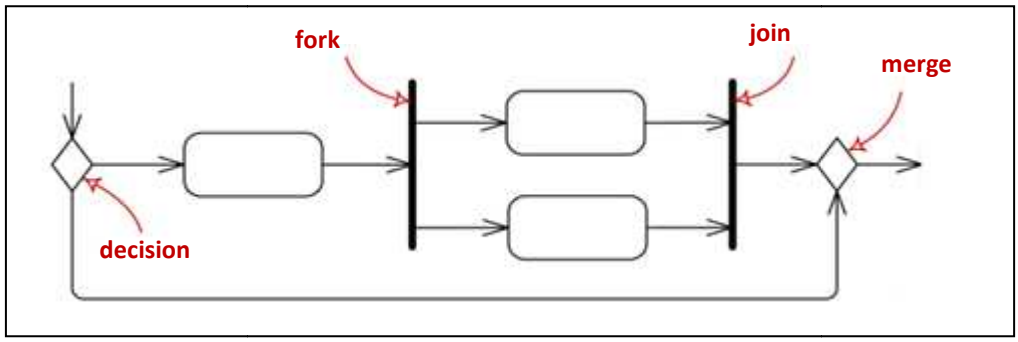


Figure 5.14 - Sous types du connecteur *Generalized_Connector*

Les connecteurs de contrôle, permettent de vérifier la validité des assemblages, en s'appuyant sur les informations fournisses par les interfaces des composants (leurs ports).

2.1- CONNECTEUR JOIN (*jonction, synchronisation*)

Ce connecteur (*figure 5.15*) est utilisé pour la synchronisation, ainsi, une entité (ou un fragment de procédé dans notre cas) ne peut commencer, que si, toutes les entités précédentes sont achevées.

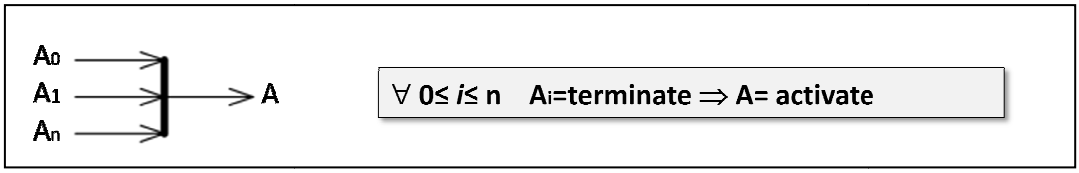


Figure 5.15- Le connecteur *Join*

Description ACME

```
Connector type JoinNode {
rule condActivation = Invariant  forall r1: Role in self.Roles |forall
r2:Role in self.Roles   | exists P2 :PortIn in r2.attachedPorts   | forall
P1 :PortOut in r1.attachedPorts | (r1 != r2 ) AND attached (r1,p1) AND
attached(r2,p2) AND P1.statut == Terminate -> P2.statut == Activate ;}
```

Listing 5.2 - Description ACME du connecteur join

Le connecteur join introduit une condition d'activation qui vérifie que le statut de l'ensemble des PortOut est ''Terminate'' pour activer le statut des PortIn attachés.

2.2- CONNECTEUR FORK (*fourche, parallélisme*)

Le connecteur fork, illustré dans la figure 5.16, est utilisé pour modéliser le parallélisme, ainsi, une série d'entité, peut démarrer simultanément, dés que l'entité précédente est achevée.

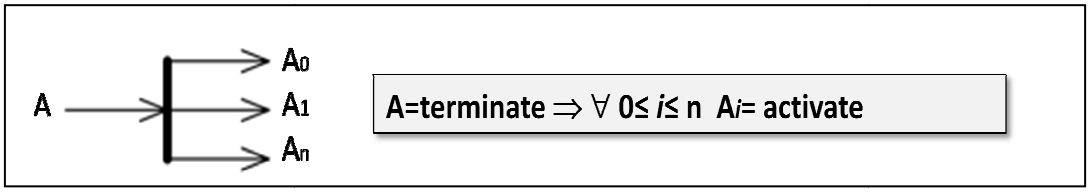


Figure 5.16- Le connecteur *Fork*

Description ACME

```
Connector type ForkNode {
rule condActivation = Invariant  forall r1: Role in self.Roles |forall
r2:Role in self.Roles   | exists P2 :PortOut in r2.attachedPorts | forall
P1 :PortIn in r1.attachedPorts   | (r1 != r2 ) AND attached (r1,p1) AND
attached(r2,p2) AND P2.statut == Terminate -> P1.statut == Activate ;}
```

Listing 5.3 - Description ACME du connecteur fork

Le connecteur fork introduit une condition d'activation qui active l'ensemble des PortIn dés que le statut de l'ensemble des PortOut est ''Terminate''.

2.3- CONNECTEUR MERGE (regroupement)

Le connecteur de regroupement, présenté dans la figure 5.17, est utilisé pour exprimer le cas ou l'achèvement d'une entité parmi une série d'entités, est suffisante pour démarrer l'entité suivante.

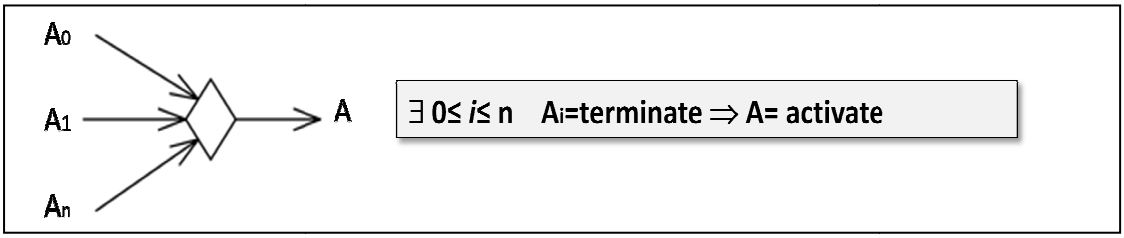


Figure 5.17- Le connecteur *Merge*

Description ACME

```
1  Connector type ForkNode {
2  rule condActivation = Invariant  forall r1: Role in self.Roles |forall
3  r2:Role in self.Roles   | exists P1 :PortIn in r1.attachedPorts | exists
4  P2 :PortOut in r2.attachedPorts   | (r1 != r2 ) AND attached (r1,p1) AND
5  attached(r2,p2) AND P2.statut == Terminate -> P1.statut == Activate ;}
```

Listing 5.4 - Description ACME du connecteur Merge

Le connecteur merge permet l'activation des PortIn dés que le statut de l'un des portOut est évalué à ''Terminate'' .

2.4- CONNECTEUR DECISION (*Alternatif*)

Le connecteur alternatif (*figure 5.18*) est un connecteur largement utilisé dans les procédés logiciels : il exprime un choix entre plusieurs entités selon certaines conditions.

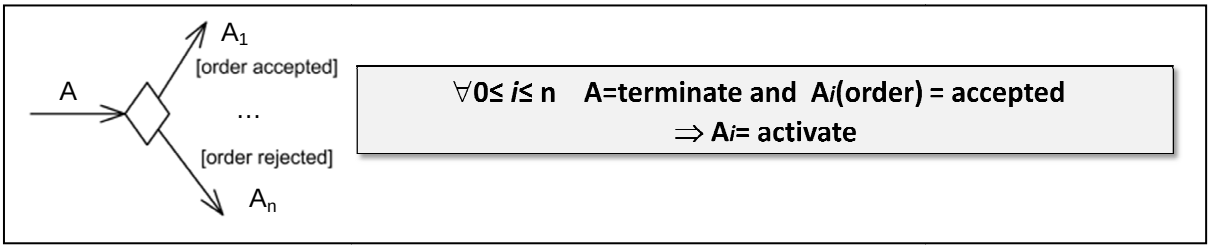


Figure 5.18- Le connecteur Décision

Description ACME

```
1   Connector type DecisionNode {
2   rule condActivation = Invariant  forall r1: Role in self.Roles |forall r2:Role in self.Roles   |
3   forall P1 :PortIn in r1.attachedPorts | exists P2 :PortOut in r2.attachedPorts | P2.statut ==
4   Terminate | (r1 != r2 ) AND attached (r1,p1) AND attached(r2,p2) AND  P1.order== Accepted  ->
5   P1.statut == Activate ;}
```

Listing 5.5 - Description ACME du connecteur Décision

Pour Activer des PortIn, le connecteur de décision, vérifie que le statut des portOut est ''Terminate'' et que la condition d'activation ''order'' est acceptée.

VIII – CONCLUSION

Nous avons présenté dans la première partie de ce chapitre notre modèle MAPL, qui repose essentiellement sur quatre (04) idées fondamentales :

- ✓ *La séparation entre partie externe et corps des CP* : Permet de masquer les détails d'implémentation, cette technique repose sur la notion d'interface du composant, qui contient toutes les informations nécessaires pour transférer des données en effectuant des abstractions de leur comportement. Ce qui présente une solution à

l'hétérogénéité syntaxique entre les différents formalismes de modélisation, un autre bénéfice est celui de la réutilisation. En effet, cette distinction est souvent considérée comme un élément indispensable à la réutilisation.

- ✓ *Le modèle pivot* : contient des concepts partagés entre les différents modèles de CP, il consiste en une homogénéisation des formats de représentation des modèles vers un format pivot, offrant ainsi, une solution acceptable pour l'hétérogénéité sémantique.

- ✓ *Modélisation explicite connecteurs* : Permet de simplifier considérablement les interactions entre les composants et les lois qui les régissent.

- ✓ *Le méta-modèle de transformation* : Permet de définir une abstraction des transformations. Les modèles instances de ce méta-modèle sont des spécifications de transformations entre deux méta-modèles spécifiques. De cette manière, la spécification de transformation devient lisible (*comprendre un modèle est plus facile que de comprendre du code*) et réutilisable (*le méta-modèle représente les règles de transformation de manière abstraite indépendante du langage de sa mise en œuvre*).

Dans la deuxième partie du chapitre, nous avons détaillé la démarche de conception de MAPL, qui comprend la migration des modèles de CP vers des modèles architecturaux, en se basant sur les techniques de modélisation et de méta-modélisation. La démarche proposée est :

- ✓ *Simple* : La plupart du travail est fait au niveau méta-méta, ou on travaille avec un nombre de notations réduit. En effet, l'utilisation de méta-méta-modèles contenant les concepts des méta-modèles facilite la composition des règles de transformations et minimise leur nombre et leur complexité.

- ✓ *Fiable* : Permet la vérification de la transformation sur différents niveaux d'abstraction.

- ✓ *Réutilisable :* Pour n'importe quel méta-modèle de procédé conforme au MOF, et pour n'importe quel ADL conforme à MADL : l'approche de la transformation d'un méta-modèle vers un autre permet de construire des règles de transformations très génériques puisque ces règles de transformation peuvent s'appliquer à tous les modèles pourvu qu'il respecte les méta-modèles source et cible.

Enfin, à l'instar de toute proposition conceptuelle, des études expérimentales sont nécessaires, pour d'une part, affiner la solution proposée et d'autre part, valider l'utilité de l'approche dans un modèle concret. Ainsi les deux chapitres suivants vont détaillés l'implémentation du modèle MAPL et sa validation par un modèle de procédé réel.

CHAPITRE 6

REALISATION

I- INTRODUCTION

Le modèle MAPL présenté dans le chapitre précédent offre des mécanismes et des concepts qui permettent de décrire une structure architecturale d'un procédé logiciel. Pour valider la spécification de ce modèle, nous avons implémenté un prototype de MAPL, appliqué au méta-modèles de procédé SPEM2.0 et RHODES, et à l'ADL pivot ACME.

Ce chapitre est présenté comme suit : Dans la section II nous exposons une vue d'ensemble de la démarche suivie, ensuite nous présentons les langages et les outils que nous avons utilisés. La section IV présente la mise en œuvre de MAPL et nous concluons par la section IV en faisant un bilan de notre implémentation.

II- ÉTAPES D'IMPLÉMENTATION DE MAPL

L'implémentation de MAPL est décomposée en trois (03) étapes essentielles, illustrées dans la figure 6.1 :

1- **Description des modèles et méta-modèles** : Dans cette étape, on décrit nos modèles dans un formalisme conceptuel au niveau abstrait, on le raffine par la suite vers des formalismes plus concrets.

2- **Description des transformations** : Cette étape consiste à décrire le modèle de transformation par une syntaxe abstraite qui permet d'exprimer naturellement les règles de transformations. Ces dernières, seront traduites en une syntaxe concrète exploitable par le moteur d'exécution.

3- **Exécution de la transformation** : s'effectue par un moteur d'exécution qui correspond au niveau concret de l'implémentation. Il prend en entrée le modèle et le méta-modèle source, le méta-modèle cible, ainsi que le modèle (règles) et le méta modèle de transformation, exprimés dans des formalismes concrets. Enfin, il utilise ces éléments pour établir la correspondance entre le modèle d'entrée et celui de sortie.

La figure 6.1 détaille les étapes d'implémentation de MAPL, ainsi que les outils et langages utilisés pour chacune d'elle.

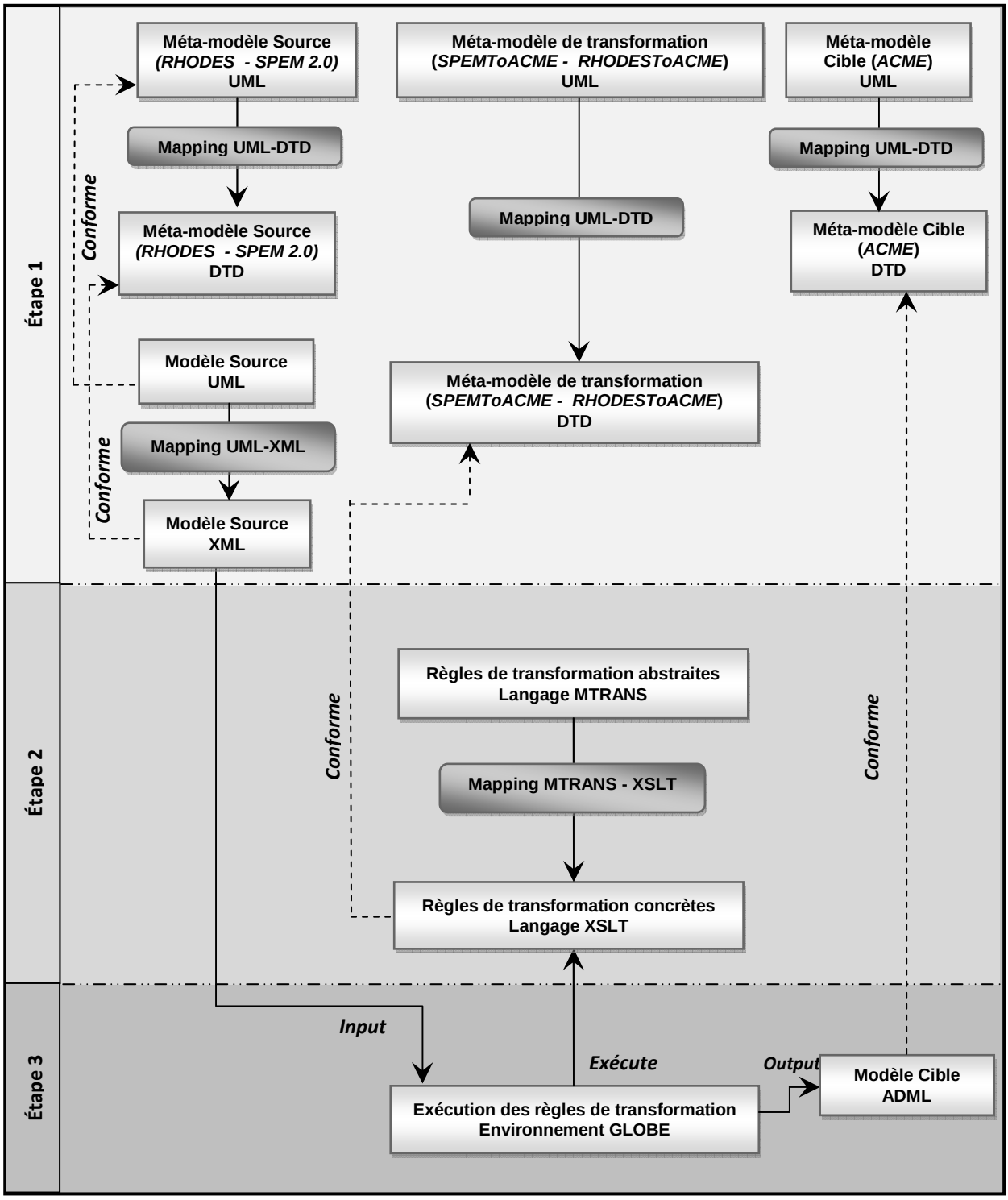


Figure 6.1- Architecture générale de l'implémentation de MAPL

III- LANGAGES ET OUTIL DE DEVELOPPEMENT

Chacune des étapes d'implémentation regroupe un ensemble d'outils et de techniques, appartenant à différents domaines et niveaux d'abstraction. Dans ce qui suit, nous allons détailler pour chaque étape les outils utilisés et les langages adoptés.

1- DESCRIPTION DES MODÈLES ET MÉTA-MODÈLES

1.1- LANGAGES REPRESENTATIFS CHOISIS

Le choix des langages représentatifs des données de modèles, et méta-modèles devra être bien étudié, en prenant en considération non seulement la capacité de ces langages à représenter des modèles, mais aussi leurs facilités de manipulation par l'utilisateur.

Ainsi, notre choix d'étude c'est posé sur deux langages développés par l'organisme W3C **[W3C]** que nous avons vue qu'ils étaient aptes à satisfaire nos objectifs fixés. Nous allons utiliser chacun de ces langages pour représenter une entité dans le moteur de transformation :

- Pour représenter les modèles d'entrée et sortie nous avons opté pour le langage XML **[XML].** ce langage qui est un standard depuis 1996, est utilisé à grande masse pour la représentation de données arborescentes. La puissance d'un tel langage dû à sa syntaxe extensible nous permettra la représentation de tous types de modèles, et étant un standard il est bien connu par la communauté informatique. plusieurs langages font partie de ses extensions, dont le plus fameux, le langage du Web HTML.

- Le choix du XML comme langage représentatif de nos modèles va nous mener naturellement à choisir son descripteur de données recommandé par le même organisme, on parle du descripteur DTD **[DTD].** Il nous servira comme langage représentatif des méta-modèles. Le contrôle de conformité des modèles par leurs méta-modèles se fera alors par un contrôle de validité DTD (*qui représente le méta-modèle*) sur le document XML (*qui représente le modèle*).

1.2- MAPPING DU DIAGRAMME DE CLASSE UML VERS DTD

XML permet d'utiliser les DTD afin de vérifier qu'un document XML est conforme à une syntaxe donnée, c'est-à-dire une grammaire permettant de vérifier la conformité du document XML. Ceci nous est utiles du temps qu'on peut exprimer nos modèle avec XML, et vérifier leur conformité en utilisant des DTD. Ainsi, la DTD joue le rôle de méta-modèle pour des documents XML pouvant alors être considérés comme des modèles.

Toutefois, afin de rapprocher les deux espaces techniques, XML et les modèles, nous avons mené une étude sur les travaux visant la génération automatique des documents XML à partir des diagrammes de classe UML. Une panoplie de travaux à été trouvée dans la

littérature, citons : **[KUD, 03], [NAR, 05], [SIN, 03], [CON, 00],** qui abordent en générale, la création de DTD et de schémas XML à partir des diagrammes de classes. Nous avons adopté les résultats des travaux de **[KUD, 03],** vu leur simplicité et leur parfaite correspondance à nos besoins.

Nous vous exposons dans le tableau 6.1 un résume du mapping UML vers DTD. Nous avons pris en compte dans ce mapping que les concepts exprimés dans les méta-modèles utilisés dans ce mémoire, nous avons fait abstraction du reste des concepts faisant partie des diagrammes de classe UML tel « l'agrégation ».

UML	DTD
Classe	Un Élément portant le même nom de la classe UML. L'élément a un attribut 'id' de type «ID» pour l'identifier. L'élément a un attribut 'name' (non obligatoire) portant le même nom que l'instance de la classe.
Attribut	Attribut d'élément portant le même nom que l'attribut UML et contenant les mêmes données en format «CDATA».
Association (1-1) à sens unique entre deux classes	Attribut référence (IDREF) de l'ID de l'élément représentant la classe cible de l'association, dans l'élément représentant la classe source.
Association (1-N) à sens unique entre deux classes	Un élément fils de l'élément présentant la classe source de l'association, ayant un attribut référence (IDREF) de l'ID de l'élément représentant la classe cible de l'association.
Relation de composition entre deux classes	L'élément représentant la classe composant et fils de l'élément représentant la classe composé.
Classe abstraite hérité	Non représentée, ces propriétés sont intégrer directement dans la classe qui l'hérite.
Enumération	Non représenté, sauf comme type d'attribut dans une classe (*voir attribut*).

Tableau 6.1- Mapping diagramme de classe UML vers DTD

2- DESCRIPTION DES TRANSFORMATIONS

Pour la description des transformations, nous optons pour un autre langage recommandé par le W3C pour la transformation de données XML est qui se trouve dérivé de ce dernier, il s'agit du langage XSLT **[XSL],** qui est un standard depuis 1997. Il nous servira à représenter les règles de transformation.

2.1- XSLT

2.1.1- INTRODUCTION

Le langage XSLT permet de définir des règles de transformations destinées à transformer un document XML bien formé en un autre.

Les caractéristiques principales d'une transformation XSLT sont :

- Une transformation peut traiter plusieurs fichiers sources et produire plusieurs fichiers cibles.
- Les fichiers sources traités dans XSLT doivent être des documents XML bien formés.
- Les fichiers cibles peuvent être produits en quatre (04) langages : soit HTML, soit XML, soit XHTML, soit texte.
- Les règles de transformations XSLT sont elles même écrites en XML, dans ce qui est appelé feuille de style "*Style Sheet*". Cette terminologie est liée au fait que les transformations XSLT sont principalement utilisées pour présenter des données XML sous forme HTML.

2.1.2 – SYNTAXE

- Syntaxe abstraite

Les *feuilles de styles XSLT* sont constituées par un ensemble de règles de transformation (*figure 6.2)*. Une *règle de transformations* se compose d'un *pattern* et d'un *replacement.* Le *pattern* est formé par des expressions XPath **[XPA]** qui indiquent sur quelles parties du document XML source doit s'appliquer la transformation. La partie *replacement* est composée d'*appels à des fonctions* de XSLT permettant de produire le fichier cible.

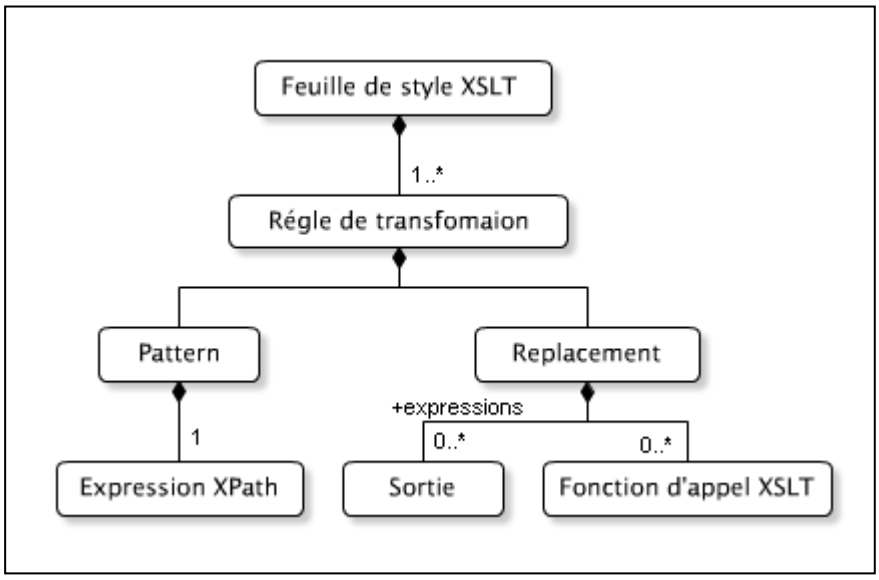


Figure6.2 - Structure d'une règle de XSLT

- Syntaxe concrète

D'un point de vue concret, un feuille de style commence et se termine par une paire de balise **<xsl:transform>** et **</xsl:transform>**, ou bien **<xsl:stylesheet>** et **</xsl:stylesheet>** Chaque règle correspond à un fragment **<xsl:template>** et **</xsl:template>** (*figure 6.3*).

L'attribut *match* contient une expression XPath permettant la localisation des nœuds. Cet attribut va définir les parts du document XML source sur lesquels s'applique la transformation. Des instructions d'exécution de transformation et les résultats output sont placés en mixte entre les deux balises **<xsl:template>** et **</xsl:template>.**

```
1   <?xml version = ''1.0''  encoding=''ISO-8859-1''?>
2     <xsl:stylesheet>
3           <xsl:template match=''noeud1 ''>
4            traitements sous forme de balises
5           </xsl:template>
6   .....
7           <xsl:template match=''noeud2 ''>
8           traitements sous forme de balises
9           </xsl:template>
10          …
11  </xsl:stylesheet>
```

Règle 1

Règle 2

Figure 6.3- Un exemple de feuille de style XSLT.

2.1.3 – DISCUSSION

Notre choix a aboutit à XSLT pour plusieurs raisons :

- Le stockage de données au format texte, ainsi on n'aura pas à manipuler du code.
- Indépendant de toutes plates-formes d'exécution.
- Standard libre du W3C**,** sur lequel se base plusieurs travaux qui couvrent divers domaine.
- Offre :
  - Modularité. Les feuilles de style de XSLT peuvent être composées pour former des transformations complexes à partir de transformations plus simples.
  - Interopérabilité. XSLT peut interagir avec d'autres langages comme Java, C# ou autres.
  - Extensibilité. XSLT fournit deux mécanismes d'extensibilité. La première permet d'étendre le jeu d'instruction de XSLT lui-même, alors que le deuxième permet d'étendre l'ensemble des fonctions proposées par XPath.

Cependant le XSLT reste confronté à certaines failles, principalement:

- La syntaxe XSLT est très verbeuse et peu lisible, au point de compromettre la maintenabilité de transformation pourtant relativement simple.
- XSLT, manque une syntaxe abstraite de plus haut niveau pour la spécification de transformations.

Pour pallier ce problème des solutions ont été proposées, citons principalement la solution retenue par Mikael PELTIER **[PELa,OO] [PELb,OO]**, qui exploite XSLT non pas comme langage de spécification des transformations, mais comme environnement d'exécution des transformations. En conséquence, nous avons adopté cette solution, qui sera exposée dans la section suivante.

2.2- LANGAGE MTRANS

MTRANS **[PEL,O1]** est un langage dédié à la transformation des modèles. Développé à un niveau d'abstraction au-dessus de XSLT, il l'utilise pour transformer des modèles, il est plus compact et facile à comprendre que XSLT.

Le langage MTRANS permet d'exprimer de manière plus commode, les règles définissant la méthode pour transformer les entités d'un modèle vers les entités d'un autre modèle. Dans ce contexte, une entité représente simplement un concept du modèle source ou cible.

Le listing 6.1 illustre la syntaxe concrète d'une règle de transformation MTRANS.

```
1   rules_transformation ::= Entity [restriction]? to Entity
2                            { attributes : [Attribute = attributes_transformation,]*
3                              roles : [Role = roles_transformation,]* }
4                           | Entity
5                           { attributes : [Attribute = attributes_transformation,]*
6                            roles : [Role = roles_transformation,]* }
7
8   restriction ::= Attribute operator value | Role operator value
9   attributes_transformation ::= Attribute | Entity.Attribute | "Literal"
10                                | ( condition ? resultIfTrue : resultIfFalse )
11                                | Attribute # Attribute
12
13  resultIfTrue ::= attributes_transformation
14  resultIfFalse ::= attributes_transformation
15
16  roles_transformation ::= Role | ( condition ? resultIfTrueBis : resultIfFalseBis )
17                                  | Role[Entity]
18
19  resultIfTrueBis ::= roles_transformation
20  resultIfFalseBis ::=roles_transformation
```

Listing 6.1- La syntaxe de MTRANS [PEL,O1].

Une règle de transformations d'entité est divisée en deux parties : la première est utilisée pour spécifier l'entité source et la seconde pour l'entité destination. En appliquant des restrictions, la règle peut être limitée aux entités respectant une certaine condition.

Une restriction est également composée de deux parties : une partie droite qui détermine un rôle ou un attribut de l'entité et une partie gauche qui correspond à la valeur acceptée pour ce rôle ou cet attribut. MTRANS permet également de créer de nouvelles entités.
De plus, MTRANS permet d'énoncer les règles spécifiant le procédé de transformation des attributs et des rôles d'une entité. Ces règles prennent la forme d'une assignation dont la partie gauche correspond au nom de l'attribut ou du rôle transformé et la partie droite définit la valeur de cet attribut ou ce rôle dans le modèle de destination.

2.3- CORRESPONDANCE ENTRE MTRANS ET XSLT

Les mises en correspondances MTRANS vs XSLT, sont détaillées dans **[PELa, OO]**, dans ce qui suit, nous allons donner juste un aperçu de ce mapping:

- **Transformation d'entité** : Consiste à ajouter un élément dans l'arbre XML de destination. Un exemple du schéma général de la règle de transformation d'entité est illustré dans la figure 6.4. A la fin de cette règle de transformations d'entité, des règles de correspondances des attributs et des rôles de cette même entité seront ajoutés.

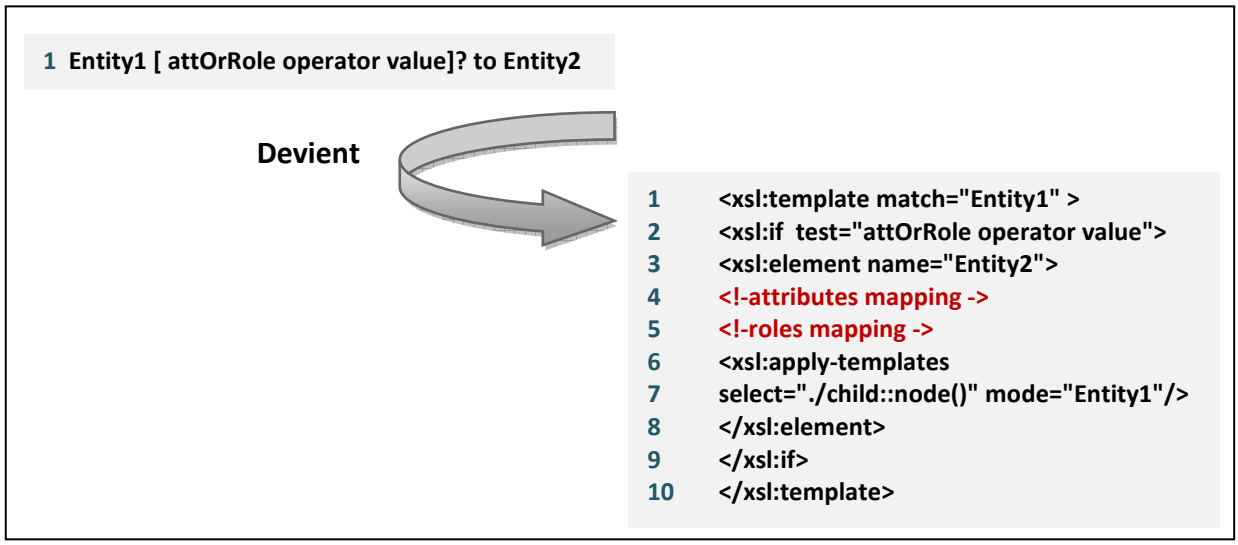


Figure6.4 – Transformation d'entité MTRANS vers XSLT

- **Transformation d'attribut**: XML peut utiliser deux façons pour représenter un attribut, il peut être un attribut propriétaire de l'entité, ou un nœud entité. Par conséquent, la transformation d'attribut tient compte de ces deux cas. Plusieurs correspondances, ont été ainsi décrites :
  - Attribute = ''value''
  - Attribute1 = Attribute2
  - Attribute1 = (Attribute2 operator 'value' ? Attribute3 : Attribute 4)

- **Transformation de rôles** : tous comme la transformation d'attribut, le rôle possède trois types de transformation :
  - Role1 = Role2
  - Role1 = Role2 [Entity1]
  - Role1 = (Role2 operator 'value' ? Role3 : Role4)

3- EXECUTION DE LA TRANSFORMATION

3.1- PARSEUR XSLT

La transformation XSLT est exécutée par un parseur XSLT dont le principe d'exécution est décrit dans la figure 6.5.

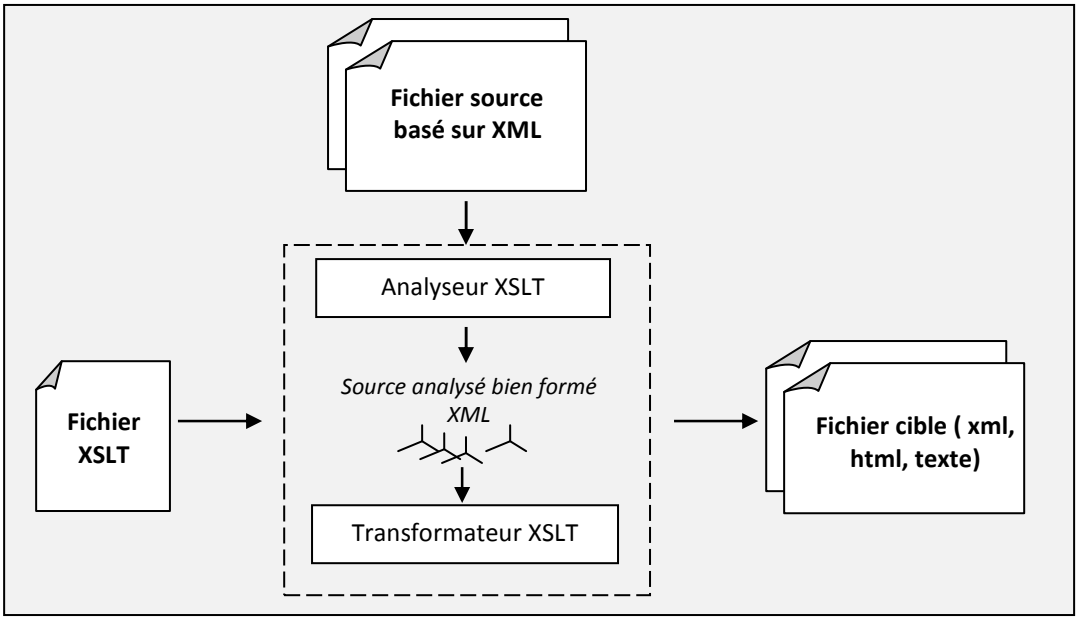


Figure 6.5- Vision conceptuelle du processus de transformation XSLT

L'entrée de l'analyseur correspond à un ou plusieurs fichiers XML et un fichier XSLT. L'exécution d'une transformation se réalise en deux (02) étapes principales :

- Analyse des fichiers sources et des fichiers de transformation XSLT : Dans cette étape XSLT vérifie si les fichiers sont bien formés ou non, et charge la feuille de style XSLT.

- Application des règles de transformation : Les règles de transformation sont ensuite appliquées en fonction du contenu du fichier source. Le parseur parcourt le document XML initial du début à la fin sous forme d'arbre, des éléments parents vers les éléments enfants sauf si la feuille de style change l'ordre de parcours

3.2 – DISCUSSION

L'étude à montré la faisabilité et la simplicité des transformations à l'aide d'XSLT. Cependant, nous avons constaté, qu'un parseur XSLT ne correspondant pas parfaitement à notre architecture de transformation *(figure 6.1*). Presque aucune contrainte de conformité n'est appliquée aux fichiers source et cible. En effet un parseur XSLT:

- Permet de prendre en entrée des fichiers XML sources quelconques du moment que ces fichiers sont bien formés. Aucune validation de conformité n'est nécessaire par rapport à un schéma ou une DTD particulière. On peut donc traiter tous les fichiers XML qu'ils aient ou non un schéma ou une DTD explicite. Le manque de typage de l'entrée pose évidemment des problèmes car aucune vérification de conformité n'est faite statiquement. Le résultat produit par une transformation sur un fichier source ne correspondant pas au format attendu, produira des résultats mais ces résultats ne seront peut être pas corrects.

- Laisse le développeur générer librement ce qu'il veut pourvu qu'il s'agit soit d'un document XML bien-formée ou d'un texte quelconque. Cette technique présente l'avantage d'être très flexible. Il n'y a presque aucune restriction sur le langage cible si celui-ci a une représentation textuelle. L'inconvénient est que l'utilisateur doit gérer lui-même le contenu des fichiers cibles, sans qu'il n'y ait aucune garantie de conformité par rapport au méta-modèle cible.

Ainsi, il serait souhaitable de disposer d'un mécanisme qui permet de vérifier la conformité des fichiers source et cible par rapport à une description données. Dans cette intention, nous avons développé un moteur de transformation, GLOBE **[ARO, 12]** (*décrit dans la section suivante*), qui intègre en plus d'un parseur XSLT, des outils d'analyse, qui vérifient respectivement la conformité de la spécification des fichiers source et cible avec leurs méta-modèles.

3.3- GLOBE 1.0

3.3.1- INTRODUCTION

GLOBE1.0 est un système de transformation de modèle, présenté en annexe 2, qu'on a développé pour la mise en œuvre de MAPL. Son moteur d'exécution supporte le langage XSLT, il est non seulement capable de transformer le modèle d'entrée en modèle de sortie (*suivant le schéma de transformation XSLT*), mais il garanti un contrôle de conformité entre les modèles et leurs méta-modèles associés, pour cela un procédé d'attachement a été proposé à l'utilisateur lui permettant la spécification des associations entre modèles et méta-modèles, ainsi qu'entre modèles d'entrée et modèle de transformation.

Les règles de transformation sous GLOBE 1.1, peuvent être exprimées sur deux niveaux d'abstraction : en langage MTRANS, qui permet d'exprimer naturellement les règles de transformations, et en XSLT pour une exploitation concrète.

3.3.2 - LANGAGE ET OUTIL DE DEVELOPPEMENT

Pour réaliser et mettre en œuvre le système GLOBE (*figure 6.6*), on a opté pour :

- Le langage JAVA **[JAVA]**, et ceci pour plusieurs raisons :
  - Tout d'abord parce qu'il offre une grande portabilité. En effet Java est l'un des langages fortement portable, pour cette raison Sun Microsystems à mis à disposition des utilisateurs sur son site web **[SUN],** une machine virtuel téléchargeable gratuitement, elle est supporté par la plupart des plateformes système connu, nous citons les plus connus d'entre elles (Windows **[WIN]**, Unix **[UNI]** et Mac **[MAC]**).

- Il offre une multitude d'API (*Application Programming Interface*) fournit en standard avec la machine virtuelle ou développés par des organismes détachés de Sun Microsystems, mais travaillent sur des projets de développement Java.
- Il existe d'autres avantages offerts par ce langage que nous ne pouvons pas cités en détaille, parmi ces avantages, sa capacité d'adaptation avec d'autre langage tel C++ pour d'éventuel extension.

- L'outil de développement intégrés Eclipse **[ECL],** qui est considéré comme l'un des plus puissant IDE (*Integrated Development Environment*) JAVA mis sur le marché, dû à la richesse de ses fonctionnalités et ses outils de développement spécialisés.

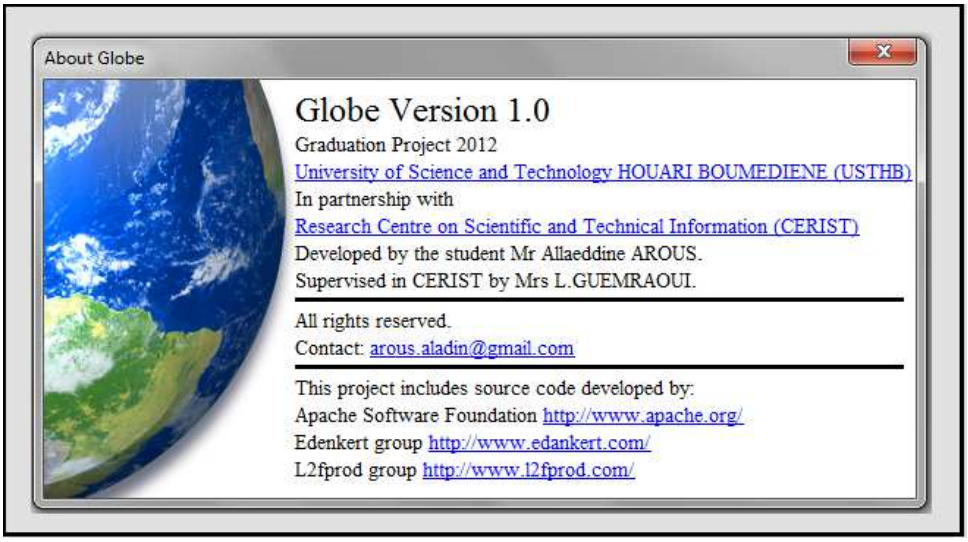


Figure 6.6- Fenêtre "About Globe " du système GLOBE1.0

3.3.3- PRINCIPALES FONCTIONNALITES

Les principaux composants de GLOBE1.0, peuvent se résumés dans la fenêtre principale illustrée dans par la figure 6.7, les éléments numérotés sont expliqués dans le tableau 6.2.

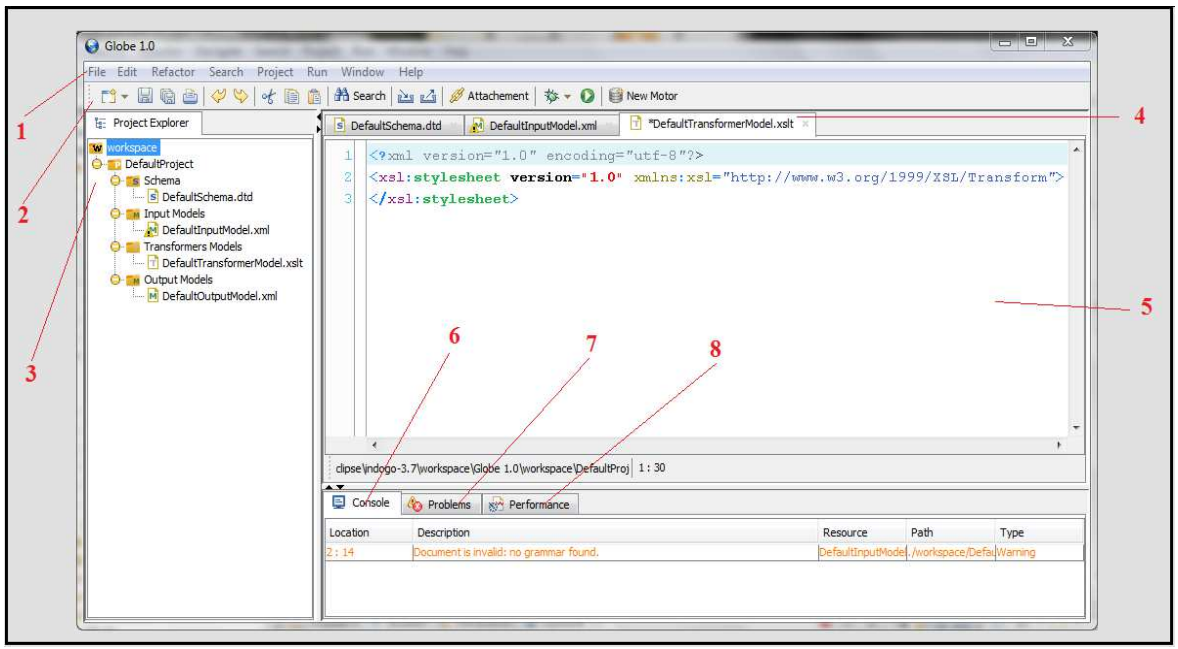


Figure6.7 – Fenêtre principale de GLOBE 1.0

N°	Nom du composant	Description du composant
1	Barre de menu	Comprend toutes les options offertes par « *Globe* ».
2	Barre d'outils	Contient quelques options utilitaires de la barre de menu.
3	Explorateur de projet	Offre une vision arborescente du système de données « *Globe* ».
4	Anglet d'exposition	Contient les feuilles d'édition.
5	Feuille d'édition	Fenêtre d'édition dont les données sont exposées.
6	Console	Affiche les erreurs de compilation éventuelles pour les données des divers langages supportés par « *Globe* » (DTD, XML, XSLT) d'un projet ou d'un fichier spécifique.
7	Fenêtre d'erreur	Affiche toutes les erreurs de compilation trouvées dans les fichiers du système de données pour les divers projets et fichier libre.
8	Fenêtre des performances	Affiche un graphe de performance à temps réel de l'application.

Tableau 6.2 - Description des principaux composants constituants la fenêtre principale de " GLOBE1.0 ".

4- Description architecturale du modèle cible

Les documents XML sont assez complexes. En effet, Il est difficile d'avoir une compréhension rapide et globale d'un vocabulaire XML. C'est pourquoi, nous avons jugé intéressant de représenter ce vocabulaire en un langage ayant une notation graphique, qui permet d'exprimer visuellement la solution, ce qui facilite la compréhension et l'évaluation des solutions. De plus, vu que l'objectif visé par nos travaux est d'aboutir à une description architecturale, nous avons ainsi, proposé la représentation du document cible en ADML **[ADM]**, une version XML d'ACME qui lui offre une représentation standardisée, interprétable par les parseurs XML ordinaires.

Le choix d'ADML est du principalement au fait qu'il est basé sur ACME, en conséquence un modèle ADML se traduit aisément en un modèle ACME et il est en mesure de tirer parti des outils de conception et d'analyse développés pour la description ACME.

IV- ETUDE DE CAS

Pour bien comprendre le processus d'implémentation, nous exposons, dans cette partie, des exemples de mise en œuvre de MAPL, extraites de la transformation SPEM2.0 vers ACME.

1- DESCRIPTION DES META-MODELES SOURCE ET CIBLE

Dans cette étape nous allons exprimer nos méta-modèles SPEM2.0 et ACME, en des descripteurs DTD.

Les méta-modèles SPEM2.0 et ACME ont été définis par des diagrammes UML, dans le chapitre 5 (*voir respectivement les figures 5.11 et 5.12*).On appliquant Le mapping diagramme de classe UML vers DTD défini dans le tableau 6.1 , sur les méta-modèles SPEM2.0 et ACME nous obtenons leurs schémas DTD présentés dans l'annexe 1. Les listings 6.2 et 6.3 donnent des aperçus des DTD SPEM2.0 et ACME.

Le listing 6.2 donne la définition d'un WorkProductDefinition SPEM2.0 en XML.

```
<!ELEMENT WorkProductDefinition EMPTY>
<!ATTLIST WorkProductDefinition id ID #REQUIRED>
<!ATTLIST WorkProductDefinition name CDATA #IMPLIED>
```

Listing 6.2 – Description du WorkProductDefinition SPEM2.0 en DTD

Le listing 6.3 présente la description XML du Component ACME.

```
<!ELEMENT Component (Port|Property)*>
                <!ATTLIST Component id ID #REQUIRED>
                <!ATTLIST Component name CDATA #IMPLIED>
                <!ELEMENT Port (Property)*>
                        <!ATTLIST Port id ID #REQUIRED>
                        <!ATTLIST Port name CDATA #IMPLIED>
                        <!ELEMENT Property EMPTY>
                                <!ATTLIST Property name CDATA #REQUIRED>
                                <!ATTLIST Property value CDATA #REQUIRED>
```

Listing 6.3 – Description du Component ACME en DTD

2 - DESCRIPTION DES TRANSFORMATIONS

Le moreau de code MTRANS illustré dans le listing 6.4, montre un exemple de règle de transformation SPEM2.0 vers ACME, qui interprète la transformation d'un *WorkProductDefinition* SPEM2.0 en un Component ACME.

```
SetOfRules
{ WorkProductDefinition to Component
    { attributes : name = name
     .... }}
```

Listing 6.4- Exemple de règle MTRANS

En appliquant le mapping MTRANS vers XSLT, on génère, à partir des règles du niveau abstrait en MTRANS, des règles XSLT plus concrètes adaptées aux modèles à transformer.
Le listing 6.5 traduit le morceau de code MTRANS du listing 6.4. La totalité du code de transformation SPEM2.0ToACME est présenté en l'annexe 1.

```
<xsl:for-each select="/Process/WorkProductDefinition">
  <Component id="{@id}" name="{@name}">
  … </Component> </xsl:for-each>
```

Listing 6.5- Exemple de règle de transformation SPEM2.0ToACME exprimée en XSLT

Un éditeur graphique des règles, intégré à la version GLOBE1.1, permet l'écriture des règles de transformation en MTRANS, et il génère automatiquement le code XSLT correspondant.

3- EXECUTION DE LA TRANSFORMATION

L'exécution des règles de transformation sous GLOBE 1.0 est simple, en effet, après l'intégration du modèle SPEM2.0 et le modèle de transformation, ainsi que les trois méta-modèles SPEM2.0, ACME et de transformation dans GLOBE, on spécifie l'attachement entre ces différentes entités en attribuant chaque méta-modèle à son modèle ainsi que l'indication du modèle de transformation via une fenêtre d'attachement (*figure 6.*8) . Il nous reste après qu'un simple clique sur le bouton d'exécution pour que GLOBE se charge de la transformation.

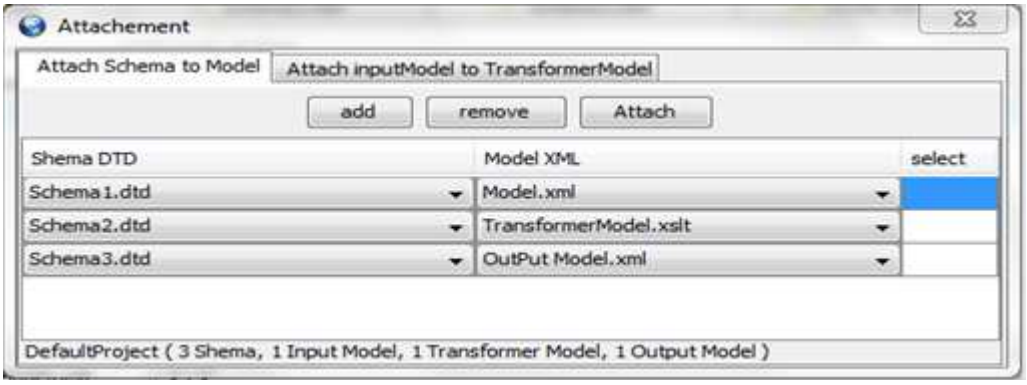


Figure 6.8- Fenêtre d'attachement de modèles

V – CONCLUSION

Nous avons exposé au fil de ce chapitre la démarche de réalisation de MAPL, qui se base essentiellement sur la dissociation entre la syntaxe abstraite et concrète de nos données. Cette distinction a pour but principal de :

- Favoriser la réutilisation des modèles déjà construit.
- Permettre d'exprimer et de définir les concepts utilisés dans les modèles et méta-modèles indépendamment de tout support de présentation des informations.

Pour la mise en œuvre de cette démarche nous avons opté pour les technologies XML. Ce choix est motivé par le souhait de ne pas enfermer nos données dans un format propriétaire, et être ensuite tributaire des possibilités techniques offertes par ce format.

Un autre risque lié au format propriétaire est de plus pouvoir réutiliser les données dans le temps, par exemple à cause d'un changement de format lors d'une évolution logicielle.

D'autre part, le monde XML offre une palette de possibilités technologiques qui sont complémentaires. Parmi eux se trouve, le langage de transformation adopté : Le langage XSLT qui est approprié pour transformer un document XML en un autre.

Cependant, l'écriture d'une feuille de style est longue et pénible car XSLT n'est pas réservé uniquement à la transformation de modèles. Par contre, l'utilisation d'un langage abstrait

(MTRANS) adapté à la transformation de modèle permet d'exprimer naturellement les règles de transformations. De plus, contrairement aux solutions de transformation basées sur un code exécutable rigide dans lesquelles le code devrait être remodelé pour pouvoir faire évoluer une transformation, l'architecture de transformation utilisant des feuilles de style XSLT et un parseur pour les interpréter est pleinement évolutive. En effet, pour pouvoir faire évoluer une transformation entre deux modèles, il suffit simplement de changer la feuille de style utilisée.

Enfin, la solution proposée à été soutenu par un moteur de transformation qui édite, interprète et gère facilement les modèles traités. En plus, il permet une exécution souple et fiable des transformations.

CHAPITRE 7

VALIDATION

I- INTRODUCTION

Pour évaluer notre approche, nous utilisons MAPL pour modéliser une partie de l'exemple ISPW-6 **[KEL, 91],** un benchmark largement utilisé pour évaluer et comparer les approches de modélisation des procédés logicielles. Une discussion sur les résultats de cette évaluation est donnée à la fin de la section II.

Dans la section III, on dresse une deuxième évaluation, à travers une étude comparative basée sur les principales caractéristiques des PML, que nous avons définie dans le chapitre 2 de ce document. Enfin, quelques observations sur les résultats obtenus concluent ce chapitre.

II- EVALUATION PAR ISPW-6 SOFTWARE PROCESS EXAMPLE

1 – ISPW-6

L'exemple ISPW-6 est un exemple de référence, un benchmark, initialement publié dans ‘’*Proceedings of the 6th International Software Process Workshop: Support for the Software Process*" en 1990, sous le titre de " *Software Process Modeling Example Problem*". Le titre a été changé pour «*ISPW-6 Software Process Example*» afin de distinguer l'exemple original des versions ultérieures.

Cet exemple a été soigneusement conçu pour faciliter une évaluation objective et la comparaison entre les modèles de processus logiciels. Il contient un grand nombre de différents types de problèmes liés aux procédés, observés dans le monde réel. Plusieurs extensions à l'exemple ont été introduites depuis, mais ce document ne traite que l'exemple de base dans la description originale.

L'exemple de base concerne l'exécution et la gestion d'un changement des exigences relatives à un composant logiciel existant. Le procédé est lancé en réponse à la modification d'une unité d'exigence. Le procédé commence avec le gestionnaire de projet, qui planifie des

changements et répartit le travail au personnel approprié. Il se termine lorsque la nouvelle version du code a passé avec succès les tests unitaires.

L'exemple ISPW-6 se compose des éléments suivants :

- **Agents**: Ce sont le personnel, par exemple : gestionnaire de projet, les ingénieurs de conception, les ingénieurs d'assurance qualité.
- **Étapes**: Ce sont les actions effectuées par les agents, par exemple : *Modify_design*, *Review_design*. Chaque agent peut effectuer plusieurs étapes et une étape peut être réalisée par plusieurs agents.
- **Entrées et sorties**: Ce sont les données requises et produites par chaque étape, par exemple : *Modified_design, Design_review_feedback.*
- **Contraintes**: Ceux-ci décrivent l'enchaînement autorisé des étapes.

Les principales étapes de cet exemple sont illustrées dans la figure 7.1 . Les détails de ce qui doit être effectué au sein de chaque étape, les entrées (Input), les sorties (Output), les rôles et les contraintes sont donnés dans **[KEL, 91].** Toutefois, afin de facilité la présentation, nous avons choisi de nous concentrer seulement sur deux étapes principales du procédé, qui modélisent deux fragments distincts.

2- ÉTUDE GLOBALE DU PROCEDE

Dans ce qui suit, nous allons donner une description informelle et globale de deux (02) étapes du procédé : *Modify_design* et R*eview_design*, en mettant en évidence, les rôles des agents participants, et les produits manipulés tout au long du développement.

2.1- MODIFICATION DE LA CONCEPTION (*MODIFY DESIGN*)

Description

Cette étape implique la modification de la conception pour l'unité de code affecté par le changement des exigences. Les modifications seront examinées et, finalement, mis en œuvre dans le code. Cette étape peut également modifier la conception basée sur le feedback de la revue de conception (*Design_review_feedback).*

Entrées

- La conception actuelle (*Current_design*)
- Le feedback de la revue de conception (*Design_review_feedback).*

Sorties

- La conception modifiée (*Modified_design*).

Agents

- Ingénieur de conception

Contraintes

- La première itération commence dés que la tâche est assignée par le chef de projet.
- Itération ultérieure peut commencer dés que la revue de conception est terminée (*Quand la conception n'est pas approuvée*).
- Cette étape est terminée lorsque sa sortie a été fournie.

L'ensemble du processus: Élaborer des changements et des tests unitaires

➘ **ÉTAPES**

- Planifier et assigner des tâches.
- Modifier la conception.
- Évaluer la modification.
- Modifier le code.
- Modifier les plans de test.
- Modifier les tests unitaires.
- Test Unit.
- Suivre l'évolution.

➘ **ORGANISATION**

- CCB (*Commission de contrôle de configuration*)
- Équipe du projet:
  - Chef de projet
  - Les ingénieurs de conception
  - Ingénieurs QA
  - Les ingénieurs logiciels

➘ **ARTEFACT**

- Entrée + Source (*fichier, agent, ou étape*) + mécanisme de communication physique.
- Sortie + Destination (*fichier, agent, ou étape*) + mécanisme de communication physique.

➘ **CONTRAINTES** concernant l'enchainement des étapes.

Figure 7.1 – Description générale de l'ISPW6.

2.2- ÉVALUER LA MODIFICATION (*REVIEW_DESIGN*)

Description

Cette étape consiste à l'examen formel de la modification de la conception. Elle est menée par une équipe comprenant l'ingénieur de conception qui a produit des modifications de conception.

Il ya trois (03) résultats possibles de l'examen:

- *Approbation inconditionnelle* : La conception est totalement approuvée; la conception approuvée est incorporée dans le document de conception de logiciels.
- *Des modifications mineures recommandées* : Modifications mineures à la conception sont nécessaires et les commentaires sont fournis pour le concepteur.
- *Des changements majeurs recommandés* : Changements majeurs à la conception sont nécessaires et la rétroaction est prévue pour le concepteur.

Entrées

- La conception modifiée (*Modified_design)*

Sorties

- Le feedback de la revue de conception (*Design_review_feedback).*
- La conception modifiée et approuvée (*Approved_modified_design*)
- Notification des résultats, le nombre de défauts identifiés, l'effort global …

Agents

Cette étape est réalisée par l'équipe de revue de conception attribuée par le chef de projet. Cette équipe comprend l'ingénieur de conception, ingénieur QA (Assurance Qualité), et deux autres ingénieurs logiciels.

Contraintes

Cette étape sera réalisée dés que la modification de la conception est disponible et elle se termine lorsque ses sorties ont été fournis.

3- FORMALISATION EN PML

Dans cette section, on va formaliser les deux étapes choisis en deux PML hétérogènes, étudies précédemment : RHODES et SPEM 2.0. Ainsi, à partir de la description informelle du procédé, nous avons pu formaliser les entités de ce procédé.

3.1- FORMALISATION DE L'ACTIVITE *''MODIFY_DESIGN''* EN RHODES

Pour représenter l'activité *''Modify_design''* sous RHODES, on a pris en considération deux points essentiels :

- Le concepteur de procédé, sous RHODES, peut sélectionner des composants élémentaires réutilisables, puis les organiser en composants complexes qui sont autonomes et réutilisables.

  Par exemple, on peut constater que l'activité «*Modify_design*» est réutilisable, et qu'elle mérite d'être développée pour devenir un composant autonome réutilisable. Pour ce faire, ce composant doit comporter certains composants plutôt qu'une seule activité. Ainsi, on prend tous les composants participant à l'activité, puis on les organise en un groupe.

- La représentation du procédé sous RHODES se base sur des diagrammes UML : les composants complexes sont modélisés par des packages UML, et les composants élémentaires par des classes UML.

Ainsi, l'étape ''Modify_design'' sous RHODES peut être modélisée, comme décrite dans la figure7.2.

En plus, de la représentation semi-formelle en UML, RHODES offre une description formelle en langage PBOOL+.

La syntaxe PBOOL+ des composants complexes est très simple (*listing7.1*). La description d'un composant complexe et de ses implémentations est fondée sur les spécifications élémentaires et sur les composants élémentaires. Comme un composant complexe correspond uniquement à sa spécification, nous le définissons en déclarant sa spécification (le mot-clé « *specification component* » qui est suivi par le nom du composant).

Dans notre exemple, le composant complexe possède une seule implémentation. Cependant, d'autres implémentations du composant peuvent être développées au cours de son exécution.

```
1   specification component MODIFY_DESIGN
2   -- les spécifications élémentaires
3           specification activity  ModifyDesigne_SPEC;
4           specification activity  ReviewDesigne_SPEC;
5           specification activity  ModifyCode_SPEC;
6           specification activity  ModifyUnitTest_SPEC;
7           specification role      DesigneEnginier_SPEC;
8           specification product   DesigneDocumentFB_SPEC;
9           specification product   DesigneDocument_SPEC;
10          end ; -- MODIFY_DESIGN
11
12  component MODIFY_DESIGN_IMP1 implement MODIFY_DESIGN
13  -- les composants élémentaires correspondant aux spécifications élémentaires
14          implementation
15          activity   ModifyDesigne;
16          activity   ReviewDesigne;
17          activity   ModifyCode;
18          activity   ModifyUnitTest;
19          role       DesigneEnginier;
20          product    DesigneDocumentFB;
21          product    DesigneDocument;
22  end ; -- MODIFY_DESIGN_IMP1
```

Listing 7.1- Composant complexe concernant l'activité « Modify design »

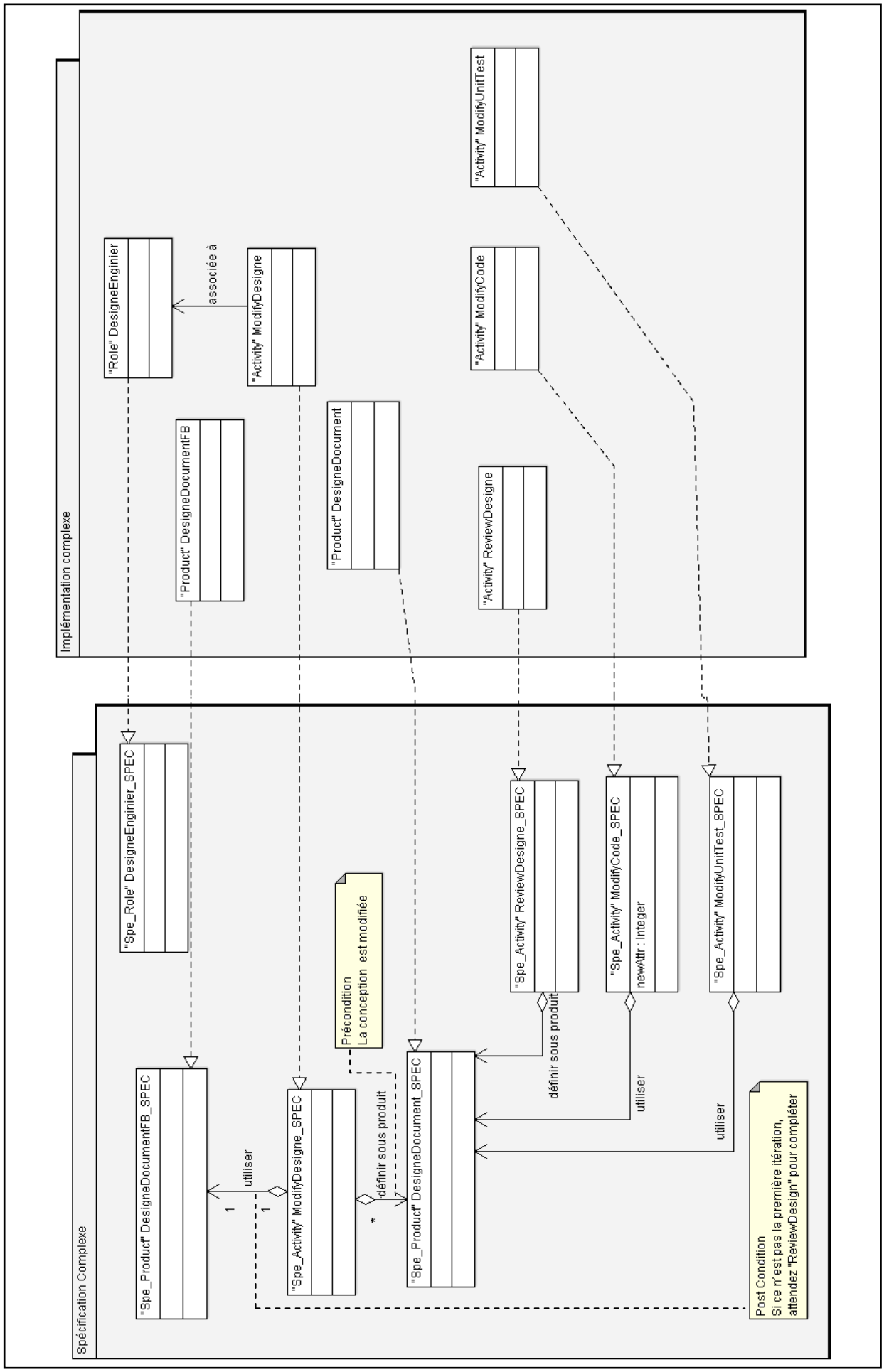


Figure7.2 - les composants concernant l'activité ''*Modify design*''

3.2 - FORMALISATION DE L'ACTIVITE ''REVIEW_DESIGN'' EN SPEM2.0

La représentation standard d'un processus de conception sous SPEM, suit un procédé bien spécifique, il est effectué dans un mode de haut en bas afin d'atteindre un niveau correct de granularité permettant l'extraction de fragments de processus.

Ainsi, un procédé est représenté dans un premier temps comme un ensemble de composants à travers le diagramme de composants de procédé, qui permet de représenter toutes les parties d'un processus de conception avec les entrées et les sorties nécessaires.
Un composant de procédé sous SPEM2.0 comporte un certain nombre de ports qui définissent les produits nécessaires pour l'accomplissement du travail et ceux fournis par le composant. Et ceci en respectant le principe d'encapsulation ''boite noire ''.

Figure7.3 montre le diagramme UML 2 basé sur l'utilisation de stéréotypes. Il décrit le composant de procédé «Review_Design», son interface est dotée de trois (03) ports : '*DesignDocument_Modified', 'DesignDocument*_Approved' et '*DesignDocumentFB*' , qui assurent sa dépendance avec les autres composants de procédé.
Chaque composant peut être détaillé par le biais d'un diagramme d'activité (*figure 7.4*).

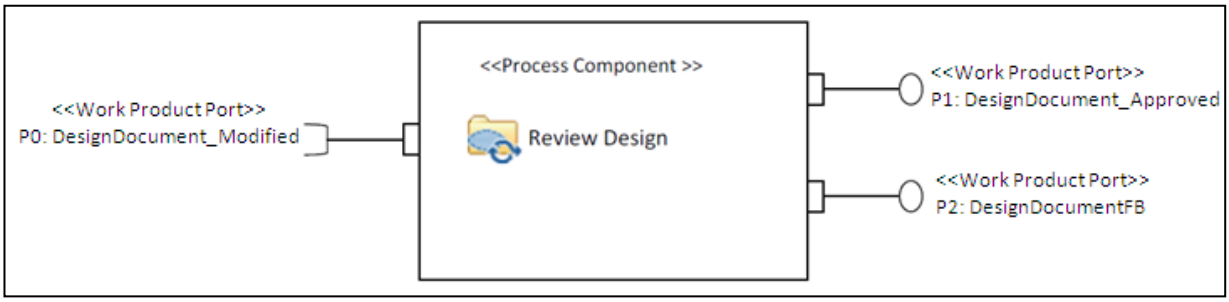


Figure 7.3- Description SPEM 2.0 de l'étape Review Design

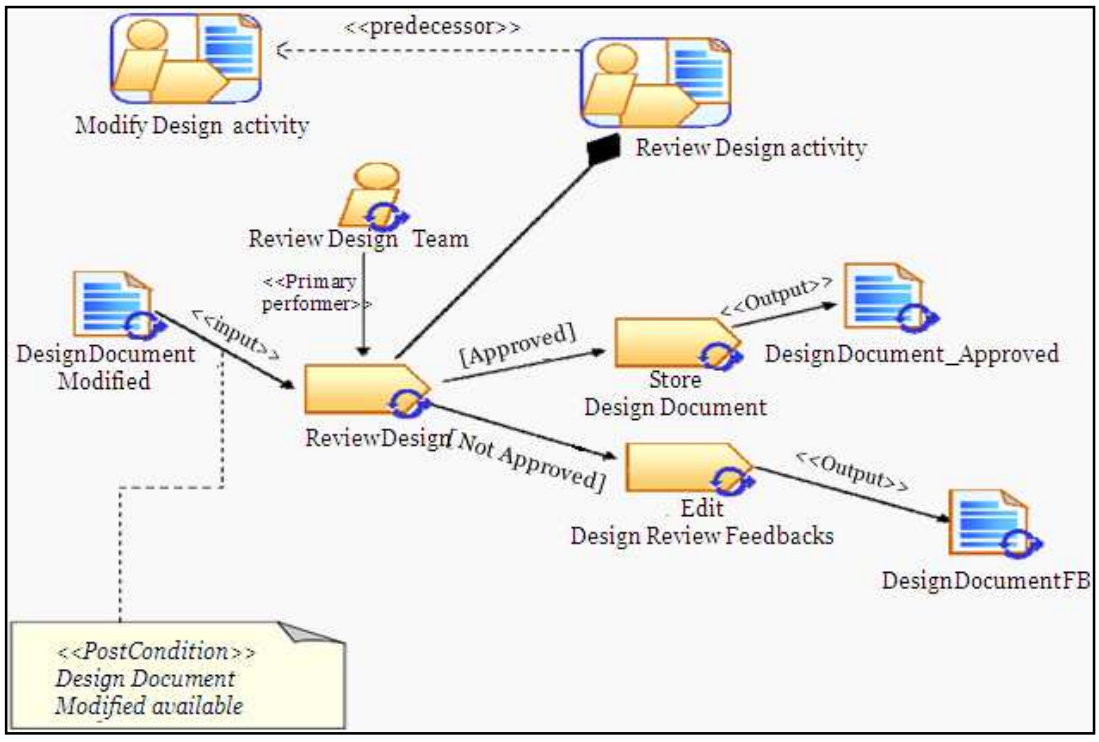


Figure7.4 – Diagramme d'activités SPEM2.0

La figure 7.4 montre l'ensemble des activités imbriquées dans le composant ''Review_Design'', la séquence des activités est indiquée par le stéréotype <<predecessor>> (*l'activité pointue est le prédécesseur de l'autre*). Chaque activité est composé de tâches, des rôles qui accomplissent les tâches et les produits d'entrée / sortie.

4- DESCRIPTION ACME CORRESPONDANTE

En se basant sur l'étude menée dans les deux chapitres précédents, nous présentons successivement, dans ce qui suit, les entités architecturales MAPL correspondantes aux deux étapes du procédé ISPW-6 ''Modify_design'' et ''Review_design'', ainsi que leurs interprétations textuelles sous ACME.

Avant d'aller plus loin, on rappelle qu'un procédé se traduit sous MAPL par :

- Un ensemble de composants élémentaires et des ports associés, ils décrivent les activités, les produits, et les rôles.
- Un ensemble de connecteurs et des rôles associés, pour identifier les interactions inter-composants.
- Une configuration qui représente le procédé global.
- Un ensemble de propriétés pour interpréter principalement les assertions relatives au procédé, ainsi, qu'un ensemble de contraintes et de type de données élémentaires.

4.1 - DESCRIPTION ACME DE L'ETAPE ''*MODIFY_DESIGN*''

Un composant complexe sous RHODES correspond uniquement à sa spécification, ces deux notions sont considérées comme identiques. Ainsi, un composant Complexe/Spécification complexe se traduit en un composant ACME qui :

- Comporte un certain nombre de ports, représentants des spécifications élémentaires incluses dans la spécification complexe.
- Est dotée d'une représentation interne se composant :
    - D'un certain nombre de propriétés, qui interprètent notamment, les assertions (pré-condition et post-condition) qui sont exploitées pour maintenir l'ordre de réalisation des activités. Pour décrire ces assertions, ACME fournit une syntaxe basée sur la logique des prédicats du premier ordre (FOPL).
    - D'un fichier externe pour conserver son implémentation.
    - Une représentation qui donne une vue détaillée sur la conception interne du composant.

La figure 7.5 schématise la description ACME de l'activité ''Modify_Design'', initialement décrite en RHODES et donne les descriptions textuelles correspondantes.

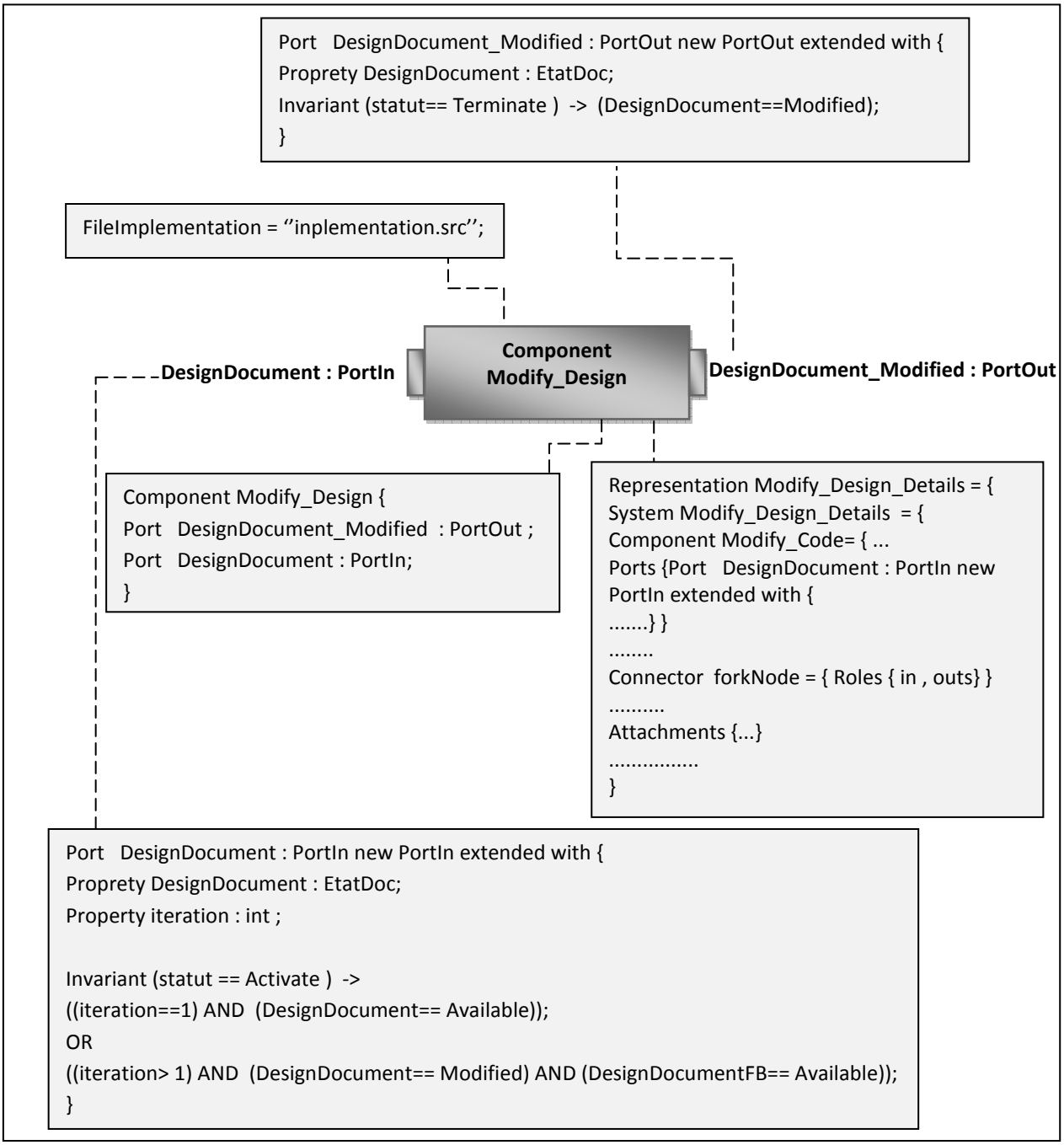


Figure7.5- Description ACME de l'activité ''Modify_Design''

4.2- DESCRIPTION ACME DE L'ETAPE ''R*EVIEW_DESIGN*''

En se basant sur les résultats du mapping SPEM2.0ToACME, nous avons traduit le ProcessComponent ''*Review_design*'' en un *Component* ACME comportant trois ports : *DesignDocumentFB, DesignDocument_Modified* et *DesignDocument_Approved,* qui correspondent respectivement aux ports SPEM2.0 définis précédemment. En plus, il définit

un certain nombre de propriétés qui donnent principalement : une représentation internant les détailles de conception, décrites auparavant (*figure 7.4*) par un diagramme d'activité SPEM2.0.

La figure 7.6 illustre la description ACME de l'étape ''*Review_design*'', initialement décrite en SPEM2.0.

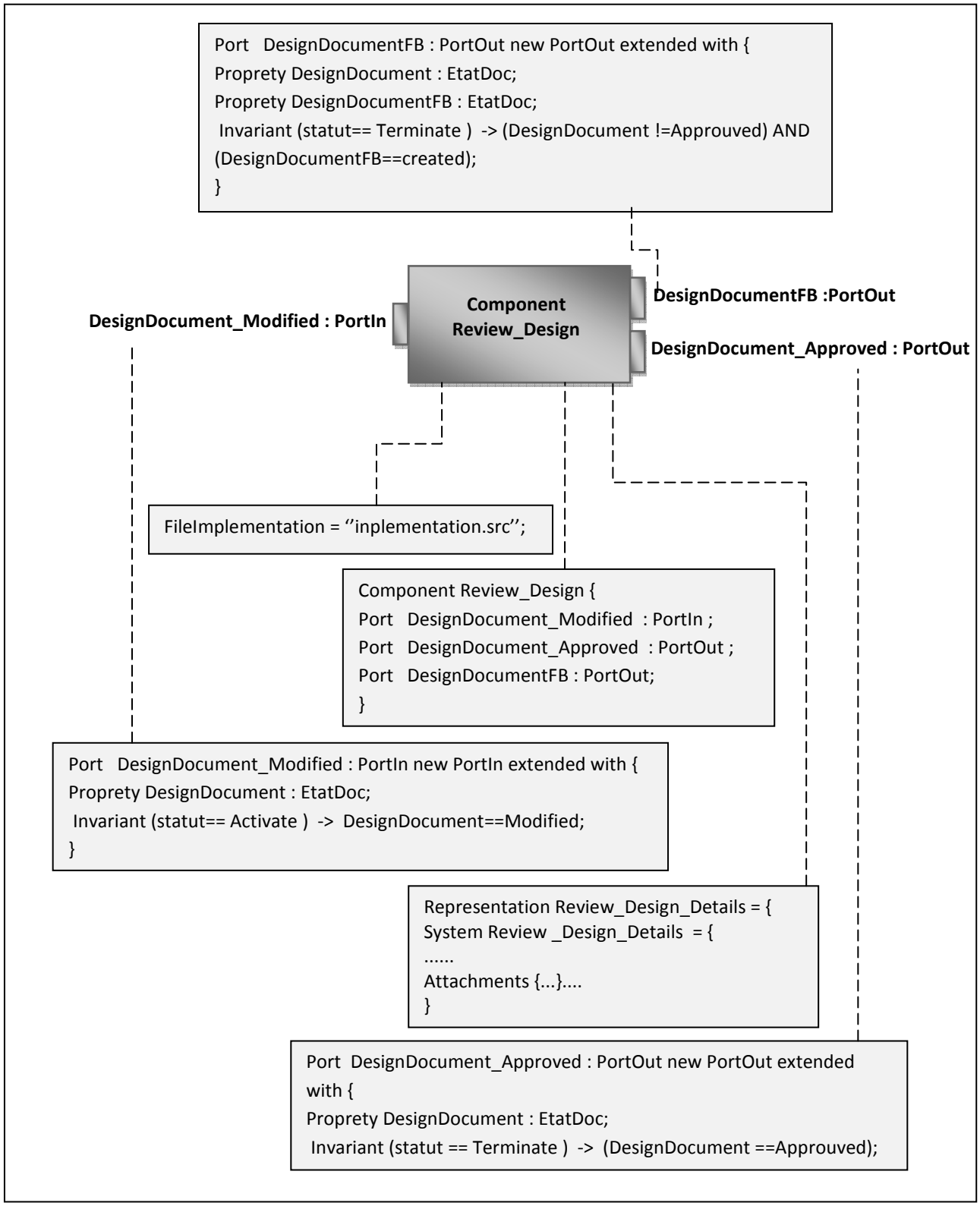


Figure 7.6- Description ACME de l'activité ''Review_Design''

4.3- ASSEMBLAGE

Pour décrire le procédé complet, dans notre cas il s'agit de l'assemblage des deux activités choisis, on a utilisé les configurations ACME (*system*). Qui définissent les propriétés architecturales de connectivité et de conformité aux heuristiques de conception.

Le listing 7.2 donne un exemple de configuration qui modélise une partie du procédé ''ISPW-6''. Ou on trouve des définitions de : composants, de ports, de connecteurs, de rôles, des analyses architecturales. Ainsi que des attachements qui décrivent l'assemblage des différentes entités du procédé.

```
1   //Configuration  ACME
2   System ISPW-6Example  {
3   // déclaration de quelques type de base
4   Property Type name = String ;
5   Property Type Descriptor =  String ;
6   Property Type SafeBoolean = Enum {yes, no } ;
7
8   // une énumération de type de port pour distinguer les interfaces requises de celles offertes
9   Property Type Interface = Enum {Required,  Available} ;
10
11  //Définition des états  d'activités
12  Property Type Statut = Enum {Activate, Terminate} ;
13
14  //Définition des états du document
15  Property Type EtatDoc = Enum { Available,Reviewed,Created,Modified, Approuved} ;
16
17  // Definition de type de port
18  Port type PortIn = {
19  Propoerty name : names ;
20  Propoerty Descriptor :  Descriptor ;
21  Propoerty Condition : SafeBoolean ;
22  Propoerty Interface : Interface = Required;
23  }
24  Port type PortOut= {
25  Propoerty name : names ;
26  Propoerty Descriptor :  Descriptor ;
27  Propoerty Condition : SafeBoolean ;
28  Propoerty Interface : Interface = Available;
29  }
30  …
31  //Déclaration composants / connecteurs / ports and roles (cf Figures  7.5 et 7.6)
32  …
33  // Définition de la topologie du système
34  Attachments {
35  ModifyDesign. DesignDocumentModified   to   Assemblage .r1  ;
36  ReviewDesign .  DesignDocumentModified:   to   Assemblage. r2 ;
37  …}
```

Listing 7.2 - Configuration ACME pour décrire le procédé ISPW-6

Le déroulement d'un procédé se modélise sous ACME, par des connecteurs qui permettent d'affiner les relations de précédences entre les activités, et de vérifier la cohérence des

produits manipulés et des rôles requis pour chaque activité. Nous avons définis aussi, des règles qui gouvernent ces interactions sous forme de ''*règles d'analyses architecturales*''.

Dans l'exemple traité dans ce document, on remarque que l'étape ''Review_design '' est un successeur de l'étape ''Modify_design''. Ainsi, on a défini un connecteur d'assemblage binaire, présenté dans la figure 7.7, qui relie les deux activités.

✍ *D'autres connecteurs relient les deux activités ''Modify_design'' et ''Review_design'' dans le procédé ISPW-6, cependant pour des raisons de clarté, on a choisi de ne présenter que le connecteur binaire.*

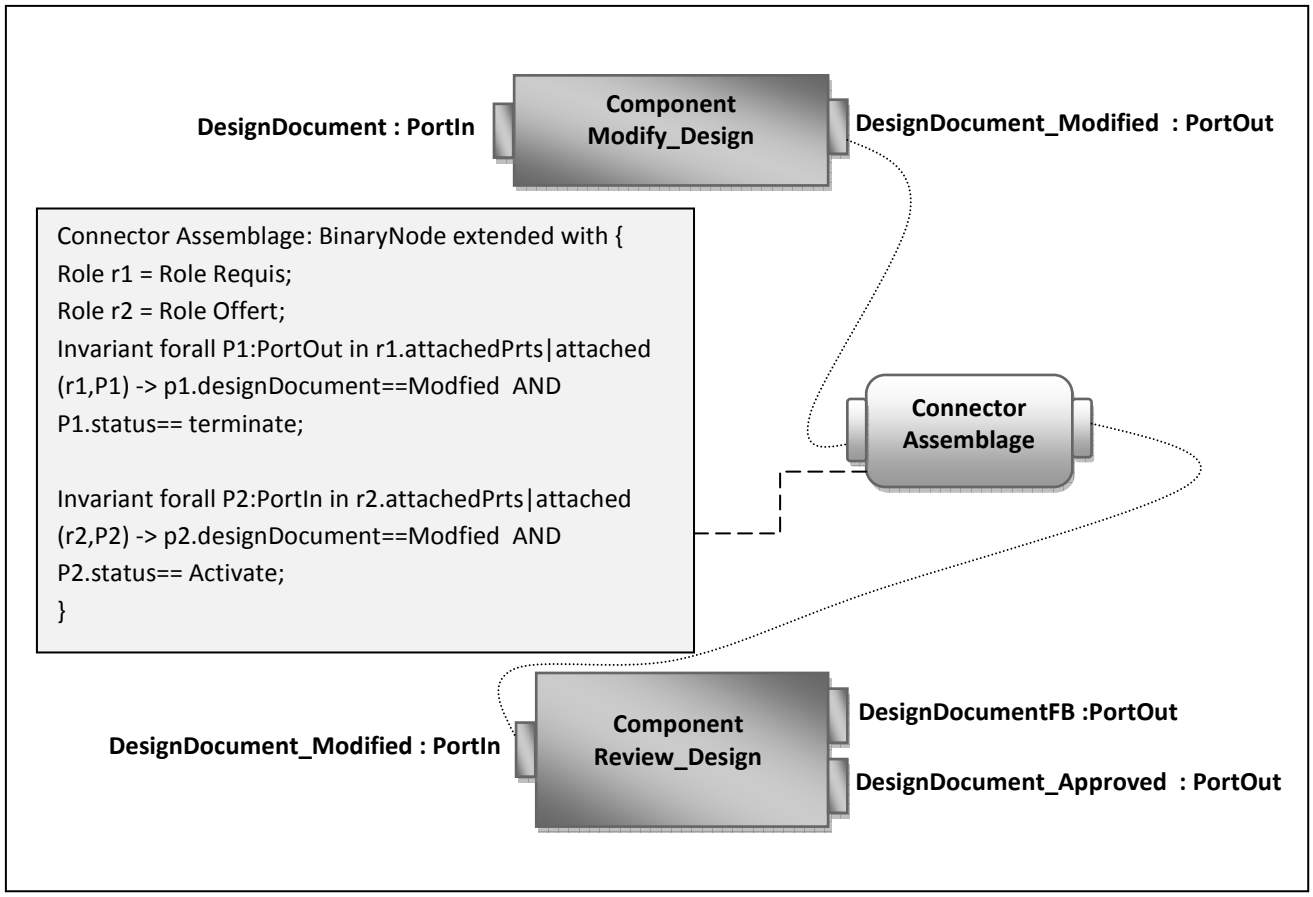


Figure7.7- Exemple de connecteur d'assemblage ACME

5- SYNTHESE

Nous avons présenté dans cette partie, une évaluation du modèle MAPL à travers le procédé ISPW-6, où nous avons pu assembler des briques de procédé, décrites par des PML hétérogènes au sein d'un modèle architectural commun.

En effet, le modèle MAPL sépare clairement le niveau conceptuel et le niveau implémentation. Il spécifie les composants de l'architecture de manière abstraite sans entrer

dans les détails d'implémentation et il définit de manière explicite les interactions entre les composants. Ce qui favorise, la réutilisabilité qui se caractérise dans la capacité à rendre générique des composants et à construire des boites noires susceptibles de fonctionner avec des langages et/ou des environnements hétérogènes.

III- ÉVALUATION PAR UNE ETUDE COMPARATIVE

Dans cette section, nous allons positionner notre travail par rapport aux modèles de procédé, qu'on a étudiés dans le chapitre 2.

Notre étude comparative, illustrée dans le tableau 7.1, s'articule autour de cinq critères de comparaison déjà définis (*chapitre 2, section 4.4*) :

	PYNODE	OPC	RHODES	SPEM 2.0	MAPL
Réutilisation	Interne au système			Comportement externe	Composant réutilisable sous différents environnement
Cohérence	interfaces de composants	Le schéma permet de décrire les entités de base et les interactions entre eux	- Assertions (*pré/post condition*) - Règles de guidage méthodologique	Pris en charge par divers mécanismes : assertions (*prés/post condition*), règle de guidage, des contraint,…	- Les interfaces de composants et des connecteurs. - Architecture donne une vue globale du procédé
Mécanismes d'interaction	vues d'observation et les vues exécution	message, évènement et méthode	les méthodes et les schémas de réalisation (*heuristique de développement*)	Connecteurs implicites	Connecteurs explicite
Hétérogénéité	Syntaxique	Syntaxique	Homogène	Syntaxique	Syntaxique Sémantique

Tableau 7.1- Étude comparative entre les approches centrées procédé et MAPL

Le tableau 7.1 décrit principalement les caractéristiques de MAPL, conformément aux critères de comparaison :

- *Réutilisation* : Les composants et les connecteurs sont spécifiés de manière abstraite. Ceci signifie que l'on pourra, définir des composants totalement indépendants des

plates-formes techniques, d'assurer leurs intégration, et par conséquence de permettre leurs réutilisation sous différents environnements.

- *Hétérogénéité :* MAPL est définit sous différents niveaux d'abstraction, il garantit ainsi :
  - D'un coté, une homogénéisation des concepts hétérogènes (hétérogénéité sémantique) au sein d'un modèle pivot,
  - Et d'un autre coté, une manipulation des formalismes hétérogènes (hétérogénéité syntaxique), en considérant les interfaces des composants comme des entités de première classe et en encapsulant leurs implémentations.

- *Cohérence* : La vérification de la cohérence du procédé est assurée par les interfaces des composants et des connecteurs et par la description architecturale globale du procédé.

- *Mécanisme d'interaction* : MAPL permet de définir de manière explicite les dépendances entre les différents composants d'un procédé, en spécifiant leur protocole d'interaction. Et d'aboutir, par la suite, à une forte cohésion et à un couplage faible entre les composants.

VI- CONCLUSION

Dans ce chapitre, nous avons évalué MAPL, dans un premier temps, par la modélisation d'un procédé bien connu et fréquemment utilisé ''ISPW-6''. En effet, différentes approches de modélisation ont été appliqués à cet exemple, par la suite, ils ont étendu et amélioré leurs approches. Avec MAPL, nous avons réussi à modéliser les activités du processus et les questions liées au problème de base. En plus, nous avons démontré qu'avec MAPL on peut construire un procédé par simple intégration de composants de procédés réutilisables, et ceci indépendamment de toute plate forme d'exécution.

Une autre évaluation de MAPL consistait à le comparer avec les approches de modélisation centrées composant de procédé, et ceci en se basant sur un ensemble de critères, que nous avons jugé important d'être respecter par les PML (*introduites dans le chapitre 2*), et nous avons conclus que MAPL réussi à réaliser la majorité d'entre eux.

CONCLUSION GENERALE ET PERSPECTIVES

CONCLUSION GENERALE ET PERSPECTIVES

I- RAPPEL DE LA PROBLÉMATIQUE

La complexité croissante des systèmes logiciels et leur évolution de plus en plus rapide ont motivé un intérêt accru pour les modèles et méthodes de réutilisation. Dans ce mémoire, nous nous sommes focalisé sur la réutilisation des procédés logiciels, qui se trouve confrontée aux mêmes défis que ceux inhérents à la production du logiciel **[OST, 87]**.

La réutilisation des procédé a été abordée selon une approche composant qui consiste à segmenter, rationaliser, encapsuler et plus généralement modulariser les procédés de développement.

Cependant, malgré une réelle avancée, force est de reconnaître que les problèmes liés aux développements et à l'usage des composants ne sont pas encore complètement résolus.
En effet, de l'étude menée sur les ECP (**[CREb, 97] [BEL, 96] [GARb, 99]** et **[SPE, 07])** on a pu constater, qu'il n'y a pas de consensus sur la définition d'un composant procédé (malgré les efforts de standardisation) et les modèles de composants de procédé proposés restent hétérogènes tant au niveau des concepts que des langages de spécification. Ainsi, leur réutilisation dans d'autres environnements devient extrêmement difficile : de nombreuses difficultés d'intégration, des interactions surprenantes…, ce qui constitue aujourd'hui l'obstacle principal à leur réutilisation.

Rendre réutilisable un composant de procédé nécessite donc, des techniques de réutilisation qui permettent une exploitation efficace et fiable du support existant, et qui offrent des solutions génériques, adaptables, intégrables et composables.

II- DÉMARCHE SUIVIE

A fin de répondre à cette problématique, nous avons proposé une solution qui se fonde sur la conception d'un modèle de procédé à base de composants architecturaux, ce qui permet de prendre en compte les descriptions de haut-niveau des systèmes complexes, et de

raisonner sur leurs propriétés à un haut niveau d'abstraction (protocole d'interaction, conformité avec d'autres architectures, évolution…).

La démarche adoptée se résume principalement dans les deux (02) étapes suivantes:

- **Unification :** Cette étape permet d'unifier les différents fragments de procédés en un modèle architectural commun. Nous avons effectué des mises en correspondances successives entre le domaine des procédés et celui des architectures sur différents niveaux d'abstraction, allant du plus abstrait vers les niveaux les plus bas.
- **Assemblage :** Pour atteindre un modèle de procédé global, cette étape propose la définition explicite des interactions entre les différents fragments de procédés unifiés.

III- SYNTHÈSE DU TRAVAIL RÉALISÉ

Les majeures contributions de ce mémoire peuvent se résumer :

- ✓ **Modularité** : L'utilisation des notions de composant et de connecteur assure une séparation claire entre la mise en œuvre des différentes entités du procédé et la communication entre elles.
- ✓ **Interopérabilité** : MAPL assure une homogénéisation des concepts hétérogènes, au sein d'un modèle pivot, et une manipulation des formalismes hétérogènes.
- ✓ **Réutilisation** : Le modèle du composant à été défini de manière à ce qu'il peut être utilisé dans différentes applications et indépendamment de toute plate-forme d'exécution, et ceci, on considérant leurs interfaces comme des entités de première classe et en encapsulant leurs implémentations.
- ✓ **Analyse** : La solution fournit des moyens d'analyse, tels que la vérification de la cohérence, la vérification de la conformité aux contraintes architecturales, …
- ✓ **Une vue complète du procédé** : La description d'une architecture globale du procédé peut être spécifiée avant de compléter la construction de ses composants.
- ✓ **Extensibilité :** La solution proposée peut être projetée vers plusieurs modèles de procédé conforme au méta-méta-modèle MOF et modèles architecturaux conforme au méta-méta-architecture MADL.
- ✓ **Abstraction :** La mise en œuvre de MAPL se base essentiellement sur la dissociation entre la syntaxe abstraite et concrète de nos données. Ce qui permet d'exprimer et de définir les concepts utilisés indépendamment de tout support de présentation des informations, favorisant ainsi, la réutilisation des modèles déjà construit.
- ✓ **Le prototype GLOBE :** Une solution flexible de transformation de modèle qui implémente une grande partie de l'étape d'unification.

IV- PERSPECTIVES

En perspective, nous prévoyons essentiellement :

- **Le raffinement du modèle architectural vers des niveaux d'abstractions plus concrets :** Consiste à développer une descriptions concrète de l'architecture en partant d'un niveau abstrait de détails vers des niveaux de plus en plus concrets pouvant aboutir à une implémentation dans un langage de programmation cible. Cette étape requiert la préservation des propriétés (structurelles, comportementales et non fonctionnelles) du modèle architectural initial.

- **Définition d'un ADL spécifique au domaine des procédés logiciels :** Nous envisageons dans un premiers temps, la définition d'un langage (ADL) spécifique au domaine des procédés logiciels. Qui sera le premier pas vers la description, l'analyse, la vérification, … des modèles de procédés sous un environnement architectural.
  Par la suite, des outils de développement, peuvent lui être associés. Et on discutera aussi, de la possibilité d'intégration de cet ADL dans ECP.

- **La vérification de la cohérence des règles de transformation :** L'incohérence des règles de transformation peut causer des erreurs lors de leurs applications. Nous avons en effet constaté que certaines erreurs se produisent parce que les règles ne sont pas conformes aux méta-modèles source et cible.

  Un exemple d'une erreur qui se présente très souvent est le fait d'essayer de lire ou d'écrire à une propriété qui n'existe pas dans le méta-modèle. En conséquence, Il est préférable que les erreurs puissent être détectées avant le temps d'exécution.

  Ainsi, nous envisageons de mener une étude afin d'adopter et d'implémenter une méthode pour la vérification des règles de transformation.

- **Projeter la solution vers d'autres modèles de procédé :** Dans ce mémoire, nous avons traité principalement la problématique de réutilisation et de l'interopérabilité des composants procédés. Cependant, Au sein d'un ECP, on peut distinguer l'existence de plusieurs catégories de PML, qui ne sont pas nécessairement basés sur les composants (objet, service…). Ainsi, notre objectif principal est d'étudier et de proposer un modèle global, générique, et efficace qui offre un niveau d'abstraction élevé permettant l'exploitation des différents modèles de procédés.

  Cet objectif peut être décomposé en plusieurs sous objectifs, on citera notamment :
  - L'abstraire et la composition des entités du procédé.
  - La représentation des connaissances au sein d'un modèle global.
  - La mise en œuvre de la réutilisation des entités (atelier de développement, …).

BIBLIOGRAPHIE

Bibliographie

[ACC, 02] : *Projet ACCORD,''Etat de l'art sur les Langages de Description d'Architecture (ADL)'', RNTL, 2002.*
Source : http://www.infres.enst.fr/projets/accord/lot3/lot_3_5.pdf

[ACM] : *"The Acme Architectural Description Langage", Source : http://www-2.cs.cmu.edu/~acme/*

[ACMa] : *'' The Acme version 2.0 BNF ''.*
Source : http://www.cs.cmu.edu/~acme/html/ArmaniParser.html

[ADM] : *''Architecture Description Markup Language'',*
Source: http://www.opengroup.org/architecture/adml/adml_home.htm.

[AES] : *"Aesop, Software Architecture Design Environment Generator",*
Source : http://www-2.cs.cmu.edu/Groups/able/aesop/aesop_home.html.

[AMB, 98] : *Ambler S.W, ''Process Patterns: Building Large-Scale Systems Using Object Technology''. New York: SIGS Books/Cambridge University Press, 1998.*

[AMI, 99] : *Mahfoud Amiour. ''Vers une Fédération de composants interopérables pour les environnements centrés procédés logiciels''. Thèse de doctorat, Université Joseph Fourier, Grenoble, 1999.*

[AOU ,10] : *Aoussat Fadila, Ahmed-Nacer Mohamed, Oussalah Mourad, ''New Approach for Software Processes based on Software Architectures'', 12th International Conference on Enterprise Information Systems, ICEIS (1),p.366-369, Portugal, 2010.*

[ARO, 12] : *Allaeddine Arous,'' Conception et Implémention d'un Système de Transformation de Modèle'', Mémoire master2, USTHB, Juin 2012.*

[BEL, 08] : *Meriem Belguidoum, ''Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels'', thèse de doctorat, école nationale supérieure des télécommunications de Bretagne, 2008.*

[BEL, 96] : *Noureddine Belkhatir, Denis Avrilionis, Pierre-Yves Cunin . J.-C. Derniame. ''Improving Software Process Modelling and Enactment Techniques''. Proceedings of the 5th European Workshop on Software Process Technology, LNCS, 1996.*

[BEN, 07] : *R. Bendraou, A.Sadovykh, M.P.Gervais, & X.Blanc, ''Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach '', in Proceedings of the 33rd EUROMICRO Conference of Software Engineering Advanced Application (SEAA). IEEE Computer Society Press,2007.*

[BEN,05] : *Bendraou R., Gervais M.P. & Blanc X., ''UML4SPM: A UML 2.0-Based metamodel for Software Process Modeling '', in Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05), Montego Bay, LNCS, Vol. 3713, p.17-38. Jamaica, Oct. 2005.*

[BENa, 07] : *Réda Bendraou , ''UML4SPM : Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle '', Thèse de doctorat, l'université Pierre et Marie Curie - Paris VI, Paris, 2007.*

[BEZ, 04] : *Jean Bézivin. '' Sur les principes de base de l'ingénierie des modèles '', RSTI-L'Objet, Volume 10, N° 4, p.145-157, Octobre 2004.*

[BEZ, 05] : *Jean Bézivin & Ivan Kurtev. ''Model-based Technology Integration with the Technical Space Concept.''. Proceedings of the Metainformatics Symposium. Springer-Verlag, 2005.*

[BLA, 05] : *Blanc Xavier, ''MDA en action Ingénierie logicielle guidée par les modèles'', édition Eyrolles, Paris, 2005.*

[BOE, 96] : *Barry Boehm, Steven Wolf. "An open architecture for software process asset reuse'', 10th International Software Process Workshop (ISPW '96), p. 2, 1996.*

[BOR,08] : *B.T. Borsoi, J.L.R. Becerra., ''A Method to Define an Object Oriented Software Process Architecture.'', ASWEC'08, 19th Australian Conference on Software Engineering, Perth, WA, p. 650-655, 2008.*

[BRU, 94] : *R.F. Bruynooghe, R.M. Greenwood, I. Robertson, J. Sa, R.A. Snowdon, & B.C. Warboys.*"PADM: Towards a Total Process Modelling System", (A. Finklestein, J. Kramer, & B. Nuseibeh, editors,) , Software Process Modelling and Technology .Research Studies Press, p.293–334, 1994.

[CAN, 94] : *Gerome Canals, Nacer Boudjlida, Jean-Claude Derniame, Cladue Godart, & Jaques Lonchamp. "ALF: AFramework for Building Process-Centred Software EngineeringEnvironments.", Software Process Modellingand Technology,1994.*

[COM, 08] : *Benoît combemale, ''Approche de métamodélisation pour la simulation et la vérification de modèle, Application à l'ingénierie des procédés'', Thèse de doctorat, université de toulouse, 11 juillet, 2008.*

[CON, 00] : *R. Conrad, D. Scheffner, & J.C. Freytag,'' XML conceptual modeling using UML''. Proceedings of the19th International Conference on Conceptual Modeling (ER'2000). October 2000.*

[CREa, 97] : *Xavier cregut, Bernard Coulette.''PBOOL : an Object-Oriented Language for Definition and Reuse of Enactable Processes'', Softwar concepts & tool, Vol 18 N2. Juin 1997.*

[CREb, 97] : *Xavier cregut, Bernard Coulette.''RHODES : un environnement d'assistance centré sur le procédé de développement''.Journées du GDR programmation. Rennes, 1997.*

[CUR, 90] : *W. Curtis, M. I. Kellner & J. Over. "Process Modelling". Communication of ACM, p. 75-90. 1992.*

[DAI,08] : *F.Dai, T.Li, N.Zhao, Y.Yu, B.Huang , ''Evolution Process Component Composition Based on Process Architecture.'', International Symposium on Intelligent Information Technology Application Workshops. Volume 00, P. 1097-1100. 10.1109/IITA.Workshops . 2008.*

[DAN, O5] *Thu Tran Dan, Hanh Nhi Tran, Thuy Dong Thi Bich, Bernard Coulette, Xavier Crégut. ''Topological properties for characterizing well-formedness of process components''. Software Process: Improvement and Practice, Wiley Interscience, Vol. 10 N. 2, p. 217-247, 2005.*

[DAR] : *"The Darwin Architecture description Langage", Source : http://www.doc.ic.ac.uk/~igeozg/Project/Darwin/.*

[DEA, 04] : *Karine Macedo De Amorim, ''Modélisation d'aspects qualité de service en UML : application aux composants logiciels '', thèse de doctorat, université de Rennes 1, 17 mai 2004.*

[DIA, 01] : *Diaz Michel,''Les Réseaux de Petri – Modèles fondamentaux'', Hermes Science Publications : Paris. 2001.*

[DIN, 02] : *E. Di Nitto, L. Lavazza, M. Schiavoni, E. Tracanella, M. Trombetta, '' Deriving executable process descriptions from UML'', Proceedings of the 24th Inter. Conf. on Software Engineering (ICSE'02), Orlando, ACM Press. Florida, 2002.*

[DTD] : *''Document Type Definition''.*
Source : http://www.w3.org/QA/2002/04/valid-dtd-list.html.

[ECL] : *''Eclipse Indigo''. Source: ''http:// www.eclipse.org/*

[ELB, 06] : *Ghizlane El Boussaidi & Hafedh Mili, ''Les langages de description d'architectures'', rapport technique, Université du Québec à Montréal, 2006.*

[EMF] : *''The Eclipse Modeling Framework (EMF) Overview'', Source: http://www.eclipse.org/modeling/emf/docs/#overviews.*

[FAV, 06] : *Jean-Marie Favre, Jacky Estublier , Mireille Blay-Fornarino, ''L'Ingénierie Dirigée par les Modèles : au-delà du MDA'', Informatique et Systèmes d'Information, Hermes Science, édition Lavoisier, février 2006.*

[FEI, 88] : *Gail E. Kaiser, Peter H. Feiler, & Steven S. Popovich."Intelligent Assistance for Software Development and Maintenance. », 1EEE Software, Mai, 1988.*

[FLE, 06] : *Franck Fleurey,'' Langage et méthode pour une ingénierie des modèles fiable '', Thèse de Doctorat, Université de Rennes 1, 2006.*

[FRO, 05] : *Emmanuel Jausseran, Agnès Front, Jean-Pierre Giraudin,Dominique Rie,'' Analyse des demarches de type xup methodes de developpement centrees composants'', rapport de projet, Projet de la Région Rhône-Alpes 2003-2005,Mars 2005.*

[GAL, 00] : *Ma. Lizbeth Gallardo, ''Une approche à base de composants pour la modélisation des procédés logiciels'', DEA préparé dans l'équipe ADELE, au laboratoire LSR, Soutenance à Grenoble, Le 20 Septembre 2000.*

[GARa , 99] : *Gary, K. & Lindquist, T. "Distributed Architectures for Process Component Support". Proceedings of the 5th Intl. Conf. on Information Systems Analysis and Synthesis (ISAS'99),invited session on Process Support for Distributed Team-based Software Development (PDTSD'99), août, 1999.*

[GARb, 99] : *Kevin A. Gary et Timothy E. Lindquist,'' Cooperation Process Components'', Proceeding :COMPSAC '99 23rd International Computer Software and Applications Conference, IEEE Computer Society Washington, p.218,USA,1999.*

[GHI, 06] : *Ghizlane El Boussaidi et Hafedh Mili, ''Les langages de description d'architectures'', rapport technique, Université du Québec à Montréal, 2006.*

[HAC, 06] : *Hachani Ouafa,''Patrons de conception à base d'aspects pour l'ingénierie des systèmes d'information par réutilisation'', thèse de doctorat, université joseph fourier – grenoble I, 4 Juillet 2006.*

[HAN, 07] : *Hanh Nhi Tran. ''Modélisation de procédés logiciels à base de patrons réutilisables''. Thèse de doctorat, Université de Toulouse-le-Mirail, novembre 2007.*

[INO, 89] : *K. Inoue, T.Ogihara , T.Kikuno , & K.Torii . ''A Formal Adaptation method for process descriptions'', Proc of ACM/IEEE ICSE, 1989.*

[JAV] : *''JAVA Sun Microsystems''. Source : http://www.java.com/.*

[JOU, 05] : *Frédéric Jouault et Ivan Kurtev,'' Transforming Models with ATL''. Satellite Events at the MoDELS 2005 Conference, Proceedings of the Model Transformations in Practice Workshop, volume 3844 de Lecture Notes in Computer Science, p. 128–138, Montego Bay, Jamaica, 2005.*

[JOU, 06] : *Frédéric Jouault, ''Contribution à l'étude des languages de transformation de modèles'', Thèse de doctorat, Université de Nantes, Septembre 2006.*

[KAB, 09] : *Mohammed Issam Kabbaj, ''État de l'art sur l'adaptation dynamique des procédés de développement de logiciels'', Rapport IRIT (Institut de Recherche en Informatique de Toulouse), France, 2009.*

[KAT,89] : *T. Katayama, ''A hierarchical and functional software process description and its enaction''. Proc of ACM/IEEE ICSE, 1989.*

[KEL, 91] : *M. I. Kellner, P. H. Feiler, A. Finkelstein, T. Katayama, Leon J. Osterweil, M. H. Penedo, & H. D. Rombach.'' ISPW-6 Software Process Example''. Proceedings of the First International Conference on Software Process, pp. 176-186, IEEE Computer Society Press, 1991.*

[KER] : *''KerMeta : metamodeling kernel'', Source: http://www.kermeta.org/*

[KLE, 03] : *A. Kleppe & J. Warmer. ''MDA Explained. The Model Driven Architecture: Practice and Promise''. Addison-Wesley, 2003.*

[KUD, 03] : *T. Kudrass & T. Krumbein, ''Rule-Based Generation of XML DTDs from UML Class Diagrams.'', Proceedings of ADBIS. p. 339-354, 2003.*

[LAV, 94] : *Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, & Luigi Lavazza, "SPADE: An Environment for Software Process Analysis, Design, and Enactment.", Software Process Modelling and Technology (A.Finkelstein, J. Kramer, & B. Nusibeh, editors), John Wiley & Sons Inc, 1994.*

[LEG, 05] : *Fabrice Legond-aubry, ''Un modèle d'assemblage de composants par contrat et programmation orientée aspect'', thèse de doctorat, conservatoire national des arts et métiers, 2005.*

[LEM,00] : *Richard Lemesle.'' Techniques de Modélisation et de Méta-modélisation''. Thèse de Doctorat, Ecole Centrale de Nantes, France, 2000.*

[LEY, 04] : *Fabien Leymonerie, ''ASL : un langage et des outils pour les styles architecturaux.'', Thèse de Doctorat, Université de Savoie, 15 décembre 2004.*

[LIU, 95] : *R.Conradi & C.Liu, ''Process modelling languages: One or many? '', EWSPT '95 Proceedings of the 4th European Workshop on Software Process Technology, Pages 98 - 118, 1995 .*

[LON, 92] : *J Lonchamp, C Godart, J.C Derniame, ''Les environnements intégrés de production de logiciel'', Technique et science informatiques, volume 11- n°1, p. 31–95, 1992.*

[MAC] : *''Mac system''. Source: http://www.apple.com/fr/mac/*.

[MAR, 02] : *Raphael Marvie, ''Séparation des préoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants'', Thèse de Doctorat, Université DES SCIENCES ET TECHNOLOGIES DE LILLE, 9 décembre,2002.*

[MCL, 76] : M. McIlory, ''Mass-Produced Software Components '', NATO Software Engineering Conference Report, p. 79-85. *Allemagne* , Octobre, 1968.

[MED, 00] : *Nenad Medvidovic, Richard N. Taylor, ''A Classification and Comparison Framework for Software Architecture Description Languages.'', IEEE Transactions on Software Engineering, Vol. 26, No. 1, p.70-93, 2000.*

[MED, 97] : *Nenad Medvidovic, Richard N. Taylor, ''A Framework for Classifying and Comparing Architecture Description Languages.'', ESEC / SIGSOFT FSE,p. 60-76, 1997.*

[MEG, 04] : *Karim Megzari, '' REFINER : Environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture'', Thèse de Doctorat, Université de Savoie, 17 décembre 2004.*

[MEN, 06] : *Tom Mens & Pieter Van Gorp, ''A Taxonomy of Model Transformation'', ENTCS, 2006.*

[MOF, 08] : ''*OMG Meta Object Facility (MOF) Core Specification*''. *Version 2.0. 2008.*

[MOF, 11] : ''*OMG Meta Object Facility (MOF) Core Specification*''. *Version 2.4.1. 2011.*

[MOS, 02] : *Mohammed Amine Mostefai, ''résolution des problèmes d'interopérabilité d'exécution dans les fédérations de composants de procédés logiciels '', mémoire de magister, USTHB, ALGER, 2002.*

[NAR, 05] : *K. Narayanan & S. Ramaswamy. ''Specifications for Mapping UML Models to XML Schemas''. Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica, 2005.*

[OST, 87] : *Leon Osterweil. ''Software processes are software too ''.University of Colorado Boulder, ACM, Colorado USA, 1987.*

[OST, 97] : *S. Sutton Jr. & L. J. Osterweil, "The Design of a Next-Generation Process Language", Proceedings of the Joint 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on the Foundation of Software Engineering ESEC/FSE'97. Vol. 1301, Springer, p. 142-158, 1997.*

[OUS, O5] : *Mourad Oussalah,''Ingénierie des composants : Concepts, techniques et outils''. Edition Vuibert, 04/07/2005.*

[PEL, 01] : *Mikaël Peltier, Jean Bézivin & Gabriel Guillaume, '' MTRANS : Ageneral framework, based on XSLT, for model transformation'' . Workshop on Transformations in UML (WTUML), Italie, 2001.*

[PELa, 00] : *Mikaël Peltier, François Ziserman & Jean Bézivin ; ''On levels of model transformation '', XML Europe Conference, Paris, France ; Juin 2000.*

[PELb, 00] : *Mikaël Peltier, François Ziserman & Jean Bézivin, ''Concrete and Abstract Spaces in Model Engineering'', Information Systems Engineering : Conceps and Industrial Applications World (ISICA), Multiconference on Systemics, Cybernetics and Informatics, 2000.*

[RAP] : *"The Stanford Rapide Project", Source: http://pavg.stanford.edu/rapide/.*

[RAT, 05] : *Olivier Ratcliffe, ''Approche et environnement fondés sur les styles architecturaux pour le développement de logiciels propres à des domaines spécifiques'', Thèse de Doctorat, Université de Savoie, 16 décembre 2005.*

[REV, 05] : *Jérôme Revillard, ''Approche centrée architecture pour la conception logicielle des instruments intelligents'', Thèse de Doctorat, Université de Savoie, 15 décembre, 2005.*

[ROU, 02] : *N. Routledge, L. Bird, L. & A. Goodchild, ''UML and XML schema''. Proceedings of the 13th Australasian Database Conference. P. 157-166. Melbourne, Australie. 2002.*

[SAM, 97] : *Johannes Sametinger ; ''Software engineering with reusable components'', Edition Springer, 272 pages, 1997.*

[SEL, 01] : *Firesmith, D. Henderson-Sellers, ''The OPEN Process Framework. A Introduction'', Addison-Wesley, ISBN 0-201-67510-2, December 2001.*

[SER, 03] : *Abdaziz Gacemi et Abdelhak Seriai. ''Réutilisation : concepts et techniques. '', Technical Report 2003-4-2. École des Mines de Douai, France. 2003.*

[SHA, 94] : *Israel Ben-Shad & Gail Kaiser. "A Paradigm for Decentralized Process Modeling and its Realization inthe Oz Environment.", Proceedings of the 16 th International Conference on Software . Engineering, 1994.*

[SIN, 03] : *J. Singh.'' Mapping UML Diagrams to XML''. Thèse de master, Jawaharlal Nehru University, New Delhi, 2003.*

[SME, 05] : *Adel Smeda, Mourad Oussalah, & Tahar Khammaci .''MADL: Meta Architecture Description Language''. Proceedings of the 2005 Third ACIS Int'l Conference on Software Engineering Research, Management and Applications (SERA'05). 2005.*

[SPE, 08] : *''OMG SPEM2.0. Software Process Engineering Metamodel''. OMG document, final adopted specification, 2008.*

[SPE, 05] : *''OMG SPEM 1.1, Software Process Engineering Metamodel '', OMG document, final adopted specification, 2005.*

[SPE, 07] : *''OMG SPEM2.0. Software Process Engineering Metamodel''. OMG document, final adopted specification, 2007.*

[STE, 08] : *Dave Steinberg,''Fundamentals of the Eclipse Modeling Framework'', IBM Rational Software,Toronto,, EMF Committer, Canada, 17 mars 2008.*

[STR, 00] : *Alfred Strohmeier, support de cours, laboratoire de génie logiciel, école polytechnique fédérale de Lausanne, 13 mars 2000.*

[SUN] : *'' Sun Microsystems''. Source : http://www.sun.com/.*

[SUT, 95] : *Sutton Jr., S.M., Heimbigner D., and Osterweil L. J. ''APPL/A: A language for software-process programming''. ACM Transaction on Software Engineering and Methodology, 4(3):221–286. 1995 .*

[SZY, 98] : *Clemens Alden Szyperski,''Component Software: Beyond Object-Oriented Programming.'', Addison-Wesley / ACM Press, ISBN 0-201-17888-5, 411 pages, 1998.*

[TAY, 88] : *Richard N. Taylor, Frank C. Belz, Lori A. Clarke,Leon J. Osterweil, Richard W. Selby, Jack C. Wileden,Alexander L. Wolf, and Michal Young. "Foundations for the Arcadia Environment Architecture.",Proceedings of SIGSOFT'88: Third Symposium on Software Development Environments, 1988.*

[THO, 08] : *F.Thomas, ''Contribution à la prise en compte des plates-formes logicielles d'exécution dans une ingénierie générative dirigée par les modèles'', thèse doctorat, Université d'Evry, novembre, 2008.*

[THU, 01] : *M. Tran Dan Thu. ''Formalisation et mise en œuvre de la notion de composant de procédés logiciels''.thèse de doctorat. INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE. Novembre 2001.*

[UMLa, 11] : ''*OMG Unified Modeling Language (UML), Superstructure*''. *Version 2.4.1. 2011.*

[UMLb, 11] : ''*OMG Unified Modeling Language (OMG UML), Infrastructure*''. *Version 2.4.1. 2011.*

[UNI] : *''Unix System''. Source: http://www.unix.org/*.

[VOJ, 10] : *Didier Vojtisek, '' ECORE MDK MANUAL: Ecore Model Development Kit (MDK)'', Manuel, 2010.*

[W3C] : *''World Wide Web Consortium''. Source: http://www.w3.org/*.

[WIL, 99] : *Wile David, ''AML: an Architecture Meta-Language.'', Proceedings ASE 99, IEEE Computer Society Press, p. 183-190, 1999.*

[WIN] : *''Microsoft windows system''. Source: http://windows.microsoft.com/fr-fr/windows/home*.

[XAD] : *''A Highly Extensible Architecture Description Language for software and system'', Source: http://www.isr.uci.edu/projects/xarchuci/index.html*.

[XML] : *''Extensible Markup Language''. Source: http://www.w3.org/XML/*.

[XPA] : *''XML Path language''. Source : http://www.w3.org/TR/Xpath/.*

[XSL] : *''EXtensible Stylesheet Language''. Source: http://www.w3.org/Style/XSL/*.

[ZUH, 01] : *Kamal Zuhairi Zamli, et Peter Lee, '' Taxonomy of Process Modeling Languages.'', Computer Systems and Applications, ACS/IEEE International Conference, 2001*.

ANNEXES

Annexe 1

DESCRIPTION DE LA TRANSFORMATION SPEM2.0 VERS ACME

Annexe 1

DESCRIPTION DE LA TRANSFORMATION SPEM2.0 VERS ACME

I- DTD SPEM2.0

```
<!ELEMENT Process (ProcessComponent|WorkProductDefinition|
WorkProductPortConnector)*>
<!ELEMENT ProcessComponent (WorkProductPort|(Activity?))*>
<!ATTLIST ProcessComponent id ID #REQUIRED>
<!ATTLIST ProcessComponent name CDATA #IMPLIED>
<!ELEMENT WorkProductPort EMPTY>
<!ATTLIST WorkProductPort id ID #REQUIRED>
<!ATTLIST WorkProductPort name CDATA #IMPLIED>
<!ATTLIST WorkProductPort portKind (in|out|inout) #REQUIRED>
<!ATTLIST WorkProductPort isOptional (true|false) #REQUIRED>
<!ATTLIST WorkProductPort WorkProductDefinitionRef IDREF
#IMPLIED>
<!ELEMENT Activity EMPTY>
<!ATTLIST Activity id ID #REQUIRED>
<!ATTLIST Activity name CDATA #IMPLIED>
<!ELEMENT WorkProductDefinition EMPTY>
<!ATTLIST WorkProductDefinition id ID #REQUIRED>
<!ATTLIST WorkProductDefinition name CDATA #IMPLIED>
<!ELEMENT WorkProductPortConnector (WorkProductPortRef)*>
<!ATTLIST WorkProductPortConnector id ID #REQUIRED>
<!ATTLIST WorkProductPortConnector name CDATA #IMPLIED>
<!ELEMENT WorkProductPortRef EMPTY>
<!ATTLIST WorkProductPortRef id ID #REQUIRED>
<!ATTLIST WorkProductPortRef ref IDREF #REQUIRED>
```

II- DTD ACME

```
<!ELEMENT System (Component|Connector|Attachment)*>
      <!ELEMENT Component (Port|Property)*>
            <!ATTLIST Component id ID #REQUIRED>
            <!ATTLIST Component name CDATA #IMPLIED>
            <!ELEMENT Port (Property)*>
                  <!ATTLIST Port id ID #REQUIRED>
                  <!ATTLIST Port name CDATA #IMPLIED>
                  <!ELEMENT Property EMPTY>
                        <!ATTLIST Property name CDATA #REQUIRED>
                        <!ATTLIST Property value CDATA #REQUIRED>
      <!ELEMENT Connector (Role)*>
            <!ATTLIST Connector id ID #REQUIRED>
            <!ATTLIST Connector name CDATA #IMPLIED>
            <!ELEMENT Role EMPTY>
                  <!ATTLIST Role id ID #REQUIRED>
                  <!ATTLIST Role name CDATA #IMPLIED>
      <!ELEMENT Attachment EMPTY>
            <!ATTLIST Attachment id ID #REQUIRED>
            <!ATTLIST Attachment name CDATA #IMPLIED>
                  <!ATTLIST Attachment portRef IDREF #REQUIRED>
            <!ATTLIST Attachment roleRef IDREF #REQUIRED>
```

III- DTD SPEMtoACME

```
<!ELEMENT xsl:stylesheet (xsl:template)>
<!ATTLIST xsl:stylesheet version CDATA #IMPLIED>
<!ATTLIST xsl:stylesheet xmlns:xsl CDATA #REQUIRED>
<!ELEMENT xsl:template (System)>
<!ATTLIST xsl:template match CDATA #REQUIRED>
<!ELEMENT System (xsl:for-each)*>
<!ELEMENT xsl:for-each (Component|Connector|Attachment*|xsl:for-
each|Port|Property|Role|xsl:if)>
<!ELEMENT xsl:if (Connector|Attachment)*>
<!ATTLIST xsl:if test CDATA #REQUIRED>
<!ATTLIST xsl:for-each select CDATA #REQUIRED>
<!ELEMENT Component (xsl:for-each*|Port)>
<!ATTLIST Component id CDATA #REQUIRED>
<!ATTLIST Component name CDATA #REQUIRED>
<!ELEMENT Connector (xsl:for-each|Role)*>
<!ATTLIST Connector id CDATA #REQUIRED>
<!ATTLIST Connector name CDATA #REQUIRED>
<!ELEMENT Role EMPTY>
<!ATTLIST Role id CDATA #REQUIRED>
<!ELEMENT Attachment EMPTY>
<!ATTLIST Attachment id CDATA #REQUIRED>
<!ATTLIST Attachment name CDATA #REQUIRED>
<!ATTLIST Attachment portRef CDATA #REQUIRED>
<!ATTLIST Attachment roleRef CDATA #REQUIRED>
<!ELEMENT Port (Property)*>
<!ATTLIST Port id CDATA #REQUIRED>
<!ATTLIST Port name CDATA #REQUIRED>
<!ELEMENT Property EMPTY>
<!ATTLIST Property name CDATA #REQUIRED>
<!ATTLIST Property value CDATA #REQUIRED>
```

IV- SPEMtoACME.XSLT

```
1    <?xml version="1.0" encoding="utf-8"?>
2    <xsl:stylesheet version="1.0"
3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4    <xsl:template match="/">
5    <System>
6        <xsl:for-each select="/Process/ProcessComponent">
7        <Component id="{@id}" name="{@name}">
8        <xsl:for-each select="Activity">
9          <Property name="{@name}" value=""/>
10         </xsl:for-each>
11         <xsl:for-each select="WorkProductPort">
12         <Port id="{@id}" name="{@name}">
13         <Property name="portKind" value="{@portKind}"/>
14         <Property name="isOptional" value="{@isOptional}"/>
15
16         </Port>
17       </xsl:for-each>
18       </Component>
19       </xsl:for-each>
20       <xsl:for-each select="/Process/WorkProductDefinition">
21         <Component id="{@id}" name="{@name}">
22         <Port id="{concat(@id,-1)}" name="{concat(@name,'port')}"/>
23         </Component>
24       </xsl:for-each>
25       <xsl:for-each select="/Process/WorkProductPortConnector">
26         <Connector id="{@id}" name="{@name}">
27         <xsl:for-each select="WorkProductPortRef">
28         <Role id="{@id}"/>
29         </xsl:for-each>
30       </Connector>
31       </xsl:for-each>
32       <xsl:for-each
33       select="/Process/ProcessComponent/WorkProductPort">
34         <xsl:if test="@WorkProductDefinitionRef">
35         <Connector id="{concat(@id,-2)}" name="{@name}">
36         <Role id="{concat(@id,-3)}"/>
37         <Role id="{concat(@id,-4)}"/>
```

```
38        </Connector>
39        </xsl:if>
40        </xsl:for-each>
41        <xsl:for-each select="/Process/WorkProductPortConnector">
42          <xsl:for-each select="WorkProductPortRef">
43          <Attachment id="{concat(@id,@ref)}"
44          name="{concat(@id,@ref)}"
45          portRef="{@ref}" roleRef="{@id}"/>
46        </xsl:for-each>
47        </xsl:for-each>
48        <xsl:for-each
49        select="/Process/ProcessComponent/WorkProductPort">
50        <xsl:if test="@WorkProductDefinitionRef">
51          <Attachment id="{concat(concat(@WorkProductDefinitionRef,-
52          1),concat(@id,-3))}"
53           name="{concat(concat(@WorkProductDefinitionRef,-
54          1),concat(@id,-3)
55          portRef="{concat(@WorkProductDefinitionRef,-1)}"
56          roleRef="{concat(@id,-3)}"/>
57          <Attachment id="{concat(@id,concat(@id,-4))}"
58          name="{concat(@id,concat(@id,-4))}" portRef="{@id}"
59          roleRef="{concat(@id,-4)}"/>
60        </xsl:if>
61        </xsl:for-each>
62    </System>
63    </xsl:template>
64    </xsl:stylesheet>
```

Annexe 2

Globe 1.0

Annexe 2

GLOBE 1.0

I- PRESENTATION DE GLOBE 1.0

1- DÉMARRAGE ET PAGE D'ACCUEIL

Lors du premier démarrage de ''GLOBE'' une fenêtre de dialogue s'affiche (*figure 1*). Demandant à l'utilisateur son choix du répertoire de travail, ce dernier contiendra toutes les éditions de l'utilisateur par la suite.

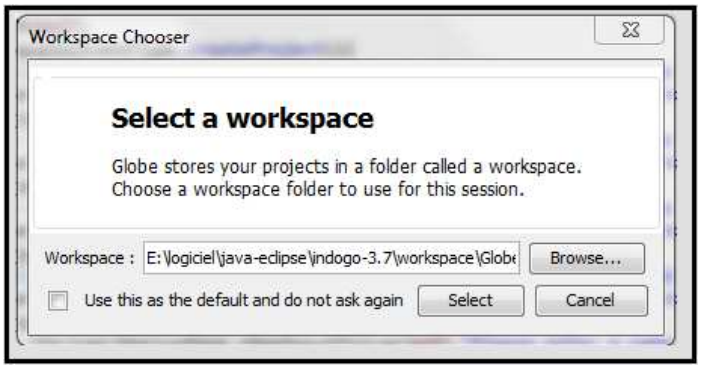


Figure 1 - Fenêtre du choix du poste de travail.

GLOBE fournit un répertoire par défaut, qui définit l'espace de travail. L'utilisateur peut changer ce répertoire en cliquant sur le bouton ''*Browse*''. Une fois le répertoire choisi l'utilisateur pourra alors valider son choix en appuyant sur le bouton ''*select*''. Il peut cocher l'option ''*use this as the default and do not ask again*'' s'il désir que ''GLOBE'' ne propose plus cette fenêtre lors d'un lancement ultérieur de l'application, ''*GLOBE*'' sauvegardera le répertoire choisi définitivement et sera instauré comme espace de travail par défaut.

Une fois l'espace de travail choisi, ''*GLOBE*'' affiche la fenêtre de bienvenue exposée dans la figure 2. Cette fenêtre propose une multitude d'options réparties sur six menus organisés selon les besoins de l'utilisateur, ces derniers sont expliqués dans le tableau 1.

Figure 2- Fenêtre d'accueil '' GLOBE''.

Menu	Description
First visit	Ce menu englobe les options d'accès à ''GLOBE ''.
Tutorials	Ce menu propose un tutorial vidéo (*ne peut être activé que si la JMF est installée*) ainsi qu'un document électronique pour ''*GLOBE* ''.
Future Project	Ce menu propose des documents d'appel au développement pour de prochaines versions de ''GLOBE '', il vise essentiellement des étudiants désirant continuer le projet pour les années à venir. les propositions sont classées selon l'estimation de leurs niveaux de difficulté.
You are conceptor	Les options de ce menu donne accès aux documents de conception de ''*GLOBE* '' tel le diagramme de cas d'utilisation ou de classe.
You are developper	Les options de ce menu donne accès aux documents de développent : code source ou la documentation sur ce dernier
About GLOBE	Ce menu informe sur les caractéristiques de l'application, les coordonnées du développeur…

Tableau 1- Description des menus de la fenêtre d'accueil.

2- FENÊTRE PRINCIPALE

Une fois que l'utilisateur à choisi l'une des options d'accès à l'outil de travail, la fenêtre principale s'affiche, la figure 3 représente cette dernière, les éléments numérotés sont expliqués dans le tableau 2.

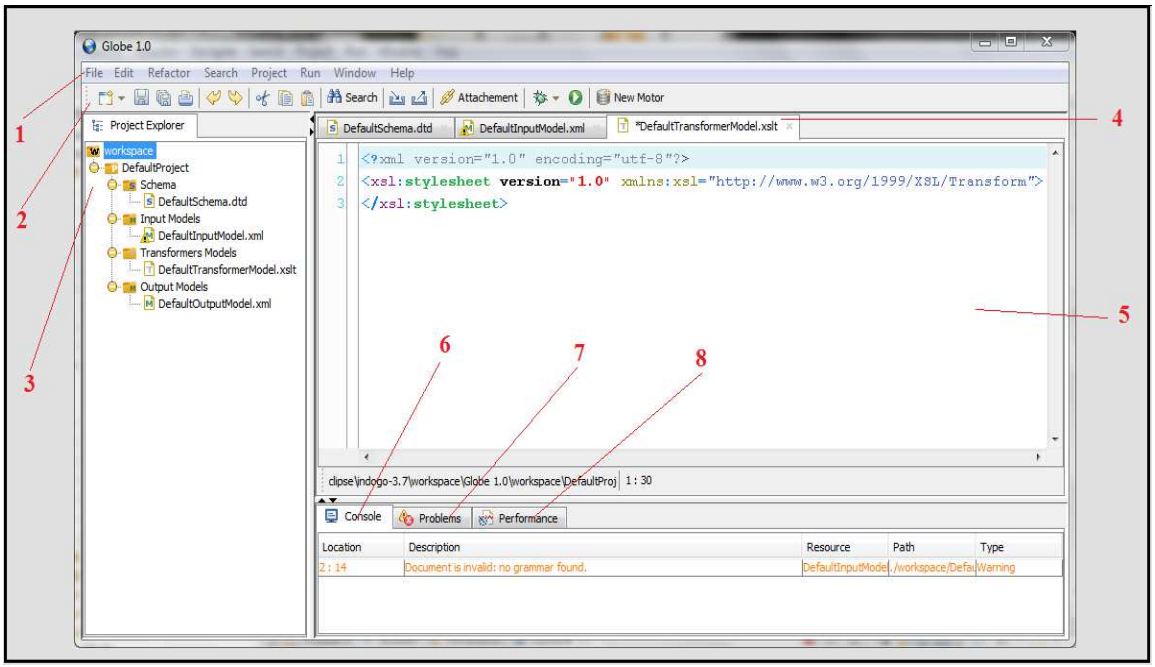


Figure 3 - Fenêtre principale.

N°	Nom du composant	Description du composant
1	Barre de menu	Comprend toutes les options offertes par « *Globe* » .
2	Barre d'outils	Contient quelques options utilitaires de la barre d'option.
3	Explorateur de projet	Offre une vision arborescente du système de données « *Globe* ».
4	Anglet d'exposition	Contient les feuilles d'édition.
5	Feuille d'édition	Fenêtre d'édition dont les données sont exposées.
6	Console	Affiche les erreurs de compilation éventuelles pour les données des divers langages supportés par « *Globe* » (DTD, XML, XSLT) d'un projet ou d'un fichier spécifique.
7	Fenêtre d'erreur	Affiche toutes les erreurs de compilation trouvées dans les fichiers du système de données pour les divers projets et fichier libre.
8	Fenêtre des performances	Affiche un graphe de performance à temps réel de l'application.

Tableau 2 - Description des principaux composants constituants la fenêtre principale

3- BARRE DE MENU

Elle est répartie sur huit menus comme (*figure 4*), elle supporte toutes les options instaurées dans '' *GLOBE*''.

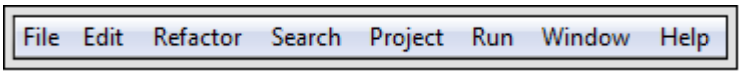


Figure 4- Barre de menu.

3.1 - MENU FILE

Ce menu (*figure 5*) contient toutes les options d'archivage des données éditées par un utilisateur, elle supporte la création et la sauvegarde des projets de transformation et de leurs modèles ainsi que quelques options utiles.

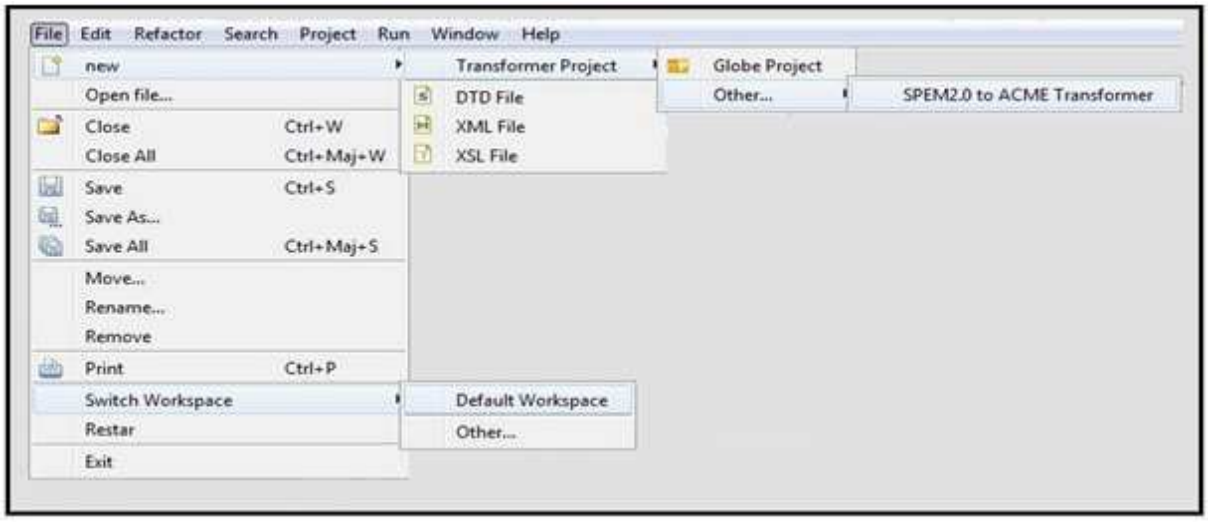


Figure 5- Menu File

3.2- MENU EDIT

Ce menu (*figure 6*) supporte des options liées à l'édition des modèles, il est en contacte directe avec la presse papier système, il supporte aussi des options de restauration de données.

Figure 6- Menu Edit

3.3- MENU REFACTOR

Ce menu (*figure 7*) Supporte des options d'ajout ou suppression de moteur de transformation spécialisé.

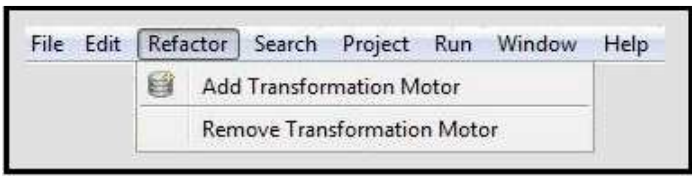


Figure 7- Menu Refactor

3.4- MENU SEARCH

Ce menu (*figure 8*) supporte des options de recherche d'entité à l'intérieur des modèles édités.

Figure 8- Menu Search

3.5- MENU PROJECT

Ce menu (*figure 9*) supporte les options de gestion de projet de transformation, une fenêtre de liaison de modèle est attachée à ce dernier.

Figure 9- Menu Project

3.6- MENU RUN

Ce menu (*figure 10*) supporte des options d'analyse et validation des entités éditées, ainsi que l'option de transformation déclenchant les moteurs de transformation.

Figure 10- Menu Run

3.7- MENU HELP

Ce menu (*figure 11*) supporte des options d'aide à l'utilisateur, il peut aussi restaurer la fenêtre de bienvenue exposée précédemment.

Figure 11- Menu Help

4- Barre d'outils

Cette barre (*figure 12*) supporte des options souvent utilisées par les utilisateurs, elle a comme objectif de faciliter le travail de ces derniers.

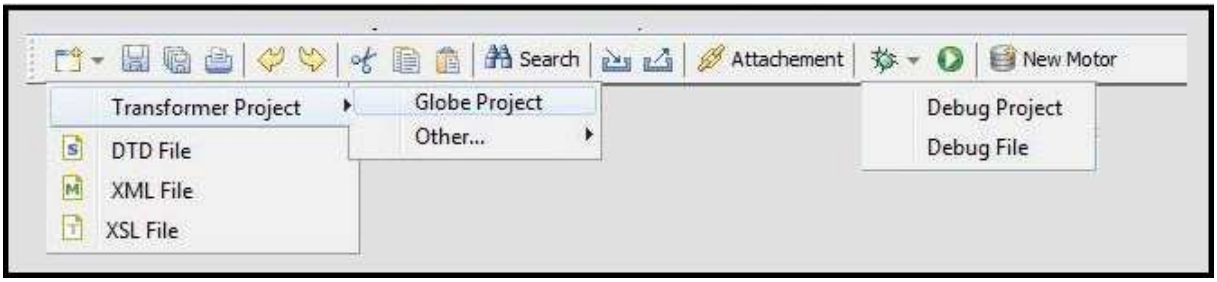


Figure 12- Barre d'options

5- EXPLORATEUR DE PROJET

Cet anglet (*figure 13*) représente graphiquement le système de fichier de '' *GLOBE* '' chaque projet est réparti sur trois (03) sous répertoire :

- ''*Schema* '' contenant les éditions des méta-modèles.
- '' *Input Models* '' et '' *Output Models* '' contenant consécutivement les modèles d'entrée et de sortie.
- '' *Transformers Models* '' contenant les modèles de transformation.

L'utilisateur a la liberté de créer et d'éditer des modèles en dehors d'un projet de transformation, ce dernier ne sera alors considéré que comme un simple fichier formaté et édité par ''*GLOBE* '', toutes les options décrites précédemment sont supportées pour ce dernier sauf l'option de transformation.

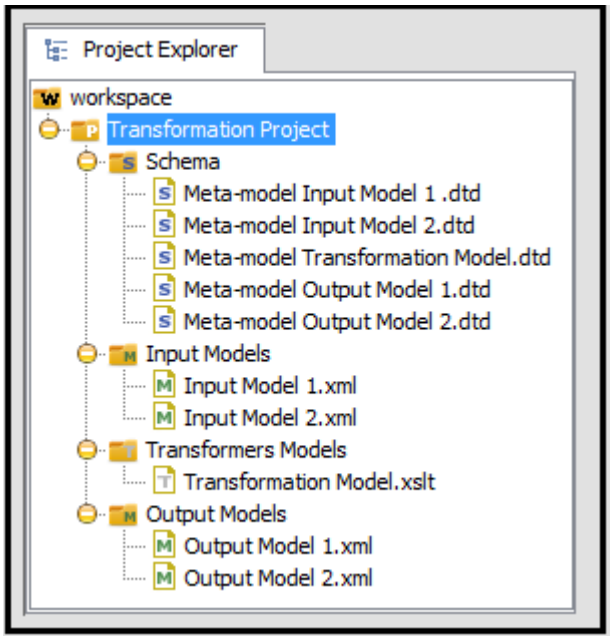


Figure 13- Explorateur de projets

6- ANGLET D'ÉDITION

Cet anglet (*figure 14*) contient les feuilles d'édition des différentes entités, l'utilisateur peut en tout instant fermer une feuille d'édition en cliquant sur le bouton (X) à coté du non du modèle édité, il peut restaurer ce dernier en cliquant trois fois avec la souris sur sa feuille dans l'explorateur de projet.

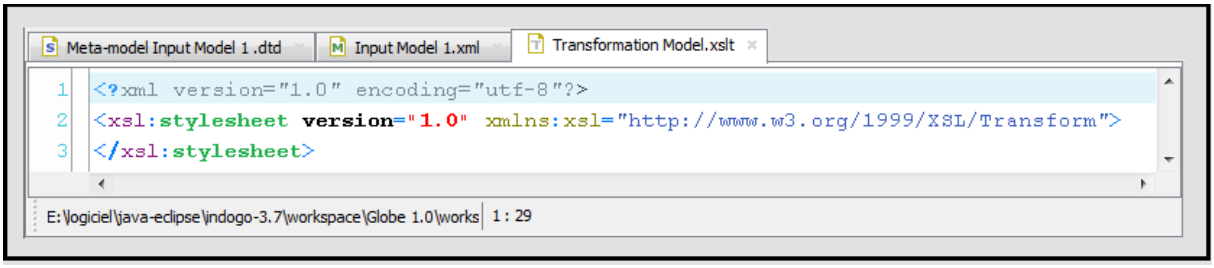


Figure 14- Explorateur de projets

7- FEUILLE D'ÉDITION

Il existe deux types de feuille d'édition : une feuille textuelle (*figure 15*) contient les données éditées ou restaurées par l'utilisateur, elle est équipée d'un détecteur de composant pour les langages à balise tel XML, et définit pour chaque composant une couleur propre pour le différencié des autre afin de faciliter l'édition lors de l'analyse du fichier.

Figure 15- Feuille d'édition Textuelle

Le deuxième type de feuille d'édition est graphique (*figure 16*) ce dernier n'est proposé que pour l'édition de modèle d'entée, il a comme objectif de faciliter l'édition à l'utilisateur.

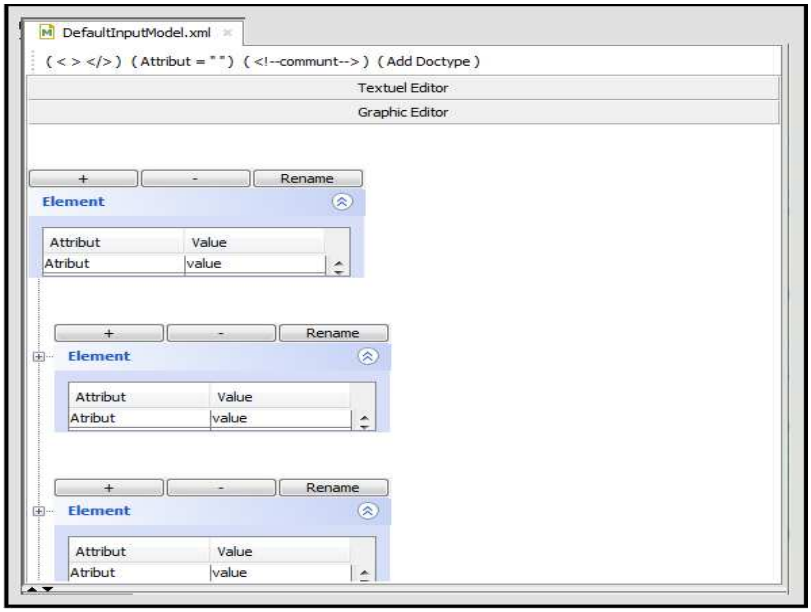


Figure 16- Feuille d'édition graphique

8- FENÊTRE D'ATTACHEMENT

Cette fenêtre (*figure 17*) a pour rôle la liaison des divers modèles d'un projet de transformation, elle contient deux anglets, le premier spécifiant les attachements entre schémas (méta-modèles) et modèle XML ou XSLT, le deuxième anglet spécifie les attachements entre modèle d'entée et son modèle de transformation.

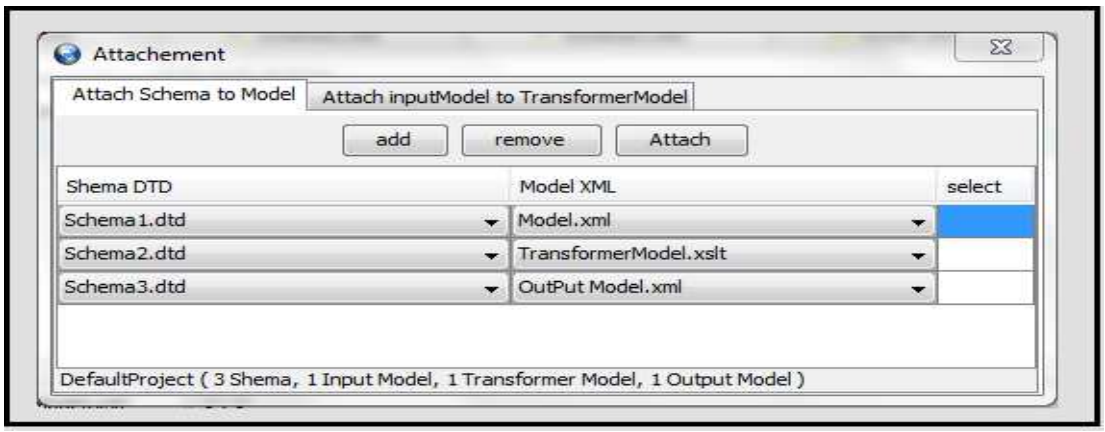


Figure 17- Fenêtre d'attachement de modèles

9- CONSOLE

Lors d'une analyse des fichiers édités il se peut que l'analyseur rencontre des non-conformités avec le type du fichier, cela génère une erreur lors de l'analyse, cette dernière sera transmise à la console (*figure 18*) pour que l'utilisateur puisse la voir. Elle indique non seulement la source qui vient de déclencher l'erreur, mais aussi la ligne et la colonne exacte de l'erreur dans le fichier, ainsi qu'un message portant sur sa nature.

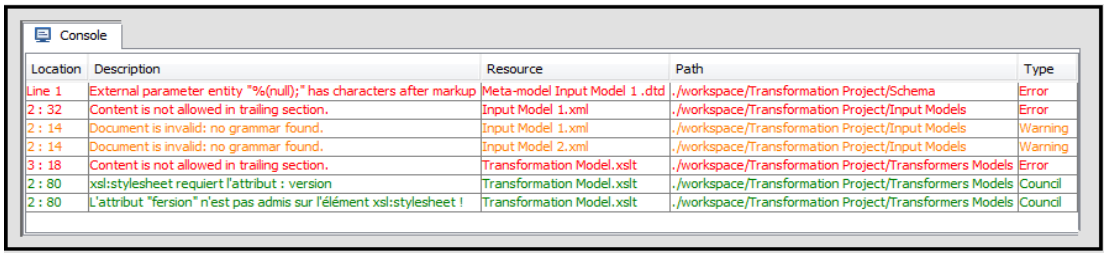


Figure 18- Console

10- PROBLÈMES

Cette fenêtre (*figure 19*) indique comme pour la console des erreurs d'analyse, sauf que cette fenêtre l'indique pour toutes les entités se trouvant sur le poste de travail.

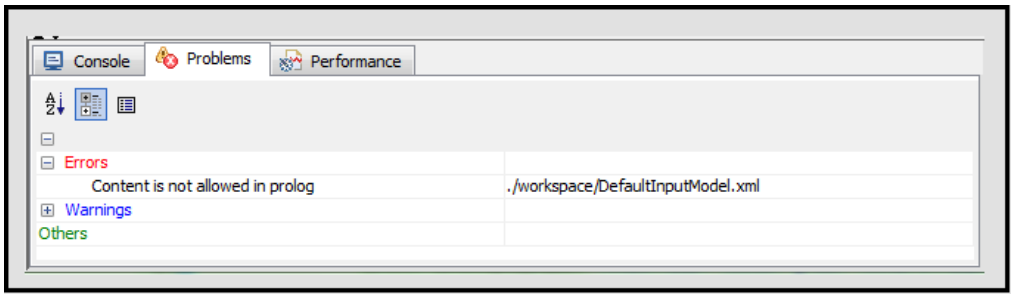


Figure 19- Fenêtre Problèmes.

11- FENÊTRE D'EXTENSION DE MOTEUR DE TRANSFORMATION

Garce à cette fenêtre (*figure 20*) l'utilisateur peut crée et mettre en marche un moteur de transformation spécialisé, il suffit d'indiquer les trois chemins des fichiers contenant les DTD pressentant les méta-modèles d'entée, de sortie, et de transformation, ainsi que le chemin du fichier contenant le modèle de transformation en format XSLT.

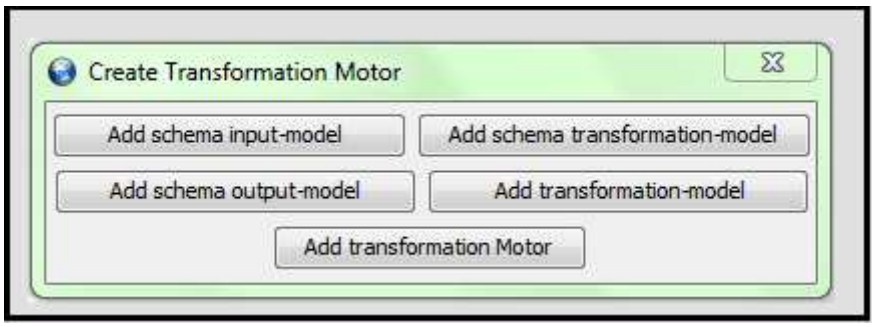


Figure 20- Fenêtre d'extension de moteur de transformation spécialisé

12- MENSURATION

Cette fenêtre (*figure 21*) indique le taux d'occupation du processeur centrale en temps réel par l'application.

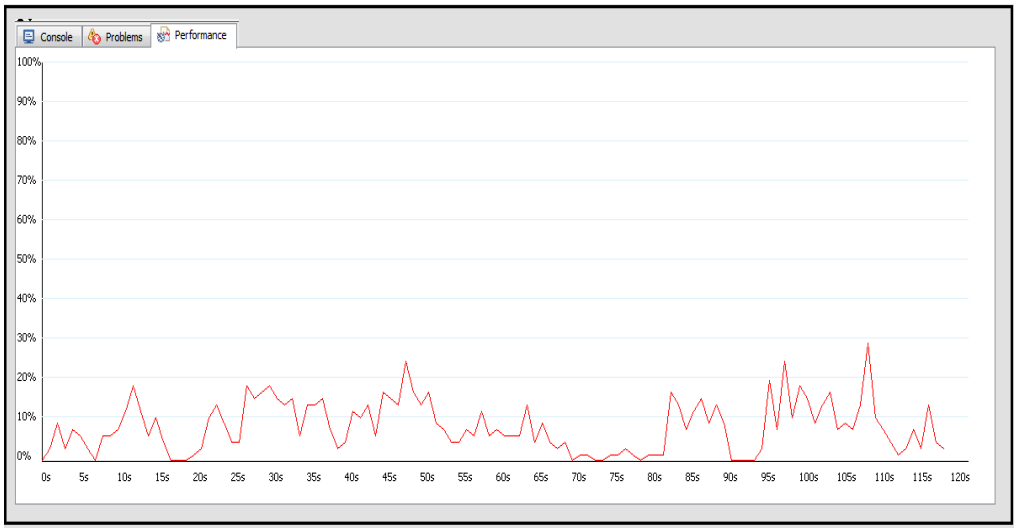


Figure 21- Fenêtre Mensuration.

II - TESTS D'INTÉGRITÉS

Nous vous exposons dans cette partie quelques tests d'intégrité qui ont été appliqués sur l'application à fin de vérifier son bon fonctionnement dans des conditions d'erreurs.

1- CONFORMITÉ AVEC LE SYSTÈME DE FICHIER DE LA PLATEFORME

Ce test a comme objectif de savoir si '' GLOBE '' arrive à détecter l'existence d'un fichier pour un chemin spécifique dans la plateforme d'exécution. L'application via un contrôle détecte si un nom de fichier du même non que le fichier de l'entité a été crée dans le même répertoire, si le fichier existe un choix est offert à l'utilisateur (*figure 22*) soit de supprimer le fichier existant ou d'annuler l'opération

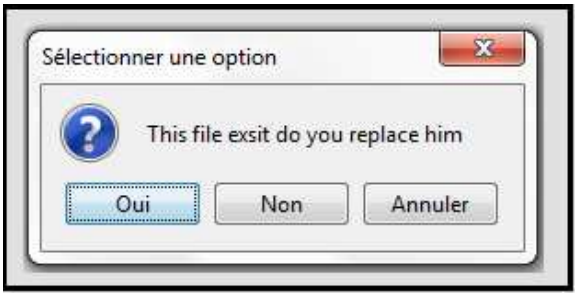


Figure 22- Fenêtre de dialogue pour l'écrasement de fichier existant

2. COMPILATION ET EXÉCUTION

Ce test a pour objectif de savoir si '' GLOBE '' arrive à détecter les erreurs d'édition commises par l'utilisateur, le centre de validité a la charge de détecter ce genre d'erreur et d'informer l'utilisateur. L'information sur l'erreur est affichée dans la console et la ligne

d'erreur dans la feuille d'édition est remplit en couleur rouge pour faciliter la localisation (*figure 23*), cette analyse d'erreur est utilisée pour la validation ou l'exécution d'une transformation.

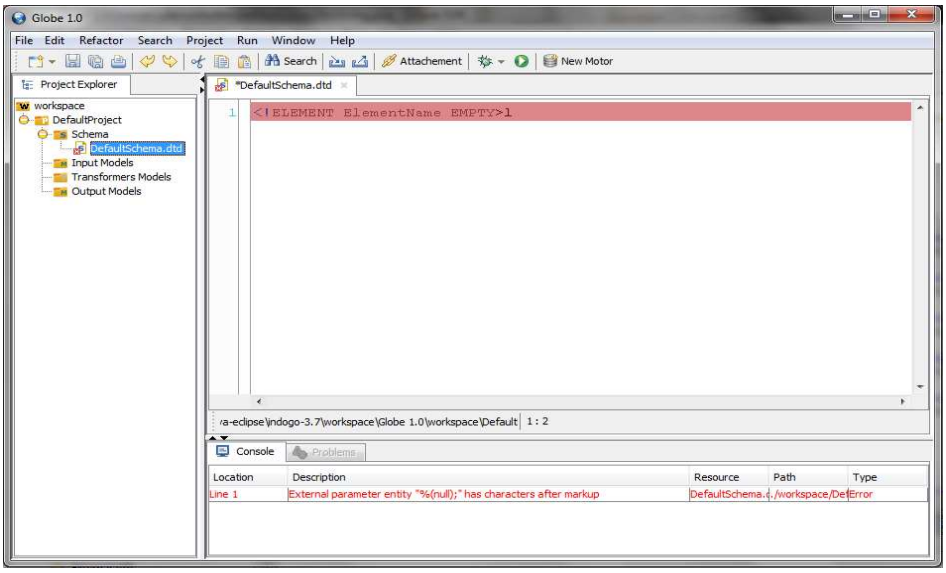


Figure 23- Détection d'erreur d'édition dans '' GLOBE 1.0

III. MISE EN PRODUCTION

1- PRODUIT '' GLOBE1.0 ''

1.1- INSTALLATION DE '' GLOBE 1.0 ''

Le répertoire '' *GLOBE* '' contenant l'application (*figure 24*) est réparti sur six répertoires et un fichier d'exécution '' *GLOBE1.0.jar* ''.

Comme première étape, il est conseillé que l'utilisateur installe l'application JMF contenu dans le répertoire JMF, cette dernière est proposée pour plusieurs plateforme dont Windows et l'Unix, une fois la JMF installé l'utilisateur peut lancer le ficher d'exécution '' *GLOBE1.0.jar* '' pour accéder à l'application.

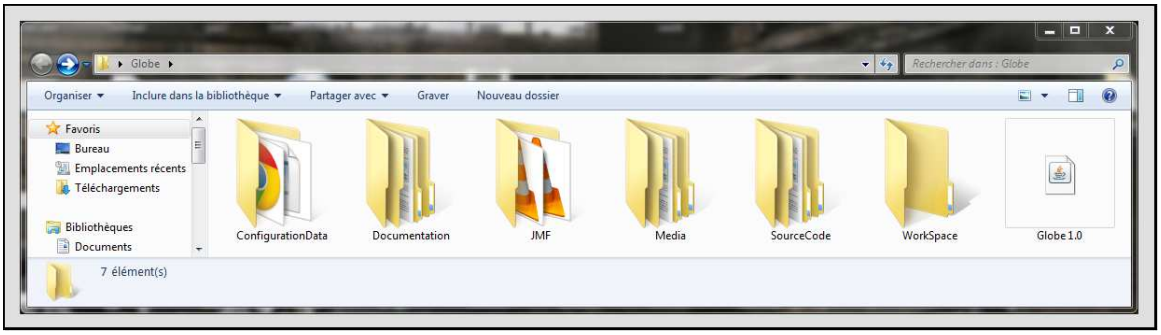


Figure 24- Contenu du répertoire GLOBE.

1.2- A PROPOS DE GLOBE

Cette fenêtre (*figure 25*) fournit des informations utiles sur le constructeur. Elle est accessible à partir du menu '' Help '' via l'option '' About GLOBE ''.

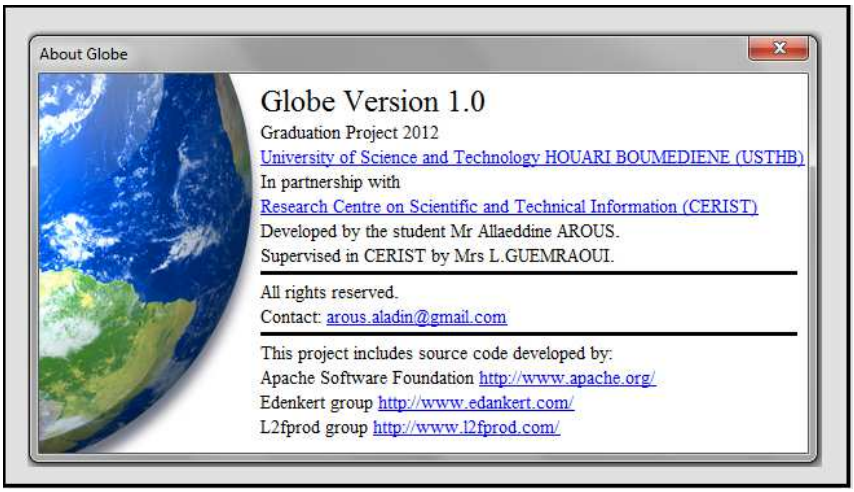


Figure 25- A propos de '' Gobe 1.0 ''

2- DOCUMENTATION

Il est mis à la disposions de l'utilisateur plusieurs documents et tutoriaux qui vont l'aider à mieux comprendre le fonctionnement de '' *GLOBE* '' ainsi qu'une documentation sur le développement de l'application.

2.1- DOCUMENTATION SUR LE PROGRAMME

Cette documentation (*figure 26*) fait partie intégrante de ''GLOBE'' l'utilisateur peut accéder à cette fenêtre en cliquant sur l'option '' JavaDoc '' dans le menu '' Help ''.

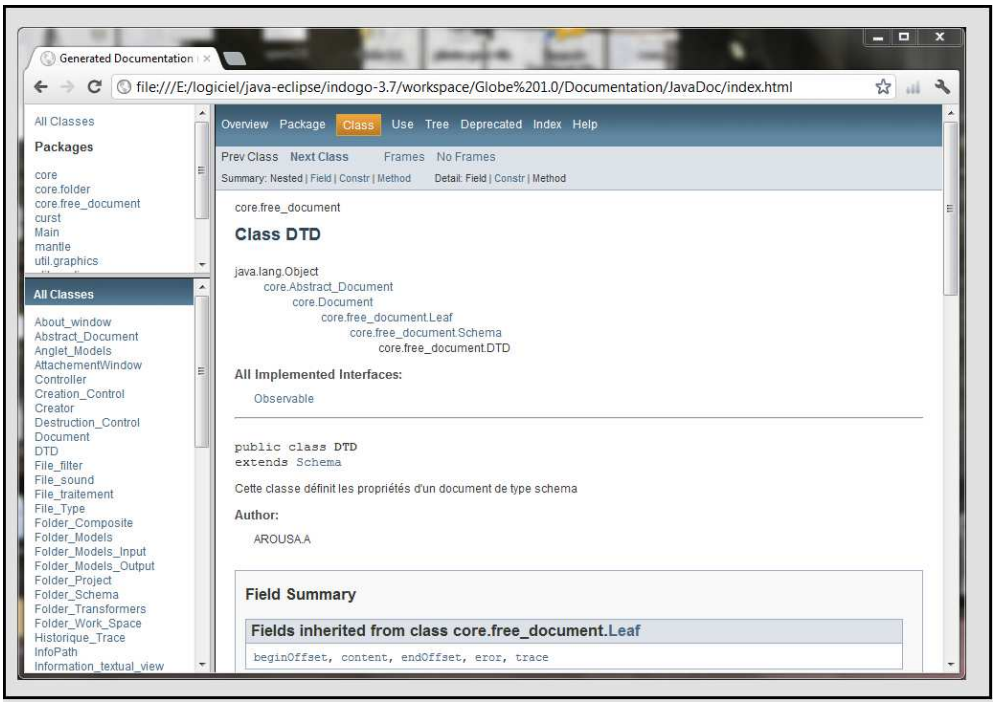


Figure 26- JavaDoc du code source.

2.2- DOCUMENTATION SUR L'APPLICATION

L'utilisateur pourra accéder à ces documents par l'intermédiaire de la fenêtre de bienvenus.

Résumé

Ce mémoire relate une approche pour la réutilisation et l'interopérabilité des procédés logiciels hétérogènes. Elle consiste à voir un procédé logiciel comme étant un ensemble de sous procédés manipulables via leurs interfaces externe. Ces sous procédés sont modélisés de manière abstraite, ils définissent de manière explicite les interactions entre les différents fragments d'un processus et fournissent ainsi, un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments.

Pour cela, on utilise les principes d'architecture logicielle en tant que « perspective haut niveau d'un système logiciel », reconnue comme le cœur d'une discipline à part entière et dont la principale caractéristique réside à être un modèle abstrait significatif de la structure et du comportement d'un système logiciel.

Toutefois, afin de rapprocher ces deux axes de recherche, à savoir l'ingénierie des procédés et les architectures logicielles, nous avons eu recours aux méthodes de transformation de modèles qui fournissent une migration technologique cohérente.

Ainsi, la méthode donne la possibilité d'améliorer et de concevoir de nouveaux modèles de procédés logiciels par simple intégration des composants de procédés réutilisables.

Mots clés :

Procédé logiciels, Composants de procédé, Architecture logicielle, Ingénierie dirigée par les modèles, Réutilisation, Interopérabilité.