HAL
open science

Réutilisation des procédés logiciels : Une approche à base d'architectures logicielles

Fadila Aoussat

To cite this version:

Fadila Aoussat. Réutilisation des procédés logiciels : Une approche à base d'architectures logicielles . Génie logiciel [cs.SE]. Université de Nantes, 2012. Français. NNT : . tel-01146332

HAL Id: tel-01146332

https://hal.science/tel-01146332

Submitted on 29 Apr 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Centrale de Nantes **Université de Nantes** **École des Mines de Nantes**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2012

N° attribué par la bibliothèque

Réutilisation des procédés logiciels :

Une approche à base d'architectures logicielles

THÈSE DE DOCTORAT

Discipline : INFORMATIQUE

Spécialité : INFORMATIQUE

Présentée
et soutenue publiquement par

Fadila AOUSSAT

*Le 2 octobre 2012 à l'*UFR *Sciences & Techniques, Université de Nantes,*
devant le jury ci-dessous

Président	:	Pr.	
Rapporteurs	:	Henri BASSON, Professeur	Université de Lille Nord de France
		Kamel BARKAOUI, Professeur	CNAM, France
Examinateurs	:	Cristophe CHOQUET, Professeur	Université du Maine, France
		Christian ATTIOGBÉ, Professeur	Université de Nantes, France
		Zaia ALIMAZIGHI, Professeur	USTHB, Alger, Algérie

Directeur de thèse : Pr. Mourad OUSSALAH

Co-Directeur de thèse : Pr. Mohamed Ahmed Nacer

Laboratoire: LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 0366-XXX

RÉUTILISATION DES PROCÉDÉS LOGICIELS :

UNE APPROCHE À BASE D'ARCHITECTURES LOGICIELLES

Software processes reusing :

An approach based on software architectures

Fadila AOUSSAT

favet neptunus eunti

Université de Nantes

Fadila AOUSSAT

Réutilisation des procédés logiciels :

Une approche à base d'architectures logicielles

IV+x+170 p.

Ce document a été préparé avec LaTeX2e et la classe these-LINA version v. 2.7 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe these-LINA est disponible à l'adresse : http://login.irin.sciences.univ-nantes.fr/.
Cette classe est conforme aux recommandations du ministère de l'éducation nationale, de l'enseignement supérieur et de la recherche (circulaire n° 05-094 du 29 mars 2005), de l'Université de Nantes, de l'école doctorale « Sciences et Technologies de l'Information et des Matériaux » (ED-STIM), et respecte les normes de l'association française de normalisation (AFNOR) suivantes :

- AFNOR NF Z41-006 (octobre 1983)
  Présentation des thèses et documents assimilés ;
- AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure ;
- AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Impression : These_AoSP-1-(Dernier).tex – 02/10/2012 – 8:56.

Révision pour la classe : these-LINA.cls,v 2.7 2006-09-12 17:18:53 mancheron Exp

— Fadila AOUSSAT,
Réutilisation des procédés logiciels : Une approche à base d'architectures logicielles .

Résumé

Boehm [25] met en évidence la dualité produit logiciel/procédé logiciel concernant les architectures logicielles. En se basant sur l'article *"Software Processes Are Software Too"* d'Osterweil [96], il confirme que si les architectures logicielles sont efficaces pour la réutilisation des produits logiciels, elles seront d'une réelle contribution pour la réutilisation des procédés logiciels. *"If open architectures are good for software product reuse, then their process counterparts will be good for software process reuse"*. Nos travaux se référent donc à la réutilisation des Procédés Logiciels (PLs) en se basant sur le paradigme d'architecture Logicielle (AL).
Cette thèse constitue une première contribution à la réutilisation des procédés logiciels à base d'architectures logicielles, notre contribution se décline en deux points essentiels :
- La définition d'un cadre de comparaison où nous identifions les caractéristiques essentielles des procédés logiciels et les spécificités des approches de réutilisation de procédés logiciels à base d'architectures logicielles. Ce cadre de comparaison permettra de cerner, classifier et évaluer les approches de réutilisation de PLs à base d'architectures logicielles.
- L'élaboration d'une approche de réutilisation de procédés logiciels baptisée AoSP (Architecture oriented Software Process). Cette approche a pour objectif d'exploiter la réutilisation "pour" et "par" des PLs tout en les combinant aux autres opportunités de réutilisation offertes par l'exploitation d'une ontologie de domaine.

Mots-clés : Procédés logicieL (PL), réutilisation, architectures logicielles, ontologie de domaine, SPEM (System and Software Engineering Metamodel), extension de profil UML, éléments architecturaux, connecteurs PL explicites, styles de PLs, transformation de modèles, capitalisation de connaissances PLs, inférence de connaissances PLs, recherche d'architectures de PLs, déploiement d'architecture de PLs.

Abstract

Boehm [25] highlights the duality between software product / software process concerning software architectures. Based on the paper *"Software Processes Are Software Too"* of Osterweil [96], he confirms that *"If open software architectures are good for product reuse, then processes Their Counterparts Will Be good for software process reuse"*. Our work therefore refer to the reuse of software processes based on the software architecture paradigm.
This thesis is a first contribution on the reuse of software processes based on software architectures ; our contribution is constituted in two main points :

- The definition of framework for comparison in which we identify the essential characteristics of software processes and the specificities of the approaches for reusing software processes based on software architectures. This framework will be used to classify and evaluate the proposed approaches for reusing software processes based on software architectures.

- The elaboration of an approach to software process reusing called AOSP (Software Process Oriented Architecture). This approach attempts to exploit "for" and "by" reusing software processes combined with other reuse opportunities offered by exploiting a domain ontology.

Keywords: Software Process (SP), reuse, Software architectures, domain ontology, SPEM (System and Software Engineering Metamodel), UML profil extention, architectural elements, Explicit SP connectors, SP styles, models transformation, SP knowledge capitalizing, SP knowledge infering, software architectures retreiving, SP architecture deployement.

Remerciements

...

Sommaire

— *Corps du document* —

Introduction générale 1
1 Les procédés logiciels 5
2 Les architectures logicielles au service de la réutilisation des procédés logiciels 21
3 Évaluation des approches de réutilisation de procédés logiciels à base d'architectures logicielles 35
4 L'approche AoSP (Architecture oriented Software Process) 61
5 Implémentations et expérimentations 89
Conclusion générale 129

— *Annexes* —

A Le métamodèle SPEM 135
B Les ontologies de domaine : Notions et concepts 137

— *Pages annexées* —

Bibliographie 147
Liste des tableaux 163
Liste des figures 165
Table des matières 167

Introduction Générale

Cadre de travail

Dans le domaine du génie logiciel, il est depuis longtemps reconnu qu'une application est un produit manufacturé complexe dont la réalisation doit s'intégrer dans une démarche méthodologique. La démarche méthodologique est explicitée à travers l'utilisation de modèles de procédés logiciels.

Un modèle de procédé logiciel est la description de l'enchainement des activités, des ressources, des outils utilisés, ainsi que la description des intervenants pour la réalisation puis la maintenance d'un produit logiciel. Ce dernier a pour but la maîtrise de la complexité croissante des projets de développement de logiciels.

Le développement logiciel est un effort complexe, collectif, créatif et évolutif, et par conséquent, le procédé logiciel doit fournir le support adéquat pour prendre en charge cette réalité du développement.

La qualité du modèle de procédé logiciel et la qualité de son exécution ont un impact direct sur la qualité du produit logiciel manufacturé. Malheureusement, les modèles de procédés logiciels existants ne sont pas toujours adéquats, trop complexes à modéliser ou à exécuter [113] peinent à combiner des caractéristiques indispensables aux procédés logiciels telles que la flexibilité et le contrôle d'exécution [118] [21], ou la simplicité et l'efficacité, et souvent, échouent à s'imposer comme solution incontournable dans le domaine industriel.

Le domaine académique a donné naissance à un nombre considérable de procédés logiciels, malheureusement, ces propositions sont restées au niveau des laboratoires de recherche et n'ont pas pu s'imposer dans le monde industriel. Ils leur sont souvent reprochés d'utiliser des notations formelles difficiles à mettre en œuvre et de fournir des outils peu exploitables. De plus, la réalité du développement, la concurrence et les pressions du marché du logiciel ont fait que les entreprises de développement de logiciels préfèrent maîtriser leurs propres procédés de développement qui sont déjà opérationnels, favorisant les opérations d'améliorations et d'adaptations à leur besoins, que d'investir dans de nouveaux procédés émanant du monde académique dont l'utilisation n'est pas évidente.

Par ailleurs, la modélisation des procédés logiciels n'échappe pas aux contraintes de développement auxquelles sont soumis les logiciels ; modéliser des procédés logiciels de qualité dans des délais et à des prix compétitifs reste une priorité.

Notre thèse s'inscrit dans le cadre de l'ingénierie des procédés logiciels, nous nous intéressons principalement à la modélisation et à l'exécution des procédés logiciels. Notre objectif est, d'une part, de contribuer à l'amélioration de procédés logiciels du monde industriel tout en se basant sur les approches académiques, d'autre part, de contribuer à l'amélioration de la qualité des modèles de procédés logiciels tout en minimisant les efforts et le temps de leur mise en place.

Problématique

La progression rapide des technologies et des outils de développement, le changement continu des méthodes et des pratiques de développement (développement orienté composant, programmation orientée aspect, programmation en binôme,...etc.), suggèrent souvent de nouvelles méthodes et processus de

développement (UP, XP, Scrum...etc.). Pour que les procédés logiciels restent performants, ils doivent intégrer continuellement ces nouvelles pratiques et s'adapter à ces nouvelles technologies.

Incontestablement, les procédés logiciels doivent évoluer et suivre les progressions technologiques, néanmoins, ces derniers ne peuvent se passer des acquis en termes d'expériences et de meilleures pratiques capitalisées lors de leurs précédentes modélisations et exécutions. En effet, les procédés logiciels sont centrés humain [110] et malgré la répétitivité des activités de développement, l'expérience et le savoir-faire humain restent primordiaux lors de la modélisation et de l'exécution des procédés logiciels.

La richesse du domaine en termes d'expériences et de savoir-faire [26] suggère l'exploitation d'une approche de réutilisation pour la modélisation des procédés logiciels.

La répétitivité des tâches, l'importance des interactions ainsi que l'exploitation des structures récurrentes font partie des caractéristiques intrinsèques du procédé logiciel. Ces caractéristiques nous évoquent les caractéristiques des architectures logicielles et nous orientent naturellement vers l'exploration des architectures logicielles pour **la réutilisation des procédés logiciels.**

Nous pensons que les architectures logicielles peuvent contribuer à la modélisation de procédés logiciels de qualité. En effet, le domaine des architectures logicielles est arrivé à une degré de maturité considérable et il est judicieux de tirer avantages des opportunités offertes par ce domaine pour promouvoir la réutilisation des procédés logiciels. Aussi, nous pensons que l'exploitation des concepts architecturaux tels que les connecteurs, les styles architecturaux et l'utilisation de langages tels que les ADLs (Architecture Description Languages), peuvent être des atouts majeurs à la modélisation des procédés logiciels par approche de réutilisation.

La capitalisation des expériences précédentes et des meilleures pratiques doit être prise en compte lors de la réutilisation des procédés logiciels ; l'utilisation d'une ontologie de domaine parait une solution judicieuse. En effet, cette dernière nous permettra le partage des connaissances capitalisées, elle offrira non seulement la possibilité de gérer l'hétérogénéité de la terminologie utilisée, mais aussi, et surtout, elle permettra l'émergence de nouveaux procédés logiciels par mécanisme d'inférence.

A travers cette thèse nous contribuons à la réutilisation des procédés logiciels à base d'architectures logicielles. En prenant en compte l'ampleur des travaux que nous devons réalisés, nous nous focalisons principalement sur l'étude de l'aspect structurel du modèle de procédé logiciel ; l'aspect dynamique étant régi par d'autres mécanismes et ayant d'autres exigences, fait partie de nos futures travaux.

Contributions

La première contribution que nous proposons est un cadre de comparaison qui permettra d'étudier et d'analyser les approches de réutilisation de procédés logiciels à base d'architectures logicielles. Ce cadre de comparaison met en exergue les caractéristiques de procédés logiciels [48] [2] [123] avec ceux des architectures logicielles [78] [16] [1] dégagées dans les différents travaux effectués précédemment dans ces deux domaines. Aussi, nous avons mené une étude bibliographique concernant les approches de réutilisation de procédés logiciels à base d'architectures logicielles, le cadre de comparaison nous a permis d'analyser ces approches et d'identifier leurs insuffisances.

La deuxième contribution consiste à proposer une nouvelle approche de réutilisation de procédés logiciels à base d'architectures logicielles : AoSP (Architectures oriented Software Processes). Cette approche se base sur les insuffisances des approches existantes, elle tente de tirer avantages des solutions offertes par les architectures logicielles tout en respectant les spécificités des procédés logiciels (centré humain, différents types d'exécutions...etc.). Aussi, et pour prendre en compte l'expérience et le savoir-faire des développeurs, cette approche exploite une ontologie de domaine permettant de capitaliser

les connaissances procédés logiciels et incluant des mécanismes d'inférence pour déduire de nouvelles connaissances procédés logiciels.

L'objectif de AoSP est de [11] :

- Proposer une solution générale qui couvre un large éventail de modèles de procédés logiciels.
- Améliorer la qualité des modèles de procédés logiciels et offrir la qualité exigée pour supporter la réalité du développement.
- Favoriser la réutilisation des procédés logiciels en exploitant les modèles de procédés logiciels déjà modélisés.
- Augmenter la réutilisabilité des procédés logiciels en modélisant des modèles de procédés logiciels réutilisables.

Organisation du document

Notre document est organisé comme suit :

- **Chapitre 1 :** Ce chapitre introduit les notions de base du domaine des procédés logiciels, il détaille les concepts de base des procédés logiciels en se focalisant sur leur modélisation et méta-modélisation. Ce chapitre permet d'identifier les caractéristiques intrinsèques des modèles de procédés logiciels.
- **Chapitre 2 :** Ce chapitre est dédié à la présentation des architectures logicielles. En plus de la présentation des concepts de base du domaine des architectures logicielles (éléments architecturaux, langages spécifiques au domaine...etc.), nous identifions les avantages que peuvent apporter les architectures logicielles à la réutilisation des modèles de procédés logiciels.
- **Chapitre 3 :** Ce chapitre aborde l'étude du domaine de la réutilisation des procédés logiciels à base d'architectures logicielles et à base d'ontologies de domaine.

  Ce chapitre présente notre proposition de cadre de comparaison qui permet d'étudier et d'analyser des approches de réutilisation de procédés logiciels à base d'architectures logicielles. L'étude et l'analyse des approches existantes sont aussi présentées dans ce chapitre.

  D'autre part, l'étude des approches de réutilisation à base d'ontologies de domaine est aussi présentée, l'objectif de cette étude est d'identifier les ontologies de domaine qui peuvent être exploitée par notre approche.

  L'étude des approches de réutilisation de PLs à base d'architectures logicielles et à base d'ontologies de domaine nous a permis d'identifier leurs insuffisances sur lesquelles nous nous somme basés pour mettre en place notre propre approche de réutilisation de procédés logiciels à base d'architectures logicielles.
- **Chapitre 4 :** Ce chapitre décrit l'approche AoSP que nous proposons comme solution pour pallier les insuffisances des approches étudiées. Cette approche couvre les deux ingénieries de réutilisation "pour" et "par" la réutilisation de procédés logiciels, points que nous détaillons dans les sections de ce chapitre.
- **Chapitre 5 :** Ce chapitre est consacré à la présentation de la partie implémentation et expérimentations effectuées pour valider notre approche. Ce chapitre détaille les solutions techniques adoptées et les algorithmes utilisés pour la mise en service de l'approche AoSP. Des exemples et des cas d'études sont présentés après présentation de chaque solution technique adoptée.
- Nous terminons ce document par une conclusion qui résume les travaux réalisés et énonce les perspectives de recherche.

  Deux annexes sont également fournies en fin de document.

Annexe 1 : Cette annexe propose une étude complémentaire du métamodèle SPEM (System and Software Engineering Metamodel).
Annexe 2 : Cette annexe est dédiée à la présentation des concepts de base des ontologies de domaine.

CHAPITRE 1

Les procédés logiciels

1.1 Introduction

Le sujet traité dans ce chapitre relève de l'ingénierie des procédés logiciels (PLs). Nous nous intéressons principalement à la réutilisation des PLs. Par conséquent, la mise au point de l'état du domaine des PLs s'avère indispensable.

L'objectif de ce premier chapitre est de clarifier la notion de PL ; en effet, l'utilisation de différentes terminologies telles que cycles de vie de logiciels, processus logiciels et procédés logiciels peut introduire des ambigüités et prêter à confusion. La modélisation et la méta-modélisation des PLs est le sujet principal de ce chapitre. Comme exemple de métamodèle de PL nous introduisons le métamodèle SPEM (System and Software Process Engineering Metamodel) [88] qui est exploité par notre approche de réutilisation de PLs. Les caractéristiques spécifiques aux PLs que nous avons jugées essentielles sont aussi détaillées. Par ailleurs, étant donné que nous dédions le chapitre -3- aux approches de réutilisation de PLs, dans ce chapitre nous introduisons seulement les principes et concepts de base de la réutilisation.

1.2 L'origine des Procédés Logiciels

L'origine des modèles de PLs remonte aux premiers projets logiciels des années 60 et 70 [105]. Des notions de cycles de vie de logiciels (qui représentent les ancêtres du PL) ont été définies et servaient de représentation de base pour planifier, coordonner et contrôler le développement logiciel. Ces cycles de vies sont des schémas conceptuels -représentés souvent par des diagrammes- qui permettent de décrire les activités de développement et leurs enchainements (figure - 1.1-). Les activités décrivaient souvent les phases principales du développement et ne reflétaient pas la complexité des tâches à réaliser [105]. Ces schémas conceptuels n'étaient pas de granularité assez fine pour expliciter les caractéristiques et les spécificités des tâches à réaliser.

C'est dans ce contexte que les modèles de PLs ont fait leur apparition. Les modèles de PLs sont plus détaillés, plus riches syntaxiquement et sémantiquement ; ils couvrent de manière plus précise la réalité du développement logiciel [105].

A partir des années 80, l'effervescence du domaine de l'ingénierie des PLs a donné naissance à un nombre considérable de travaux sur les PLs [48]. Malgré le large éventail de langages, d'environnements et d'approches proposés pour la modélisation et l'exécution des PLs, les travaux sur les PLs sont toujours d'actualité, plusieurs nouvelles orientations telles que : la simulation des PLs, le développement agile [65], les méthodes situationnelles [36]...etc. ont vu le jour et sont actuellement explorées pour améliorer la qualité du PL.

1.2.1 Le modèle de cycle de vie de logiciel

Il est difficile de faire la différence entre un cycle de vie de logiciel et un procédé logiciel ; si nous résumons la définition du modèle de cycle de vie de logiciel, nous dirons qu'un modèle de cycle de vie de logiciel est une représentation souvent graphique (diagramme, schéma...etc.) qui décrit la façon dont le logiciel est ou doit être développé. Il permet de décrire les séquences et les étapes à suivre afin de réaliser un logiciel [105], il se focalise généralement sur la vision temporelle du cycle de développement (figure-1.1-).

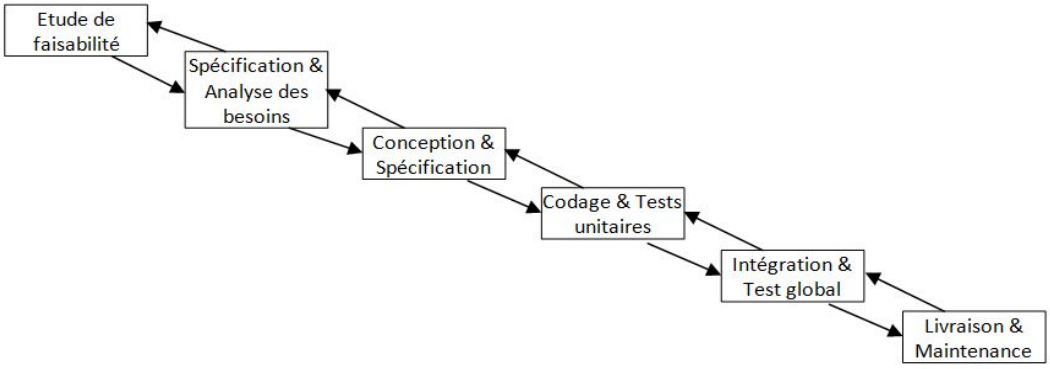


Figure 1.1 Cycle de vie du logiciel Waterfall (Royce 1970) [105].

1.2.2 Le modèle de procédé logiciel

Le modèle de procédé logiciel est aussi une représentation des séquences et des étapes à suivre pour réaliser un logiciel, cependant, la représentation est de fine granularité, en d'autres termes, le modèle de procédé logiciel permet de décrire de manière plus détaillée les tâches, les outils et les ressources utilisés lors du développement logiciel. Les séquences de tâches sont des enchainements d'actions qui ne sont pas forcement linéaires ; l'itération, le parallélisme et l'ordonnancement partiel des actions sont admis. La force des modèles de procédés logiciels réside dans leur richesse sémantique et syntaxique qui leur confère la possibilité d'être exécutable, contrairement aux modèles de cycle de vie de logiciels qui sont considérés comme des modèles descriptifs seulement.

1.2.3 Le modèle de processus logiciel

Le terme "Processus Logiciel" est également utilisé pour désigner les procédés logiciels. En réalité, dans le domaine des PLs il n'y a pas de différence avérée entre les termes "Processus" et "Procédé", et leur traduction en anglais est la même : "Process".

1.3 Modélisation et Méta Modélisation des procédés logiciels

Afin d'organiser et de structurer les modèles, l'OMG (Object Managment Group) [92] a défini une architecture de modélisation à plusieurs niveaux appelée : Architecture multi niveaux des modèles (figure-1.2-). Cette architecture fournit un support d'abstraction basé sur les modèles qui permet de maîtriser la complexité croissante des logiciels.

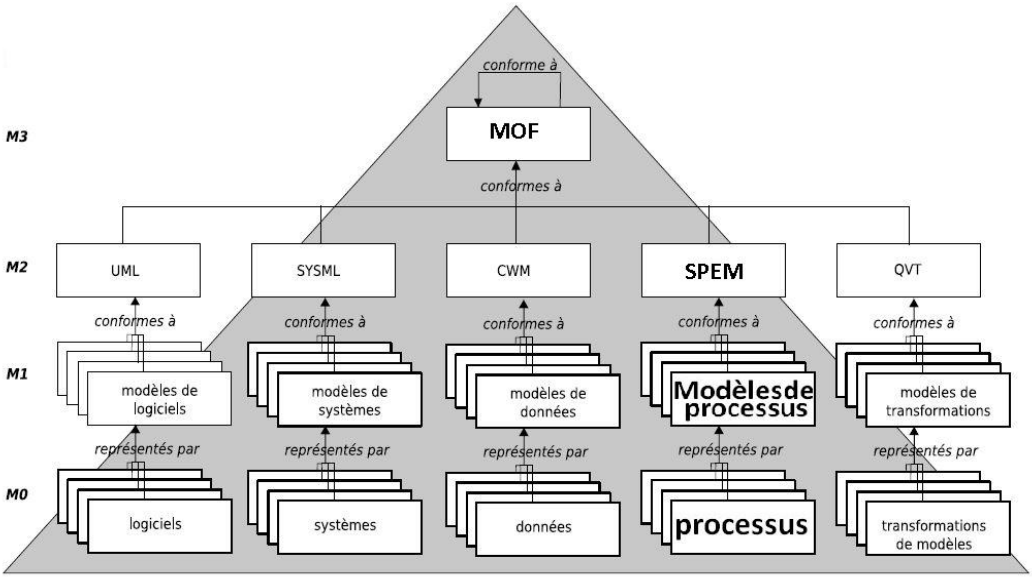


Figure 1.2 L'architecture à 4 niveaux de l'OMG [35].

Les PLs étant régis par les modèles, ils respectent eux aussi cette architecture, ainsi, la méta modélisation des PLs est organisée comme suit (figure-1.3-) :

- Le niveau M0 décrit les PLs.
- Le niveau M1 décrit les modèles de PLs.
- Le niveau M2 décrit les métamodèles de PLs.
- Le niveau M3 décrit les méta-métamodèles de PLs.

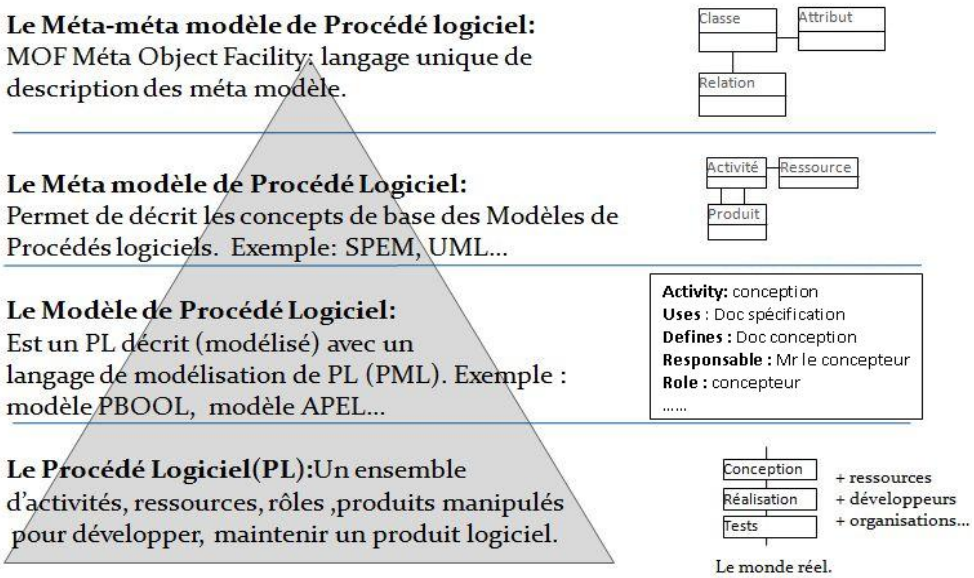


Figure 1.3 La méta modélisation des procédés logiciels.

1.3.1 Le niveau M0 : Les procédés logiciels

Un procédé est définit comme une suite d'opérations ou d'événements faisant intervenir des équipes de personnes, des outils et des techniques pour assurer le développement et la maintenance de produits ou de services [34]. Un Procédé Logiciel (PL) est, par conséquent, un procédé qui permet d'assurer le développement et la maintenance de produits logiciels.

Dans la littérature plusieurs définitions de PLs ont été avancées, nous citons celles que nous jugeons assez significatives :

"...The set of partially ordered process steps, with sets of related artifacts, human and computerized resources, organizational structures and constraints, intended to produce and maintain the requested software deliverables..." [73].

"...A software process can be defined as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product..." [48].

"...A software process is defined as a sequence of steps that must be carried out by the human agents to pursue the goals of software engineering..."[122].

"...software process means the set of activities required to produce a software system, executed by a group of people organized according to a given organizational structure and counting on the support of techno-conceptual tools..."[2].

"...Software process is a partially ordered set of activities undertaken to manage, develop and maintain software systems..."[2].

"... nous décrivons les procédés logiciels comme étant une suite d'étapes réalisées dans un but donné qui servent à gérer et assister le développement logiciel..." [49].

Toutes les définitions présentées s'accordent à définir le PL comme un enchainement d'unités de travail à suivre, de l'ensemble des produits logiciels requis et fournis par ces unités de travail, des équipes, outils, des techniques et des stratégies utilisés dont le but d'assurer le développement et la maintenance de produits logiciels (figure-1.4-). Concrètement, le PL représente le monde réel du développement.

1.3.2 Le niveau M1 : Les modèles de procédés logiciels

Un modèle est une abstraction, une simplification d'un système qui est suffisante pour comprendre le système modélisé [34].

"...a Process Model (PM) is an abstract description of an actual or proposed process that represents selected process elements that are considered important to the purpose of the model and can be enacted by a human or machine" [40].

Les travaux sur les PLs s'accordent, pour la plupart, sur la définition du modèle de PLs. Un modèle de PL est la représentation formalisée du PL, il explicite les propriétés et les variables qui régissent le monde réel du développement.

L'objectif principal du modèle de PL est de fournir une représentation plus au moins formelle pour

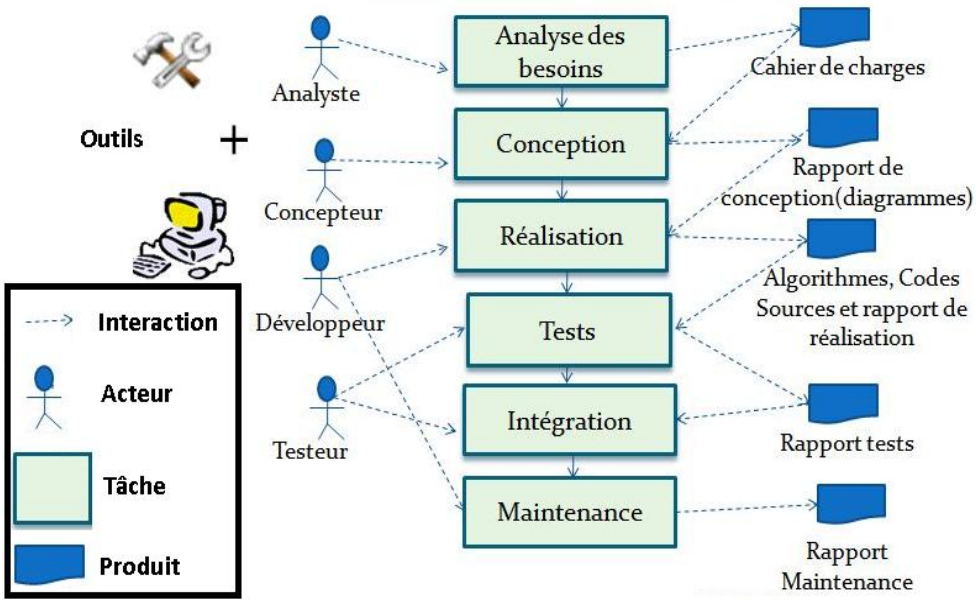


Figure 1.4 Le procédé logiciel (le monde réel du développement).

le développement logiciel. Il doit pendre en charge, entre autres, l'évolution prévue et imprévue [68] de la réalité du développement.

Différents modèles de PLs peuvent décrire différents angles de développement logiciel [2]. En effet, par soucis de clarté et de précision, les modèles de PLs peuvent se focaliser sur un aspect particulier du développement et reléguer d'autres au second plan. Plusieurs classifications de PLs ont été définies, ces classifications ont mis en avant les critères suivants :

- Le niveau de formalisation du modèle de PL : non formel, semi formel, exécutable [104].
- L'élément central du modèle de PL : centré activité, centré rôle, centré artefact..etc. [63].
- L'orientation du modèle de PL : orienté gestion de configuration, orienté décision, orienté stratégie...[63].
- Le type du langage de modélisation de PL utilisé : orienté objet, réseau de pétri, à base de règles... [2] [123] [22].
- Le type d'exécution supporté : exécution distribuée, simulation...etc. [2] [124] [22].

1.3.3 Le niveau M2 : Les métamodèles de procédés logiciels

Le métamodèle de PL est un modèle regroupant les concepts de base du langage d'expression d'un modèle de PL. Le métamodèle de PL doit prendre en considération les points de vue et les orientations des PLs. En effet, les concepts représentés varient selon l'orientation du PL, il existe, donc, autant de métamodèles que d'orientations de PLs [63]. Cependant, tous les PLs sont conformes au même noyau conceptuel (figure-1.5-) : Le PL est un enchainement d'activité, chaque activité a besoin de produits "produit de travail" en entrée (inputs) pour fournir des produits en sortie (outputs). Le PL étant centré humain, une activité est sous la responsabilité d'un rôle "Role".

Ainsi, les concepts de base sont :

- **Activité (Activity)** : représente l'unité de travail effectuée. Les enchainements et les états d'avancements des activités peuvent être décris dans une séquence d'activités.

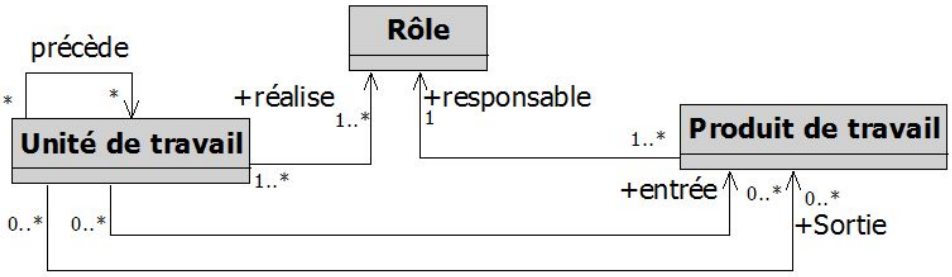


Figure 1.5 Le Métamodèle noyau du procédé logiciel.

– **Produit de travail (Work Product)** : représente les produits manipulés. Les transmissions, le format, les versions et le stockage de ces produits peuvent être traités aussi.

– **Rôle (Role)** : décrit en général les responsabilités et qualifications requises ou existantes que peut ou doit avoir un acteur lors du développement logiciel.

Il est clair que d'autres concepts tels que stratégie, organisation, outil,...etc. peuvent être présents dans un métamodèle de PL, cela dépendra de l'orientation et de la spécialisation des modèles de PLs que l'on veut décrire.

Il existe un nombre important de métamodèles décrivant les concepts de base de différents modèles de PLs [63], nous citons : OOSPICE1 [112], OPEN2 Process Framework [95], la norme australienne Standard Metamodel for Software Development Methodologies [14], la norme ISO 24744 [109] ainsi que la norme de l'OMG, SPEM 1.1 (System and Software Process Engineering Metamodel) et sa nouvelle version SPEM 2.0 [88]. Dans la section suivante, Nous nous intéressons au métamodèle SPEM 2.0 définit par l'OMG comme standard pour la modélisation des PLs [88], il sera exploité dans notre solution de réutilisation de PLs.

1.3.4 Le niveau M3 : Les méta-métamodèles de procédés logiciels

Le méta-métamodèle est une représentation qui permet de décrire les concepts et les structures qui permettent de décrire les métamodèles.

L'OMG spécifie un méta-métamodèle unique et auto-descriptif qui est le MOF (Model Object Facility) [86]. Le MOF décrit les concepts de base de tous les métamodèles de tous les domaines y compris celui des PLs (figure-1.2-).

Dans notre thèse nous nous focalisons sur les niveaux modèle (M1) et le niveau métamodèle (M2) de procédés logiciels ; le niveau méta-métamodèle n'est pas traité dans notre travail.

1.4 Le métamodèle SPEM (Systems and Software Process Engineering Metamodel)

SPEM (Systems and Software Process Engineering Metamodel) est un standard adopté par l'OMG, c'est un métamodèle qui permet de décrire les concepts et principes de base pour la modélisation des PLs. L'objectif de SPEM n'est pas de devenir un langage générique de modélisation de PLs mais de couvrir la description des concepts d'un large éventail de PLs, sans se spécialiser dans un type particulier, tout en prenant en compte la diversité reconnue des PLs [88].

SPEM est un profile UML [1] , il se base sur les notations UML (profiles et modèles) pour décrire les concepts PLs sous différents angles.

1.4.1 Noyau conceptuel de SPEM

Le noyau conceptuel de SPEM est celui de tout PL [33], ainsi, les concepts de base de SPEM sont : "activité" appelée "Unité de travail", "Produit de travail", "Rôle" et les relations entre ces concepts.

Pour représenter ces concepts et les relations entre ces concepts, SPEM a défini quatre types de classes :

- Classes représentant la notion d'**unité de travail**, par exemple : Task Definition, Task Use, Activity...etc.
- Classes représentant **le réalisateur** de l'unité de travail , par exemple : Process Performer, Default Process performer...etc.
- Classes représentant **le produit de travail**, par exemple : Work Produt Definition, Work product Use...etc.
- Classes représentant la notion d'élément ; ces classes permettent de décrire d'autres concepts de PL tels que composant, outil, assistance,...etc. ainsi que d'autres concepts nécessaires à la modélisation des PLs (Process Component, Tool Definition, Guidance,...etc.).

Ces quatre types de classes sont bien agencés et sont présents à plusieurs niveaux de SPEM, ils permettent d'offrir plusieurs vues des concepts PLs selon différents angles.

1.4.2 Les métamodèles de SPEM

Pour couvrir les besoins de modélisation et d'exécution d'un large éventail de PLs, SPEM 2.0 est structuré en sept packages qui représentent sept métamodèles, chaque métamodèle décrit une vision particulière du PL (figure-1.6-).

SPEM est organisé en plusieurs métamodèles afin, d'une part, de faciliter la compréhension et l'extension de SPEM, et d'autre part, d'augmenter la réutilisation des concepts décris par SPEM, que se soit les structures des PLs, les connaissances des méthodes de développement ou la documentation et assistance. Ces métamodèles sont reliés à travers des "Merge", ce qui signifie que le contenu d'un métamodèle est étendu par le contenu du métamodèle relié.

Le métamodèle **"CORE"** est dédié à la description des concepts de base de SPEM, il fournit la base conceptuelle pour l'extension des autres métamodèles.

Le métamodèle **"MANAGED CONTENT"** fournit les concepts nécessaires pour la description de la documentation et de l'assistance qui peuvent être exploitées par des modèles de PLs. Il permet aussi de décrire des PLs non formels qui exploitent le langage naturel.

Le métamodèle **"PROCESS STRUCTURE"** contient les éléments de base pour la structuration d'un PL. Il décrit des fragments de PLs ou des enchainements d'activités qui sont indépendants de toute méthode de développement. Ce métamodèle permet de décrire des modèles de PLs ad-hoc qui n'ont pas de méthodes ou de cycles de vies particuliers, il permet aussi de décrire des patrons de PLs et des enchainements d'activités récurrents et réutilisables.

1. Un profile UML est une spécialisation du modèle UML pour un domaine d'utilisation particulier. Il est constitué de : 1)stéréotypes : qui permettent de rajouter des éléments de modélisation selon le domaine décrit, 2) Tagged Values : qui permettent de rajouter aux classes des informations spécifiques au domaine décrit, et 3) contraintes : qui permettent de préciser les conditions d'emploi des éléments UML selon le domaine décrit.

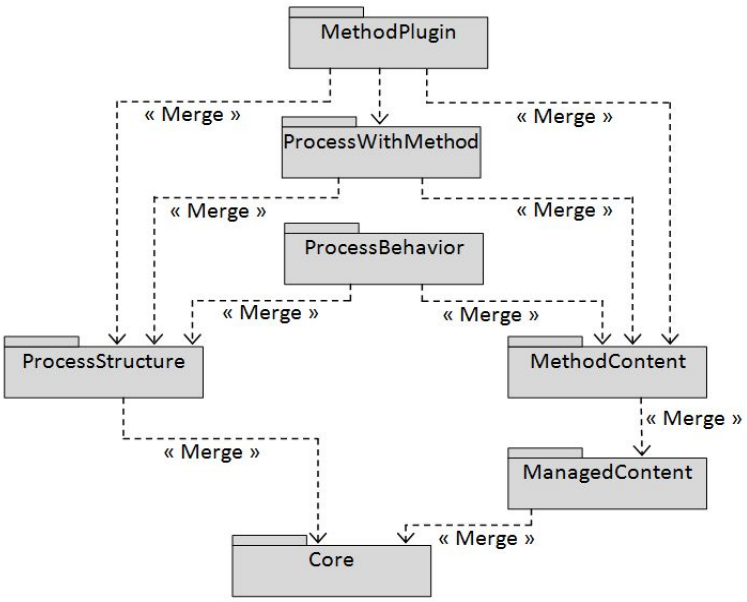


Figure 1.6 Les métamodèles de SPEM.

Contrairement, au métamodèle "PROCESS STRUCTURE", le métamodèle **"METHOD CONTENT"** fournit les concepts de base pour définir les méthodes, les techniques, et les meilleures pratiques de développement, réutilisables indépendamment d'un projet de développement ou d'une particulière structure de PL.

Comme son nom l'indique, Le métamodèle **"PROCESS WITH METHODS"** permet de décrire un PL structuré qui respecte les concepts du métamodèle "PROCESS STRUCTURE", et en même temps, qui suit une méthode de développement définie dans le métamodèle "METHOD CONTENT", plaçant le PL dans un contexte particulier et en respectant une méthode ou un cycle de vie particulier.

L'objectif du métamodèle **"METHOD PLUGIN"** est de fournir les outils pour utiliser et réutiliser les connaissances des métamodèles "METHOD CONTENT", "PROCESS STRUCTURE" et "PROCESS WITH METHODS". L'utilisation passe par la définition de configurations (Method Configurations) et de correspondances entre les éléments "METHOD CONTENT" et les éléments "PROCESS STRUCTURE" (Method Plugins).

Par contre, la réutilisation des PLs est matérialisée par l'introduction de la réutilisation à base de composants procédés. Ainsi, des concepts architecturaux tels que composant (Process Component), port (Work Product Port) et connecteur (Work Product Port Connectors) sont introduis, cependant, la réutilisation des PLs n'est qu'à ses débuts et plusieurs problèmes de réutilisation ont été soulevés [88].

La réutilisation des modèles de PLs et des comportements externes qui ne sont pas conformes à SPEM sont pris en charge par le package "PROCESS BEHAVIOR".

1.4.3 Organisation des classes de SPEM à travers les métamodèles

SPEM est composé d'un nombre important de classes ; à chaque concept de base du PL (Unité de travail, Produit de travail et Rôle) est associé à un nombre important d'éléments UML qui le décrivent selon plusieurs facettes.

Afin de capturer toute vision possible du concept décrit, des classes, des associations et des propriétés qui décrivent les concepts PLs sont définies et distribuées sur les différents métamodèles de SPEM.

Ce nombre important de concepts donne l'impression que SPEM est ambigu est difficilement compréhensible. Cependant, cette impression se dissipe dés que l'architecture de SPEM est cernée. Outre l'organisation de SPEM en packages et en extensions entre packages, les classes de SPEM sont bien agencées. En général, leur organisation respecte les règles suivantes :

- Chaque package (qui constitue un métamodèle) possède une classe abstraite qui décrit le comportement commun des éléments de ce package, excepté le package "METHOD PLUGIN" qui n'a pas de classe générique. La figure-1.7- représente les stéréotypes qui correspondent aux principales classes abstraites de SPEM.
- Chaque classe abstraite du package est la classe génératrice des classes décrivant les concepts de base du PL (selon la vision du package).
- Chaque concept de base du PL (unité de travail, produit de travail, rôle, relation entre ces concepts) a une classe correspondante dans chaque package SPEM.

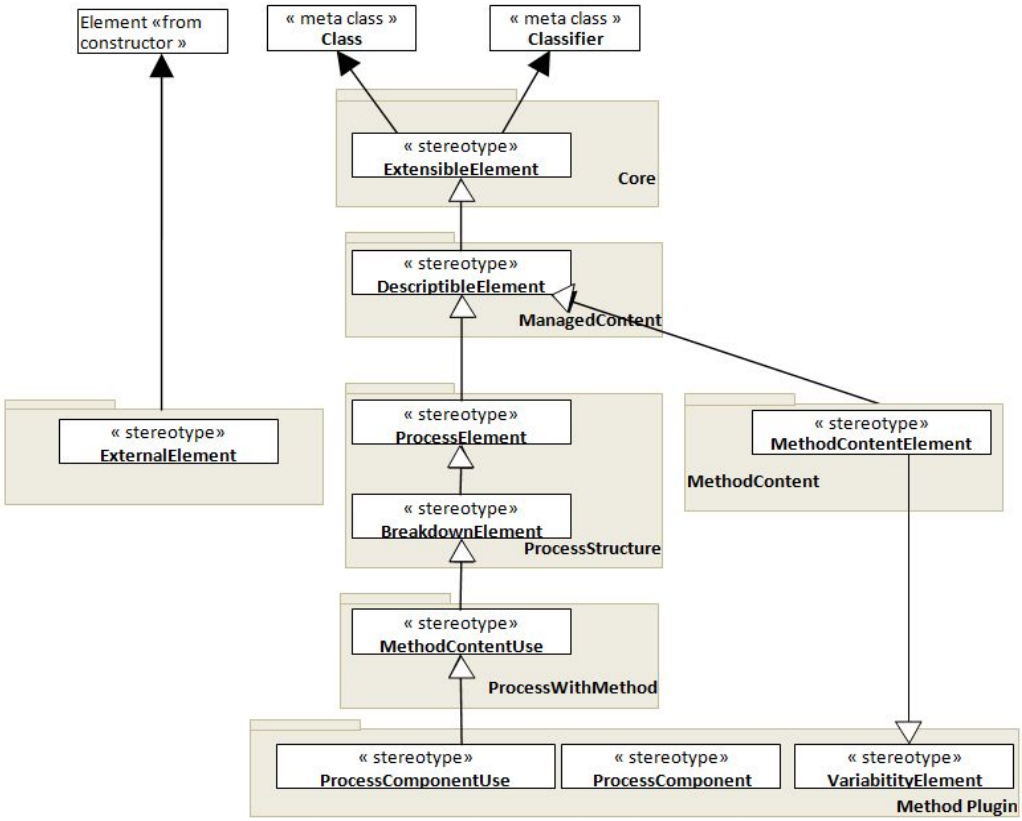


Figure 1.7 Les stéréotypes représentant les concepts abstraits de chaque métamodèle de SPEM.

1.5 Les singularités du modèle de procédé logiciel

1.5.1 Les différences entre produit logiciel et procédé logiciel.

Les PLs ont souvent été assimilés à des logiciels [96] [27] [60], il est certain que l'utilisation des langages de programmation et des formalismes dédiés aux logiciels à des fins de modélisation des PLs, en plus de l'importance de la qualité, du temps et du coût de la modélisation du PL justifient cette

assimilation. Cependant, les PLs ont des spécificités propres qui rendent leur modélisation particulière, et finalement différente de celle des logiciels. Deux caractéristiques principales différencient le PL du produit logiciel :

1.5.1.1 La dimension humaine

Les PLs sont centrés humain, en d'autres termes, Les PLs sont exécutés par des humains en association avec des machines [110]. L'exécution de certaines tâches telles que la conception d'un logiciel ou l'analyse des besoins se fait manuellement, elle dépend de la créativité, de l'intelligence et de la personnalité des intervenants (développeur, client, responsable...etc.). Par contre, l'exécution d'un logiciel se fait automatiquement et par la machine seulement, ainsi, contrairement à l'exécution du PL, l'intelligence et la créativité humaines n'influent pas sur la réussite ou l'échec de l'exécution du logiciel (lors de l'exécution).

1.5.1.2 Les instances d'exécution du modèle de procédé logiciel

Cette caractéristique est la conséquence directe de l'influence de la dimension humaine sur l'exécution du modèle de PL. Un modèle de PL peut s'exécuter "correctement" de plusieurs manières. La dimension humaine qui représente un facteur subjectif de l'exécution ; en plus des événements inattendus influent incontestablement sur l'exécution du modèle de PL. Ainsi, il existe autant d'instances d'exécution correctes que de contextes d'exécution possibles. Ces contextes d'exécution prennent en compte, souvent, les considérations humaines, sociales et culturelles des développeurs. Par contre, l'exécution correcte du logiciel est unique et ne dépend pas de la personnalité ou des traditions de son exécutant.

1.5.2 Les caractéristiques difficilement accordables du modèle de procédé logiciel

La qualité du PL a un impact direct sur la qualité du produit logiciel réalisé. Selon le standard ISO 9000 [66], *La qualité d'un logiciel est l'ensemble des caractéristiques intrinsèques qui lui confère l'aptitude à satisfaire les besoins et attentes exprimés ou implicites des clients et autres parties intéressées*.

Ainsi, un PL de qualité doit avoir des caractéristiques qui permettent de répondre aux exigences de ses utilisateurs.

Les caractéristiques des PLs identifiées dans les premiers travaux sur des PLs et jugées indispensables pour la réussite d'un projet logiciel telles que l'évolution, la flexibilité, et la compréhension [105] [68] [8] [40], sont toujours et restent les caractéristiques de base de tout PL de qualité. Néanmoins, un PL peut avoir certaines caractéristiques qui peuvent être vues comme opposées, ou du moins, difficilement conciliables. Assurément, le PL de qualité doit combiner entre :

1.5.2.1 La compréhension et la couverture des aspects de développement

Le modèle de PL est destiné à l'utilisation humaine, par conséquent, il doit être compréhensible, simple et clair. La pertinence des concepts PL rentrants dans la modélisation du PL a certainement un impact sur la compréhension du modèle de PL. D'autre part, la terminologie et le vocabulaire utilisés ont aussi un impact sur la compréhension du modèle de PL. En effet, malgré la mise en place des standards et des normes terminologiques [90] concernant le développement logiciel, les développeurs des PLs peuvent toujours utiliser leur propre vocabulaire qui correspond à leur réalité ou tradition de développement.

Par ailleurs, le PL doit avoir la puissance syntaxique et sémantique nécessaires pour exprimer tous les aspects traités lors du développement logiciel. Le large éventail des aspects de développement, ajouté à cela, le large éventail du vocabulaire utilisé rendent la couverture des aspects de développement assez difficile, ce qui justifie la modélisation de PLs spécialisés.

Ainsi, un PL de qualité doit trouver l'équilibre entre compréhension et couverture des aspects de développement.

1.5.2.2 L'abstraction et l'exécutabilité

Pour maîtriser la complexité croissante des PLs, l'abstraction des PLs et l'exploitation de leur structure deviennent de plus en plus indispensable.

Les classifications des PLs dans la littérature [124] [2] [64] [20] [8], nous a permis de constater que l'exécutabilité ainsi que l'abstraction des PLs sont difficilement conciliables. En effet, la modélisation des PLs exécutables repose, le plus souvent, sur les langages de programmation (fonctionnels, à base de règles...etc.). Par contre, les PLs descriptifs exploitent des formalismes de plus forte abstraction (diagrammes, représentation graphique...etc.), l'intérêt est de limiter la dépendance des PLs aux langages de bas niveaux et d'affranchir les intervenants (qui ne sont pas forcément des programmeurs) de la nécessité de maîtriser parfaitement le langage de programmation utilisé. Malheureusement, ces derniers n'offrent pas de support d'exécution et de contrôle d'exécutions pour les PLs, ils sont le plus souvent utilisés pour la documentation du PL.

Afin de concilier le modèle abstrait et le modèle exécutable du PL, plusieurs solutions ont été proposées [21] [85], néanmoins, cette préoccupation est toujours d'actualité et la conciliation de l'abstraction et de l'exécutabilité du PL reste un des critères de qualité du modèle de PL.

1.5.2.3 La gestion de l'exécution centrée humain et la gestion de l'exécution automatique

Le PL étant centré humain [3] [110], deux types d'exécution sont possibles : exécution automatique où les tâches sont exécutées par des machines (compilation, comparaison, conversion...etc.), et exécution manuelle où les tâches sont exécutées par l'humain ; ces dernières sollicitent le plus souvent l'intelligence et la créativité de l'humain (conception, programmation...etc.). Par conséquent, le PL doit s'accommoder avec ces deux types d'exécution et fournir un support qui gère cette différence.

1.5.2.4 Le contrôle d'exécution et l'agilité

Les nouvelles méthodes de développement dites "Agiles" [65] sont décrites comme itératives et incrémentales. Ces méthodes sont définies pour pallier la rigidité et le manque de réactivité des méthodes de développement traditionnelles. Elles intègrent de nouvelles pratiques pour stimuler la dynamicité du développement. L'auto-organisation des équipes de développeurs, l'introduction du client dans le cycle de développement, le travail soutenu mais sans surmenage, la diminution de la documentation sont autant de pratiques adoptées par les méthodes agiles.

Les solutions proposées par les méthodes agiles, ne résolvent pas entièrement les problèmes de rigidité des modèles des PLs [118]. En effet, le support de flexibilité et d'évolution offert par ces dernières est centré humain et se base sur les capacités humaines. Le manifeste des méthodes agiles [18] définit des valeurs (communication, collaboration...etc.) et des principes (conception simple, courage,...etc.) qui dépendent pour la plupart de la personnalité, de l'intelligence et de l'efficacité des intervenants ; le contrôle d'exécution dépend des facteurs subjectifs ce qui peut conduire à des déviations et des dépassements lors de l'exécution du modèle de PL. La dépendance du contrôle d'exécution aux capacités humaines est un

inconvénient pour un modèle de PL de qualité. Par conséquent, le PL de qualité doit pouvoir combiner agilité et contrôle d'exécution.

1.5.3 Les autres caractéristiques des procédés logiciels

Certaines caractéristiques dépendent du type du PL et du type de son exécution, ainsi, certains PLs peuvent avoir une ou plusieurs des caractéristiques suivantes :

- L'hétérogénéité : l'hétérogénéité peut être au niveau 1) des concepts : les fragments de PLs ne manipulent pas tous les mêmes concepts PL ; 2) du langage : les fragments de PLs ne sont pas tous modélisés avec le même langage de modélisation de PL ; 3) de la plateforme d'exécution : les fragments de PLs peuvent s'exécuter sur différents plateformes avec différents systèmes d'exploitations.
- La distribution : les fragments de PLs s'exécutent sur des machines différentes.
- L'évolution : les PLs doivent intégrer l'évolution comme une caractéristique à part entière [68].
- La dynamicité : les PLs peuvent s'adapter et être adapté lors de l'exécution.
- La simulation : en prenant en compte la complexité de certains PLs, la simulation des modèles de PLs au lieu de leur l'exécution des PLs est une des solutions adoptées pour l'analyse de leur comportement.

1.6 La modélisation des procédés logiciels par approche de réutilisation

La réutilisation consiste à développer un logiciel en partant systématiquement d'un stock de "briques de construction" afin d'exploiter les architectures et les besoins communs à plusieurs applications. La réutilisation a pour objectif l'amélioration de la qualité et de la productivité, ainsi que l'industrialisation de la production des logiciels.

La réutilisation des modèles de PLs est une des approches de modélisation les plus prometteuses ; l'utilisation même d'une approche de réutilisation confère aux PLs certaines qualités indispensables telles que la facilité de modification, l'adaptation et l'évolution [23] [47] [69].

Les résultats attendus de la réutilisation des PLs sont les mêmes que ceux attendus de la réutilisation des produits logiciels : réduire le coût et le temps de la modélisation du PL tout en augmentant la qualité du résultat.

Plusieurs raisons peuvent justifier notre choix de modélisation de PLs par approche de réutilisation :

- La répétitivité des tâches des modèles de PLs.
- L'existence des structures récurrentes et réutilisables telles que les cycles de vie de logiciels [28] et les partons de PLs [117].
- L'importance des interactions dans les modèles de PLs [6].

De plus, utiliser une approche de réutilisation permettra d'exploiter le savoir-faire et les meilleures pratiques capitalisées par des années d'expérimentation et de projets de réalisation.

1.6.1 Les difficultés de réutilisation spécifiques aux procédés logiciels

La mise en place d'une stratégie de réutilisation de PLs se heurte à des difficultés spécifiques au domaine des PLs, nous citons :

- La rigidité des PLs et leur dépendance à leur environnement de développement : L'abstraction joue un rôle important dans le processus de réutilisation. La rigidité des PLs et leur forte dépendance à leur environnent constituent un important obstacle à la réutilisation des PLs [49] [21].

- L'hétérogénéité des modèles de PLs : Afin de maîtriser les différents aspects de développement, un nombre important de concepts, de paradigmes, de langages et d'environnements ont été proposés pour formaliser les PLs. De ce fait, il est difficile de proposer une solution de réutilisation qui couvre plusieurs types de PLs.
- Les terminologies utilisées lors de la modélisation : Les modèles de PLs sont dédiés à l'utilisation humaine, par conséquent, ils doivent être compréhensibles par l'humain. Malgré l'existence de standard [90], la réutilisation de modèles de PLs peut se heurter à l'hétérogénéité du vocabulaire et de la terminologie utilisés.

1.6.2 Ingénierie pour la réutilisation des Procédés Logiciels

En général, l'ingénierie de réutilisation est organisée en deux grands axes : l'ingénierie "pour" la réutilisation et l'ingénierie "par" la réutilisation.

L'ingénierie pour la réutilisation consiste à offrir le support nécessaire pour la production d'entités logicielles réutilisables. Une entité logicielle réutilisable est un produit logiciel (corps) associé aux critères qui le décrivent et qui permettent sa réutilisation (description).

Nous utilisons le terme "entité" pour signifier que tout produit logiciel ou PL quelque soit son degré de formalisation : non formel, semi formel, formel peut être considéré comme une entité réutilisable.

En se limitant au domaine des PLs, nous définissons une entité de PL réutilisable comme un fragment de PL associé aux critères qui le décrivent et qui permettent sa réutilisation.

L'ingénierie pour la réutilisation de PLs met en place les stratégies nécessaires pour offrir des fragments de PLs facilement réalisables, cela se résume à :
- Identifier les entités PLs réutilisables (corps et description).
- Assurer les caractéristiques nécessaires à sa réutilisation (modularité, auto-description, opérabilité, compréhension, facilité de réutilisation, composition).
- Offrir le support de stockage adéquat (catalogue, base de données, base de connaissances, ontologies, librairies...etc.).
- Offrir les outils nécessaires pour la classification et le tri des fragments de PLs réutilisables.

1.6.2.1 La réutilisation des modèles de procédés logiciels à base d'ontologies

Une des préoccupations de l'ingénierie pour la réutilisation est d'offrir au domaine concerné un support de stockage adéquat.

Pour couvrir l'ingénierie pour la réalisation des PLs, nous explorons les approches de réutilisation à base d'ontologies de domaine. Le choix de l'ontologie comme support de stockage est justifié par :
- L'ontologie de domaine permet de gérer l'hétérogénéité des PLs et de capturer l'essence du PL indépendamment de son formalisme d'origine.
- Les PLs étant centrés humain, de ce fait, les connaissances, les expériences et le savoir-faire des développeurs PLs ont un impact sur la qualité du modèle de PL, ils peuvent être stockés et réutilisés formellement.
- L'ontologie de domaine permet l'émergence de nouvelles solutions par inférence de connaissances PL. Ces dernières sont extraites à partir des connaissances de modèles de PLs hétérogènes et qui n'ont pas été combinées auparavant.
- L'ontologie de domaine, si elle est acceptée par la communauté peut être partagée et réutilisée pour d'autres travaux sur les PLs.

1.6.3 Ingénierie par la réutilisation des Procédés Logiciels

L'ingénierie par la réutilisation consiste à offrir le support nécessaire pour la consommation des entités logicielles réutilisables préparées lors de la phase de l'ingénierie pour la réutilisation. Il s'agit de la réutilisation effective des entités logicielles et la mise en place d'un résultat opérationnel final.

L'ingénierie par la réutilisation de PLs consiste à offrir les mécanismes nécessaires pour réaliser des PLs opérationnels qui répondent à la demande de l'utilisateur à partir des fragments de PLs réutilisables.

Les principaux objectifs de l'ingénierie par la réutilisation consistent à définir des stratégies et algorithmes pour :

- La recherche et la sélection des entités à réutiliser.
- L'adaptation des entités réutilisables sélectionnées.
- Le remplacement des entités non conformes.
- L'assemblage des entités réutilisables et la génération du résultat final.
- La vérification de la cohérence et validation du résultat.

1.6.3.1 La réutilisation des procédés logiciels à base d'architectures logicielles

La préoccupation de l'ingénierie pour la réutilisation des PLs est d'offrir les stratégies adéquates pour la réutilisation effective des fragments de PLs.

La réutilisation à base d'architecture logicielles (ALs) couvre les deux ingénieries (pour et par) :

- Pour la réutilisation, en définissant des blocs réutilisables tels que les composants, les connecteurs, et les configurations. Mais aussi, en définissant de styles architecturaux qui permettent de capturer les spécificités des structures récurrentes.
- Par la réutilisation, en permettant le déploiement des ALs et la génération de leur code source, et en permettant la vérification et la validation du résultat final.

Pour couvrir l'ingénierie par la réutilisation de PLs, nous explorons les approches de réutilisation à base d'ALs. Ce choix est justifié par :

- La possibilité de gérer les interactions des PLs indépendamment des traitements. Cet possibilité est réalisable en définissant des connecteurs de PLs.
- La possibilité d'exploiter la vue abstraite du PL en définissant des configurations de PLs.
- La possibilité d'exploiter le mécanisme de déploiement d'ALs pour générer de manière, plus au moins automatique, le résultat final.

1.7 Conclusion : synthèse et analyse

Malgré les avancées avérées et la maturité du domaine de l'ingénierie des PLs, aucune tendance de modélisation de PLs ne s'est réellement imposée comme référence dans le domaine des PLs. La modélisation et l'exécution des PLs sont toujours d'actualité ; les propositions actuelles de modélisation de PL sont plutôt tournées vers l'amélioration des PLs (SP impovement)[83] [114] et leur simulation que vers la création de nouveaux langages ou environnements. En prenant en compte la maturité du domaine de l'ingénierie des PLs, la réutilisation des PLs semble une solution logique à l'amélioration les modèles de PLs.

En parcourant la littérature sur la réutilisation des PLs, nous constatons que : Le "quoi" réutiliser est bien plus facile à cerner que le "comment" réutiliser :

Les tâches répétitives, les structures récurrentes, les meilleures pratiques, les stratégies de développement et le savoir-faire des développeurs PLs sont autant de sources pertinentes pour définir des entités de

PLs réutilisables et pouvant alimenter le "quoi" réutiliser. Par contre, le large éventail des types de PLs, l'hétérogénéité des outils et le nombre important de formalismes de modélisation de PLs rendent difficile la mise en place des solutions de réutilisation de PLs, et constituent l'obstacle principal du "comment" réutiliser.

L'objectif de notre travail est de proposer une approche de réutilisation de PLs et de contribuer à l'amélioration de la qualité des PLs en proposant une solution permettant non seulement d'offrir les caractéristiques indispensables au PL de qualité, mais surtout, de combiner les caractéristiques des PLs décrites comme difficilement conciliables.

Les caractéristiques de base que nous jugeons indispensables pour un PL de qualité et qui sont visées par notre contribution concernent :

- La compréhension ;
- La couverture des aspects de développement ;
- La gestion de l'exécution centrée humain ;
- La gestion de l'exécution automatique ;
- L'abstraction ;
- L'éxécutabilité,
- Le contrôle d'exécution ;
- L'évolution.

En plus de surmonter rigidité des PLs, leur dépendance à leur environnement de développement et leur forte hétérogénéité, notre solution de réutilisation de PLs doit couvrir les deux ingénieries : pour et par la réutilisation. Ainsi, pour la mise en place de notre approche, notre choix s'est porté sur l'étude de la possibilité de combiner les ALs et les ontologies de domaines. Par conséquent, le chapitre suivant est dédié à la présentation des ALs et des opportunités qu'elles peuvent offrir à la réutilisation des PLs.

Notre contribution principale est d'exploiter les ALs au bénéfice de la réutilisation des PLs, l'ontologie est exploitée comme un socle qui permet de capitaliser les connaissances des PLs, les concepts de base des ontologies de domaine sont détaillés dans l'annexe -2-.

CHAPITRE 2

Les architectures logicielles au service de la réutilisation des procédés logiciels

2.1 Introduction

Après la crise du logiciel [46], la complexité des logiciels a permis la remise en cause de la qualité des Procédés Logiciels (PLs) utilisés lors de leur développement. Incontestablement, les PLs doivent offrir un support pertinent pour prendre en charge cette complexité croissante. Malheureusement, la complexité des logiciels s'est répercutée sur les PLs qui sont devenus, eux mêmes, complexes et difficiles à modéliser et à exécuter. De la même manière que pour les logiciels, les Architectures Logicielles (ALs) peuvent contribuer à la maîtrise de la complexité des PLs. En outre, les ALs sont une solution à la réutilisation des logiciels ; la séparation des préoccupations, l'abstraction, la définition de blocs réutilisables sont autant de caractéristiques que les ALs exploitent pour promouvoir la réutilisation des logiciels. *"...If open architectures are good for software product reuse, then their process counterparts will be good for software process reuse..."* [25] ainsi, selon Boehm si les ALs apportent des solutions à la réutilisation des logiciels, elles peuvent surement contribuer à l'amélioration de la réutilisation des PLs.

Dans ce chapitre, nous nous intéressons aux ALs, mais plus particulièrement à la contribution qu'elles peuvent apporter à la réutilisation des PLs. Notre objectif n'est pas de tracer l'historique et l'évolution des ALs, car plusieurs travaux ont été faits dans ce sens [45] [108] [91], mais plutôt de présenter les concepts de base tels qu'ils sont définis et exploités actuellement et sur lesquels nous nous appuierons dans nos travaux.

Cette démarche est motivée par le désir d'étudier la faisabilité, les difficultés et les avantages que peuvent apporter les ALs pour réutiliser les PLs, car, dans notre travail, nous préconisons à définir et exploiter des architectures de PLs.

Ainsi, ce chapitre est dédié à la présentation des ALs, nous les définissons et énumérons les avantages de leur utilisation. Ces avantages sont identifiés de manière à évaluer la contribution des ALs à la réutilisation des PLs.

Aussi, la notion de langages de description d'architectures (Architecture Description Languages -ADLs-) est définie. Comme exemple d'ADL, nous présentons l'ADL ACME que nous jugeons approprié à la description des architectures de PLs. Par la suite, nous définissons les éléments architecturaux de base des ALs, leurs caractéristiques et les styles architecturaux.

Nous terminons ce chapitre par l'étude de faisabilité de description de PLs à base d'ALs. Afin d'anticiper la description des architectures de PLs, nous introduisons des correspondances entre éléments PLs (identifiés dans la chapitre précédent) et les éléments architecturaux (identifiés dans ce chapitre).

2.2 Les architectures logicielles

2.2.1 Définitions

L'accroissement de la complexité des systèmes à développer en taille, architecture, standardisation, intégrations...etc. a obligé les développeurs à revoir leur vision du logiciel. Le besoin d'abstraire et d'architecturer les logiciels avant de les développer est alors devenu indispensable. C'est dans ce contexte que les ALs se sont imposées comme solution à la maîtrise de la complexité des logiciels.

Même si la plupart des définitions de l'architecture logicielle convergent, aucune définition ne s'impose comme référence pour décrire les ALs. Il est évident que la définition des ALs a évolué avec les avancées des travaux dans ce domaine, et donc, a été raffinée, augmentée et améliorée [100] [116] [50] [54] [45] [71].

Dans le cadre de notre travail, nous ne traçons pas l'évolution des ALs nous avançons seulement les définitions que nous jugeons pertinentes pour notre contribution.

Selon Garlan et Perry [54] "A Software Architecture is the structure of the components of program/system, their interrelationships, and principles and guidelines governing their design and evolution over time"

Aussi Bass et al [17] décrivent les ALs comme suit : *"...The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them..."*

Plus précisément, nous considérons une architecture logicielle comme un ensemble d'unités de traitements ou de stockage (composants), qui interagissent à travers des unités d'interactions (connecteurs). Cet assemblage (configuration) est régi par des règles et des contraintes de composition qui prennent en compte les propriétés des éléments architecturaux entrant dans la configuration.

Ainsi, les éléments architecturaux de base de toute architecture logicielle sont : composant, connecteur et configuration.

2.2.2 Les avantages des architectures logicielles

La dissociation entre conception et implémentation, la focalisation sur la vue abstraite du système en masquant les détails de l'implémentation, la séparation des préoccupations, la définition de plusieurs types de blocs réutilisables (composant, connecteur, configuration), sont les mécanismes fondamentaux des ALs, ainsi grâce à ces mécanismes, les ALs permettent de :

- **Augmenter la réutilisation :** les ALs prônent la réutilisation de logiciels, en effet, toutes les caractéristiques qui seront citées ci-dessous permettent d'augmenter la réutilisation.
- **Maîtriser la complexité des logiciels :** fournir une vue abstraite et représenter le logiciel quelque soit sa taille en modules indépendants permet de maîtriser sa complexité.
- **Favoriser la compréhension :** une AL permet de rendre explicite les décisions architecturales et les raisons de ces décisions. Ceci facilite considérablement la compréhension de l'organisation de systèmes larges et complexes.
- **Offrir un traitement particulier aux interactions :** la définition de connecteurs offre un support spécifique aux interactions et permet de personnaliser et cerner les différentes interactions dans un

système.
- **Faciliter l'évolution :** une architecture logicielle peut exploiter des stratégies d'évolution du système. Ceci permet de mieux gérer la propagation des changements et d'évaluer les coûts associés à l'évolution. Les évolutions d'ALs peuvent être de différents types : dynamique, statique et peuvent concerner les différents éléments architecturaux [103] [91]. L'étude de l'évolution constitue en soi un domaine de recherche assez vaste dans les ALs.
- **Offrir la possibilité d'analyse :** une AL permet de raisonner sur la qualité du système qui va être implémenté (ses performances, sa maintenabilité...etc.). En outre, elle permet de faire des vérifications de conformité par rapport à certaines contraintes de style, des vérifications de cohérence, et des corrections par exemple.

Aussi, les ALs permettent de :
- **Prendre en compte l'exécution :** les ALs traitent l'aspect conceptuel des systèmes, cependant, même s'ils sont peu nombreux, des mécanismes de génération de code sont proposés et intégrés à certains ADLs tels que UNICON-2[107] et ArchWare[4] : le mécanisme de déploiement des ALs permet d'offrir la caractéristique d'exécutabilité aux ALs et de passer automatiquement du niveau abstrait au niveau concret.
- **Capitaliser l'expertise des architectes systèmes :** l'expertise du domaine des ALs peut être formalisée puis réutilisée à travers la notion de style architectural.

2.2.3 L'apport des architectures logicielles aux procédés logiciels

Dans la section précédente, nous avons avancé les avantages des ALs, ces avantages peuvent être mis au service de la modélisation des PLs, le tableau-2.1- permet de donner une première réponse à la manière de fournir les caractéristiques requises par le PL.

Toutes les caractéristiques PLs trouvent réponse dans le domaine des ALs, exceptée "la couverture des aspects de développement" qui dépendra du langage de modélisation de PL utilisé lors de la génération du code PL, et la "gestion de l'exécution centrée humain" qui exigera un traitement spécifique qui, a priori, n'est pas fournit par les ALs.

2.3 Les Langages de Description d'Architectures (ADLs)

2.3.1 Définition

Selon Shaw et Garlan [55] : *"Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns".*

Une AL est un niveau de conception qui exige la description de ses éléments architecturaux, des interactions à travers ces éléments et des patterns qui guident leur composition. Les ADLs (Architecture Description Languages) ont été développés à cet effet. Les ADLs fournissent des notations formelles pour décrire et analyser l'architecture conceptuelle des systèmes logiciels.

2.3.2 Les caractéristiques des ADLs

Un ADL fournit aussi bien la syntaxe concrète que le cadre conceptuel pour caractériser des ALs [55]. Le cadre conceptuel concerne la sémantique qui reflète les caractéristiques du domaine pour lequel l'ADL est prévu. En effet, un nombre important d'ADLs a été développé, ces ADLs sont souvent dédiés

Caractéristiques des PLs	**Avantages des ALs**
Réutilisabilité	Augmente la réutilisation.
Compréhension	Augmente la compréhension.
Simplicité	Maîtrise la complexité.
Couverture des aspects de développement	–
Abstraction	Exploite l'abstraction.
Exécutabilité	Prise en compte de l'exécution par génération de code exécutable.
Gestion de l'exécution centrée humain	–
Gestion de l'exécution automatique	Offre un traitement particulier des interactions à travers la définition de connecteurs.
Agilité et adaptation	Offre un traitement particulier des interactions à travers la définition de connecteurs.
Évolution	Prise en compte de l'évolution et offre la possibilité d'analyse.

Table 2.1 Correspondances adoptées entre les caractéristiques exigées par les PLs et les avantages offerts par des ALs.

à des domaines particuliers, ils ne donnent pas forcement la même interprétation aux différents éléments architecturaux et n'utilisent pas forcement la même notation syntaxique [79] [78].

Les concepts de base d'une description architecturale sur lesquels s'accordent l'ensemble des travaux portant sur les ADLs sont : les composants, les connecteurs et les configurations. Ainsi, un ADL doit fournir les notations et les sémantiques nécessaires pour la spécification de ces concepts de base. Le tableau-2.2- résume les éléments architecturaux de base qui sont pris en compte par chaque ADL, nous remarquons que la plupart des ADLs définissent les concepts : composant, connecteur ainsi que les interfaces composant, par contre, le concept de configuration et interface connecteur ne sont pas toujours traités par ces ADLs.

Aussi, le tableau-2.2- présente les domaines d'application des ADLs, nous remarquons que les domaines d'applications divergent et qu'il n'y a pas de tendance particulière.

D'autres parts, les ADLs sont généralement accompagnés par divers outils destinés à décrire, analyser et simuler les ALs, et dans certains cas, générer le code des systèmes modélisés. Un certain nombre d'ADLs offre également un support pour modéliser le comportement et les contraintes sur les propriétés des composants et des connecteurs.

Par conséquent, avant de choisir l'ADL qui permettra de décrire une architecture particulière, il est nécessaire de faire une analyse préalable et étudier l'apport de chaque ADL en concept, domaine traité et outillage existant.

Nous remarquons aussi qu'il n'y pas d'ADLs spécifiques aux PLs, ce qui peut être à priori un inconvénient pour la modélisation des architectures de PLs. Ainsi, pour décrire une architecture de PL, deux possibilités s'offrent à nous : soit nous utilisons un ADL qui existe déjà en l'adaptant au domaine des PLs, soit nous créons un nouvel ADL pour les PLs. Nous avons opté pour la première solution afin de bénéficier de tous les outils s'y rapportant. Ainsi, dans la section suivante nous présentons l'ADL ACME car nous estimons que cet ADL a toutes les caractéristiques attendues pour décrire les architectures de PLs.

ADL	Concepts manipulés	Domaine d'applications
C2 [77]	Configuration, Composants, Connecteurs, Interfaces de composants	Systèmes distribués évolués et dynamiques
ACME [56]	Configuration, Composants (composites), Connecteurs, Interfaces de composants et de Connecteurs	Langage d'échange d'architectures
UniCon [107]	Composants (composites), Connecteurs, Interfaces de composants et de connecteurs	Systèmes temps réel
Wright [5]	Configuration, Composants, Connecteurs, Interfaces de composants et de connecteurs	Définition et analyse du comportement dynamique des systèmes concurrents
ArchJava [4]	Composants (composites), Connecteurs, Interfaces de composants et de connecteurs	Modèles exécutables
Sofa [101]	Configuration, Composants (composites), Connecteurs Interfaces de composants et de connecteurs	Systèmes à base de composants
Fractal [31]	Composants (composites), Connecteurs, Interfaces de composants	Systèmes à base de composants
Darwin [75]	Composants (composites), Connecteurs, Interfaces de composants	Systèmes distribués
UML [94]	Composants (composites), Connecteurs, Interfaces de composants	Systèmes à base d'objets

Table 2.2 Domaine d'application et élément architecturaux de base des principaux ADLs [108].

2.3.3 Exemple d'ADL : ACME

"Acme is a simple, generic software architecture description language (ADL) that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools." [56]

ACME est un langage de description d'architectures qui a pour but de fournir un ensemble assez complet de concepts pour décrire la structure architecturale d'un système. Les composants et les connecteurs, les propriétés et les contraintes, les types et les styles architecturaux, sont les principaux concepts pris en compte par ACME pour la description d'ALs.

Bien qu'ACME soit générique et non dédié à un domaine particulier, il peut ne pas être approprié à la description de certaines architectures telles que les architectures dynamiques. Malgré cela, il offre une bonne introduction à la modélisation architecturale, et un moyen facile de décrire un large éventail d'ALs.

L'autre objectif principal d'ACME est de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADLs ; un langage compatible avec leurs terminologies et qui permet d'intégrer facilement de nouveaux ADLs. Car, beaucoup d'ADLs ont des points en communs dans leur manière de décrire et d'analyser une AL. En effet, la plupart des ADLs fournissent des notions similaires comme le composant et le connecteur. ACME apparait alors comme un langage fédérateur de tous les ADLs existants.

Les fonctions principales d'ACME sont les suivantes [53] :

– ACME fournit un ensemble de sept concepts permettant de spécifier la structure d'une architec-

ture ; il s'agit du : composant, connecteur, système, port, rôle, de la représentation et de la carte de représentation (rep-map).
- ACME fournit un mécanisme d'annotation favorisant l'intégration d'informations spécifiques ne concernant pas la structure et apportées par certains ADLs comme la spécification comportementale de Wright [108].
- ACME offre un mécanisme permettant de créer et de réutiliser des conceptions comme le style d'AL.
- ACME offre un canevas favorisant l'intégration d'outils facilitant la spécification formelle de la sémantique d'une AL (open semantic framework).

2.3.4 ACME au service des procédés logiciels

Étant donné qu'il n'y pas d'ADLs spécifiques aux PLs, et afin de décrire des architectures de PLs, notre choix s'est porté sur l'ADL ACME [56], nous pensons qu'il possède les caractéristiques nécessaires pour décrire des architectures de PLs sans ambigüité, ce choix est motivé par les raisons suivantes :
- ACME est générique donc n'est pas dédié à un domaine particulier.
- Il intègre un ensemble complet d'éléments architecturaux par rapport aux ADLs existants ; les concepts de connecteur explicite, d'interface de connecteur et de style architectural sont représentés explicitement.
- Il est considéré comme un langage fédérateur et peut intégrer les notations d'autres ADLs ; ce qui nous permet d'intégrer d'autres aspects que l'ADL ACME ne permet pas de décrire tel que l'aspect dynamique de l'architecture de PL.
- Il possède son propre outil "ACME studio" [106] et donc peut nous permettre d'expérimenter nos propositions.

2.4 Définition des éléments architecturaux de base

Il existe différentes définitions des éléments architecturaux composant, connecteur et configuration. Néanmoins, tous partagent une base conceptuelle similaire. Dans ce qui suit, nous définissions les éléments architecturaux essentiels dans le domaine des ALs. Nous utilisons les notations ACME pour décrire ces éléments architecturaux.

2.4.1 Le composant

Dans la littérature plusieurs définitions ont été avancées pour le concept composant, le chapitre 11 "What others say" du livre de Clemens [116] cite les différentes définitions apportées au composant logiciel. Nous citons deux définitions que nous jugeons pertinentes.

Selon Szyperski [116] :

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Aussi, Jed Harris, président du CI Lab, a donnée la définition suivante d'un composant, en 1995 [116] :

"Un composant est un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support. De plus, il est doté d'interfaces standards pour pouvoir interopérer."

Ainsi, les composants sont définis comme des unités de composition qui décrivent et/ou assurent des fonctions spécifiques de calcule et de stockage de données, ils possèdent des interfaces d'interaction. Ils peuvent être déployés indépendamment et composés avec d'autres composants [108].

Les composants doivent être réutilisables et par conséquent, composables, auto descriptifs et autonomes [97].

2.4.2 Le connecteur

Les connecteurs déterminent les interactions entre les composants [52].

Selon Medvidovic and taylor [78] :

"...Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions..."

Ainsi, les connecteurs sont des blocs architecturaux qui formalisent les interactions entre composants et les règles qui gouvernent ces interactions. Les connecteurs peuvent être de formes simples d'interaction, comme les appels de procédures ou les évènements, ou bien de formes plus complexes, telles qu'un protocole client-serveur ou un lien SQL entre une base de données et une application [80] [84].

De manière générale, dans les ADLs les connecteurs peuvent être classés en trois groupes :

- Les connecteurs implicites qui n'ont pas d'existence propre comme ceux définis dans Darwin [75], MetaH [24] et Rapide [74].
- Les connecteurs prédéfinis qui sont concrets mais qui sont énumérés et ont une structure prédéfinie, comme ceux d'UniCon [107] et C2 [77].
- Les connecteurs explicites qui ont une existence propre et dont les sémantiques sont définies par les utilisateurs, comme ceux de Wright [5] et ACME [56].

2.4.3 La configuration

Selon Medvidovic and Taylor [78] *"Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure".*

Une configuration[1] représente un graphe de composants et de connecteurs et définit la façon dont ils sont reliés entre eux (figure-2.1-). Cette notion est nécessaire pour déterminer si les composants sont bien reliés, que leurs interfaces s'accordent, que les connecteurs correspondants permettent une communication correcte et que la combinaison de leurs sémantiques aboutit au comportement désiré. En plus des modèles de composants et de connecteurs, les descriptions des configurations permettent l'évaluation des aspects distribués et concurrents d'une AL.

2.4.4 L'interface

Les interfaces sont des points de communication qui permettent à un élément architectural (composant, configuration, connecteur) d'interagir avec son environnement, et avec les autres éléments. [55][17] [98] [1]

Dans l'ADL ACME et comme dans plusieurs ADLs, les interfaces d'un composant (configuration) portent le nom de "ports" (figure -2.2-). En général, pour identifier la direction des services offerts ou requis par le composant (respectivement la configuration) deux types de ports peuvent être distingués : les ports fournis qui exportent les services des composants, et les ports requis qui importent les services vers

1. Dans ACME la configuration est notée "System".

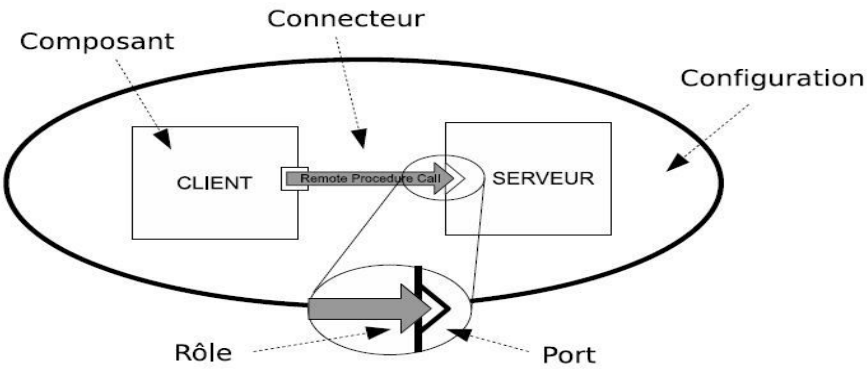


Figure 2.1 Les éléments architecturaux de base d'une architecture logicielle [56].

```
System simple_cli_serv = {
        Component client = {
             Port send-request
        }
        Component server = {
             Port receive-request
        }
        Connector rpc = {
             Roles {caller, callee}
        }
        Attachments {
                client.send-request to rpc.caller ;
                server.receive-request to rpc.callee
        }
    }
```

Figure 2.2 Les éléments architecturaux de base d'une architecture logicielle décrit à l'aide de L'ADL ACME.

les composants [108]. Chaque port identifie un point d'interaction entre le composant et son environnement. Un composant peut fournir des interfaces multiples en employant différents types de ports. Un port peut représenter une interface aussi simple comme une signature, ou des interfaces plus complexes, telles qu'une collection d'appels de procédures qui doivent être appelées dans un ordre spécifique.

Les interfaces d'un connecteur portent le nom de "rôles" (figure -2.2-). Deux types de rôles peuvent être définis : le rôle fourni et le rôle requis [108], chaque rôle d'un connecteur définit un participant de l'interaction représentée par le connecteur.

Deux types d'interaction entre interfaces sont définis : attachements et bindings.

- Attachement (enchainement) : représente la connexion entre un port d'un composant et un rôle d'un connecteur.
- Binding (délégation) : dans une description architecturale un composant et une configuration peuvent être composés hiérarchiquement d'un ou plusieurs sous-composants. Les connexions entre les ports d'une configuration ou d'un composant composite et les ports de leurs sous composants sont appelées délégation (bindings). De même que pour les connecteurs, les connexions entre les rôles d'un connecteur composite et les rôles de ses sous connecteurs sont aussi appelées délégation (bindings).

2.5 Les caractéristiques communes aux éléments architecturaux de base

Les éléments architecturaux de base ont des caractéristiques qui permettent de les spécifier [108] [97] [78], ces caractéristiques sont comme suit :

2.5.1 Les propriétés

Chaque élément architectural possède des propriétés ; les propriétés servent en général à documenter les détails d'une configuration, d'un composant ou d'un connecteur, relevant de leur conception et de leur analyse. Elles deviennent utiles lorsqu'elles sont utilisées par des outils à des fins de manipulation, d'affichage et d'analyse [91]. Il existe deux types de propriétés : les propriétés fonctionnelles et les propriétés non fonctionnelles.

Les propriétés fonctionnelles peuvent concerner aussi bien la structure, le comportement ou les fonctionnalités de l'élément architectural. Elles peuvent être paramétrées selon un contexte d'exécution particulier.

Les propriétés non fonctionnelles représentent d'autres besoins tels que la sécurité, la portabilité ou la performance[108].

```
System simpleCS = {
        Component client = {
          Port send-request;
           Properties {
              Aesop-style: style-id = client-server;
                 /* propriété permettant
                 /* d'indiquer que le composant client dans ACME
                 /* est défini comme un style dans Aesop
              UniCon-style : style-id = cs;
              source-code : external = "CODE-LIB/client.c" }}

        Component server = {
          Port receive-request;
           Properties {
              idempotence : boolean = true;
              max-concurrent-clients : integer = 1;
              source-code : external = "CODE-LIB/server.c" }}

        Connector rpc = {
          Roles {caller, callee}
           Properties {
                          synchronous : boolean = true;
                          max-roles : integer = 2;
                          protocol : Wright = "... " }}
        Attachments {
                          client.send-request to rpc.caller ;
                          server.receive-request to rpc.callee }
   }
```

Figure 2.3 Les propriétés et les contraintes d'une AL décrites à l'aide de L'ADL ACME.

La figure -2.3- illustre un exemple de propriétés décrites à l'aide de l'ADL ACME, l'avantage de cet ADL est qu'il est possible d'intégrer d'autres ADLs et de décrire les contraintes avec d'autres langages, ce qui lui confère l'aptitude à couvrir les besoins les plus spécifiques de description d'ALs.

2.5.2 Les contraintes

Les contraintes sont considérées comme des propriétés spécifiques et permettent de spécifier des restrictions sur les éléments architecturaux sur lesquels elles s'appliquent. Elles représentent les moyens permettant à un modèle d'architecture de rester valide durant sa durée de vie et de prendre en compte l'évolution et le remplacement des composants (resp. configurations, connecteurs) dans cette architecture

[50]. Ces contraintes peuvent inclure des restrictions sur les valeurs permises de propriétés, sur l'utilisation d'un service offert par un composant (resp. configuration, connecteur) et garantir par exemple la validité des résultats retournés par ce service.

Les contraintes peuvent être définies soit dans un langage de contraintes séparé tel que le langage de contraintes Armani [82] qui peut être utilisé avec l'ADL ACME, soit directement en utilisant les notations de l'ADL hôte.

2.5.3 Les sémantiques

La sémantique du composant (resp. connecteur) est exprimée en partie par son interface. Cependant, l'interface ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements [78].

2.5.4 Les types

Un type est l'abstraction qui permet d'encapsuler les fonctionnalités que peut fournir un élément architectural [108].

Le type de l'élément architectural (composant, connecteur) peut être instancié plusieurs fois dans une même AL ou peut être réutilisé dans d'autres ALs. Les instances d'un type ont la même structure et le même comportement que celui du type. Cette abstraction permet non seulement d'augmenter la réutilisation des ALs, mais aussi de faciliter la compréhension et l'analyse de ces dernières.

Selon les fonctionnalités de chaque élément architectural (composant et connecteur) le type du composant permet d'encapsuler des fonctionnalités de traitement et de stockage de données. En revanche, le type de connecteur encapsule des fonctionnalités d'interactions.

La configuration n'a pas de type mais plutôt un style architectural, dans ce qui suit nous détaillons la notion de style architectural.

2.6 Les styles architecturaux

Un style architectural permet d'expliciter et de capitaliser les aspects clés qui caractérisent une famille de configuration qui ont les mêmes propriétés structurelles et sémantiques (exemples : Pipe-Filter, Client-Serveur) [51]. Les styles architecturaux permettent de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment traité et analysé des architectures similaires [71].

"...Un style architectural peut être considéré comme une classe générique d'architectures. Il définit les propriétés pour les éléments de conception, les règles et les contraintes sur la façon de les lier et permet des analyses spécialisées et spécifiques au style..." [71].

La figure-2.4- illustre un exemple de style architectural client/serveur décrit à l'aide de l'ADL ACME[2], nous constatons que les types des composants et des connecteurs sont explicités ainsi que les attachements types entre les éléments architecturaux.

2. Dans ACME un style est toujours associé à une configuration référence.

```
Style client-server = {
      Component Template client(rpc-call-ports: Ports) = {
                        /* un gabarit composant client
                        /* est spécifié avec un port en entrée
         Ports rpc-call-ports;
         Properties {Aesop-style: style-id = client-server ;
                     Unicon-style : style-id = cs ;
                     source-code : external = "CODE-LIB/client.c"}
}
      Component Template server (rpc-receive-ports : Ports) = {
         Ports rpc-receive-ports ;
         Properties {Aesop-style : style-id = client-server ;
                      Unicon-style : style-id = cs; ...}}
      Template rpc (caller_port, callee_port: Port) defining (conn: Connector) =
                     /* un connecteur est spécifié avec un gabarit et deux
                     /* ports en entrée. La clause « defining »
                     /* signifie qu'un identifiant unique « conn » doit
                     /* être généré lorsque ce gabarit est utilisé
         {Conn = Connector {
                     Roles {caller, callee}
                     Properties { synchronous : boolean = true;
                                  max-roles : integer = 2; }
                                  protocol : Wright = "..." }}
      Attachments {conn.caller to caller port;
                     conn.callee to callee port; }}
   }
   System complex cs : client-server = {
                     /* description de la configuration d'un système
                     /* typé par le sytle client/serveur: c'est à
                     /* dire de l'assemblage statique des composants
                     /* au moyen des connecteurs.

     c1 = client (send-request);
     c2 = client (send-request);
     c3 = client (send-request);
     s1 = server (receive-request);
     s2 = server (receive-request);
   Rpc (c1.send-request, s1.receive-request);
   Rpc (c2.send-request, s1.receive-request);
   Rpc (c3.send-request, s2.receive-request); }
```

Figure 2.4 Description du style client-serveur à l'aide de L'ADL ACME.

2.6.1 Les objectifs des styles architecturaux

Un style architectural permet de :

- Déterminer l'ensemble du vocabulaire désignant les types des entités de base (composants et connecteurs) tels que pipe, filtre, client, serveur, événement, processus). La figure-2.4- illustre un exemple du vocabulaire utilisé dans les architectures clients-serveur.
- Spécifier l'ensemble des règles d'agencement qui déterminent les compositions et les assemblages permis entre les éléments architecturaux. Par exemple, une contrainte peut interdire un cycle dans le style client-serveur. Selon Leymonerie [71] trois types de contraintes peuvent être identifiées :
  - Les contraintes topologiques précisent les éléments de construction utilisables, leur nombre d'occurrences possibles au sein de l'architecture, ainsi que les règles de configuration contraignant leurs interactions.
  - Les contraintes comportementales concernent l'évolution temporelle de l'architecture et le comportement des éléments architecturaux. Concernant l'évolution temporelle de l'architecture,

un style peut spécifier comment l'architecture peut évoluer dynamiquement, quelle sorte d'élément peut apparaître ou disparaître et quelles contraintes les architectes doivent respecter en modifiant une architecture à la volée. Les styles peuvent aussi spécifier des propriétés de mobilité propres à des éléments architecturaux, et des propriétés comportementales (exemple : pas d'inter-blocage). Ces propriétés peuvent être celles du comportement d'une configuration entière ou seulement d'un élément architectural.

- Les contraintes d'attributs concernent les aspects non structurels et non fonctionnels d'une AL. Les attributs apportent des informations sur les éléments architecturaux. Ils peuvent être contraints sur leur type, leur nom et leur gamme de valeurs. Les styles peuvent spécifier comment les valeurs des attributs sont liées avec d'autres aspects architecturaux (la structure et le comportement).

– Donner une interprétation sémantique, en d'autres termes, définir la signification des compositions des éléments contraints par les règles de conception ; par exemple, un système multi-agents communiquant via le tableau noir signifie que c'est un partage de connaissances entre ces agents.
– Déterminer les analyses qui peuvent être réalisées sur un système construit selon un tel style. Par exemple, la vérification de la causalité dans un système distribué basé sur la communication par événement.

2.6.2 Les avantages des styles architecturaux

Nous résumons les avantages de l'utilisation des styles architecturaux comme suit :
– Capitaliser et exploiter les expériences précédentes des architectes systèmes.
– Augmenter la réutilisation des ALs.
– Améliorer la qualité des ALs en fournissant des règles et attributs qui régissent chaque famille de configurations et qui doivent être respectés pour assurer la cohérence du résultat.
– Simplifier la description des ALs en utilisant des templates prédéfinies.
– Offrir un support d'analyse pour l'évolution des ALs.

2.6.3 Les styles architecturaux et les procédés logiciels

Les styles architecturaux permettent de capturer et d'expliciter les spécificités des structures récurrentes. Dans le domaine des PLs cette tâche est remplie par les cycles de vie de logiciels [28] et les patrons de PLs [117]. Certains auteurs ont fait le rapprochement entres ces deux principes, en effet, dans [27] Boehm met en évidence la ressemblance entre cycle de vie de logiciel et style architectural, cependant, les travaux présentés ne sont pas assez détaillés et ne présentent pas de solutions concrètes. Cette dualité style architectural/ cycle de vie constitue le point de départ de nos travaux pour la définition de styles architecturaux pour les architectures de PLs.

2.7 Conclusion : synthèse et analyse

Dans ce chapitre nous avons défini les ALs et les avantages qui peuvent être mis au service des PLs, nous avons aussi présenté la notion d'ADL, et opté pour l'ADL ACME pour décrire les architectures de PLs.

Afin d'évaluer la faisabilité de la description d'architectures de PLs indépendamment des travaux existants, nous établissons des correspondances entre les éléments de base des PLs et les éléments de base des ALs.

Le tableau -2.3- résume ces correspondances, nous remarquons que :

Elément procédé	**Élément architectural correspondant**
Activité (Unité de travail)	Composant
Produit	Interface Composant
Transmission de données	Connecteur
–	Interface Connecteur
Structure du PL	Configuration
Cycle de vie de logiciel	Style architectural
Rôle	–

Table 2.3 Les correspondances adoptées entre les concepts de base des PLs et les concepts de base des ALs.

- Certaines correspondances telle que l'unité de travail qui correspond à un composant ou structure PL qui correspond à une configuration sont faites directement et sans ambiguïté.
- D'autres éléments tels que "Rôle"[3] ou "Interface Connecteur" ne trouvent pas de correspondance directe.

Ces premières correspondances même si elles sont intuitives nous démontrent qu'il est possible de décrire des architectures de PLs, elles nous servirons donc de base de réflexion pour définir les éléments de notre architecture de PLs.

L'étude des ALs et leur potentiel à contribuer à la réutilisation des PLs nous a conforté dans notre idée de décrire des d'architectures de PLs. Plusieurs travaux on été fait dans ce sens, dans le chapitre suivant nous présentons les approches de réutilisation des PLs à base d'ALs. Aussi, nous présentons un cadre de comparaison qui nous permettra d'évaluer l'efficacité des approches de réutilisation des PLs à base d'ALs existantes, cela nous permettra, également, de cerner leurs avantages et leurs inconvénients afin de proposer notre propre approche.

3. le rôle signifie la spécialité ou les qualifications que peut ou doit avoir un intervenant dans l'exécution du modèle de PL. A ne pas confondre avec Rôle connecteur.

CHAPITRE 3

Évaluation des approches de réutilisation de procédés logiciels à base d'architectures logicielles

3.1 Introduction

Dans les chapitres précédents, nous avons abordé les concepts de base relatifs au domaine des Procédés Logiciels(PLs) et au domaine des Architectures Logicielles(ALs). A la fin de ces chapitres, et dans le but de promouvoir la réutilisation des PLs, nous avons opté pour la description d'architectures de PLs. Aussi, pour améliorer la réutilisation des PLs, nous avons également opté pour l'exploitation d'ontologie de domaine comme espace de stockage des connaissances réutilisables.

Un point de départ évident consiste à étudier les approches de réutilisation de PLs à base d'ALs qui ont été proposées dans la littérature, par conséquent, nous commençons par présenter les différentes approches de réutilisation de PLs à base d'ALs étudiées.

Afin de comprendre leur raisonnement et de bénéficier de leur expérience dans le domaine de la réutilisation des PLs ; ces approches doivent être évaluées et analysées. C'est dans cette perspective que nous présentons un cadre de comparaison qui offrira un support pour l'analyse et l'évaluation des approches de réutilisation de PLs à base d'ALs.

Plusieurs cadres de comparaison ont été proposés dans le domaine des PLs [124] [2] [20] [8]. Cependant, l'introduction de nouveaux paradigmes dans la modélisation et l'exécution des PLs tels que les systèmes multi-agents [119], les ALs [32], la programmation orientée aspects [81] ...etc. mène souvent à l'introduction de nouveaux critères de comparaison et impose dans certains cas la définition de nouveaux cadres de comparaison spécifiques à ces nouveaux paradigmes.

Dans le cadre de notre travail, l'introduction des ALs comme paradigme de modélisation de PLs doit être évalué, d'où la nécessité de définir un cadre de comparaison adéquat.

Les cadres de comparaison existants introduisent certains critères d'évaluation tels que la réutilisation ou la modularité [8], seulement, ces critères ne permettent pas d'évaluer l'aspect architectural du PL de

manière efficace, par conséquent, la définition d'un nouveau cadre de comparaison est indispensable.

Dans ce chapitre et après la définition de notre cadre de comparaison pour l'évaluation des approches de réutilisation de PLs à base d'ALs, nous proposons une analyse des approches étudiées. Nous nous appuyons sur cette analyse pour établir une synthèse et un bilan sur l'état du domaine de la réutilisation des PLs à base d'ALs.

Comme mentionné dans le chapitre -2-, nous avons opté pour l'utilisation d'une ontologie de domaine comme espace de stockage des connaissances PLs. Nous étudions, donc, les solutions de modélisation de PLs exploitant des ontologies de domaine, notre objectif est de cerner les orientations, les avantages et les inconvénients de ces solutions, et repérer éventuellement les ontologies qui peuvent être exploitées par notre approche. L'étude des approches de réutilisation de PLs à base d'ontologies de domaine n'est pas aussi détaillée que celles des approches de réutilisation de PLs à base d'ALs,car notre objectif n'est pas de contribuer à l'amélioration des PLs à base d'ontologie, mais seulement de repérer une ontologie existante qui peut être réutilisée dans notre approche.

3.2 Les approches de réutilisation de Procédés Logiciels à base d'architectures logicielles

Plusieurs approches de réutilisation des PLs à base d'AL ont été proposées. Chaque approche apporte une solution particulière pour répondre à une préoccupation particulière (distribution, évolution, hétérogénéité, simulation, dynamicité...etc.). Néanmoins, ces approches ont toutes un point en commun ; elles exploitent des concepts architecturaux pour assoir leur solution.

Selon l'élément architectural sur lequel se focalise l'approche, les approches de réutilisation de PLs à base d'ALs sont classées en trois catégories, des approches orientées composants, des approches orientées connecteurs et les approches orientées configurations.

3.2.1 Les approches de réutilisation de procédés logiciels centrées composants

3.2.1.1 Endeavors [62]

Le système Endeavors est une infrastructure distribuée destinée à modéliser et exécuter les PLs. Il traite la communication, la coordination et le contrôle des PLs.

Le système est basé sur la combinaison d'une approche homogène au niveau de la modélisation et d'une approche hétérogène à l'exécution. Ainsi, un procédé peut être décrit à travers un ensemble de modèles (fragments de procédés) en utilisant le même formalisme orienté objet et les mêmes concepts (activité, produit et ressource). Chaque modèle est interprété par un ensemble de "handlers". L'implémentation en Java de ces "handlers" permet de les porter sur des plate-formes hétérogènes.

3.2.1.2 PYNODE [15]

La principale motivation de PYNODE, est de développer un environnement interactif et intégré, modélisant et exécutant des PLs à base de composants. L'idée consiste à éliminer les aspects monolithiques des formalismes des PLs et de fournir des possibilités de réutilisation et d'intégration de représentations de PLs hétérogènes. La structure abstraite du composant procédé est décrite à l'aide de langage orienté objet, cependant, l'implémentation décrit essentiellement un fragment de modèle de PL écrit dans un langage de modélisation de procédé. Chaque composant procédé se compose de deux parties : une partie interface et une partie implémentation.

Les ports de l'interface composant décrivent les relations d'entrée, de sortie et de synchronisation pour permettre aux composants d'interagir dynamiquement et d'échanger des informations (lecture/ écriture) de manière asynchrone durant l'exécution.

L'exécution de procédés est supportée via les vues d'exécution. Une vue d'exécution définit une collection de composants et de relations de composition, elle permet aussi la composition dynamique des composants procédés.

3.2.1.3 Support de réutilisation de PLs à base de gestion de configuration [19]

L'objectif de cette approche est d'offrir un support pour la modélisation et la maintenance des PLs distribués en exploitant la technique de gestion de configuration pour la définition et la composition de composant de PLs. Cette technique est combinée au PIL (Process Interconnexion Language), un formalisme qui permet de décrire la structure des composants PL. Le principe de base de cette approche est la séparation de la description de l'implémentation des éléments procédés, ainsi, les notions de base sont :

- Unité procédé (interface+ implémentation) : caractérisée par une interface et une implémentation qui décrivent un fragment de procédé.
- Family (interface + variants) : introduit pour décrire la variabilité d'une implémentation, elle fournit les opérations de versionnement de l'implémentation d'une unité de procédé.
- Configuration : est une unité composite contenant des versions qui permettent de construire un modèle de PL.

3.2.1.4 OPC (Open Process Component) [57] [58] [59]

Le Framework OPC 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 l'interopérabilité et la réutilisation de composants procédés hétérogènes construite selon trois niveaux d'abstraction.

- Le niveau métamodèle qui identifie les entités fondamentales du procédé et leurs relations.
- La machine d'états finis qui est la représentation du comportement des entités du procédé.
- Le framework en couches orienté objet qui permet de décrire la structure du procédé à base de composant, ce Framework (formé de trois couches) décrit aussi les différents formalismes de représentation du procédé et leur sémantique qui permettrons de décrire l'implémentation du PL.

3.2.1.5 APEL (Abstract Process Engine Language) [43]

APEL est un environnement centré procédé pour la modélisation et l'exécution de PLs. L'objectif d'APEL est d'offrir un support distribué indépendant des plateformes d'exécution (peut utiliser différents moteurs d'exécutions), et de couvrir un large éventail de processus. Il est basé sur une architecture dénommée "fédération de composants".

L'idée consiste à partager le modèle global d'un procédé entre plusieurs aspects (activités) indépendants ; chacun des aspects représentés par des modèles de procédés locaux seront réalisés par un composant spécifique. Tous les composants dans cet environnement sont des Process (Sensitive) Support Systems (PSS) hétérogènes au niveau représentation ; cependant, ils partagent les mêmes concepts de procédés fondamentaux (activité, produit, rôle,).

3.2.1.6 RHODES [37] [38]

Est un AGL-P centré procédé pour la modélisation et l'exécution des procédés logiciels à base de composants, l'objectif de cet AGL est de réutiliser les connaissances pour le contrôle et l'assistance au développement. RHODES s'inspire des travaux sur les composants logiciels et adopte leurs caractéristiques aux composants procédés.

Selon RHODES un composant procédé est un fragment de PL, deux types de composants sont définis (composant élémentaire et composant composite), chaque composant a une spécification et une implantation, les interactions entre composants se font par appels de fonction. Les procédés RHODES sont décrits en PBOOL+ qui est un langage de modélisation de procédé intégrant la notion de composant procédé.

3.2.2 Les approches de réutilisation de procédés logiciels centrées connecteurs

3.2.2.1 Approche d'assemblage de modèles à base de connecteurs[76]

Cette approche propose un nombre de connecteurs de modèles. Ces connecteurs sont définis pour traiter la discontinuité des informations entre modèles de données des différentes phases de développement. Concrètement, ces connecteurs sont intégrés à des activités de développement.

Ces connecteurs sont spécifiques, ils concernent certaines étapes de développement et traitent certains modèles de données seulement. Même si ces connecteurs dépendent des modèles sources et destinations, ils partagent un certain nombre de caractéristiques (direction, efficacité, automatisation) qui peuvent être le point de départ pour la définition de connecteurs PLs. Ainsi, dans cette approche trois types de connecteurs sont définis :

- Requirements-to-Architecture Model Connector : Le modèle de connecteurs CBSP (Component, Bus, System, Property) a été développé pour passer du modèle des besoins au modèle d'architecture. L'exécution de ce connecteur est hautement centré humain et dépend de la réflexion des développeurs du procédé.
- Architecture-to-Design Model Connectors : Les connecteurs de cette phase transforment le modèle d'architecture en modèle de conception en passant par plusieurs étapes intermédiaires.
- Inter-Design Model Connectors : L'ensemble d'activités et de techniques qui permettent de détecter les inconsistances entre modèles de conception de manière automatique. Ces connecteurs sont considérés comme un Framework d'intégration de modèles de conceptions.

3.2.2.2 Support basé ADL ZETA pour la modélisation de procédés [6]

Cette approche a pour objectif de fournir un support d'interaction pour les PLs intensifs qui sont décrits comme complexes, hétérogènes et distribués.

Un procédé intensif est vu comme un ensemble de "processlets" qui interagissent entre eux. Les "processlets" sont des sous procédés représentés en boites noires qui interagissent en exploitant des interfaces d'interactions.

Pour modéliser ces interactions, cette approche exploite l'ADL ZETA et décrit les différents modèles d'architectures centrés interaction pour les procédés intensifs. Cet ADL permet de décrire et contrôler comment et quand interagissent les "processlets" même pendant l'exécution, et cela sous la direction d'un gestionnaire de procédé. Les concepts architecturaux de base de cette approche sont :

- Interface interaction, qui est l'interface du "processlet" : elle fournit les capacités d'interactions fournies ou requises du "processlet". Elle décrit les activités, les produits, et les ports requis ou

fournis par le "processlet".
- Interaction : définit les règles et contraintes ordonnées qui doivent être respectées pour router le message d'un "processlet" à un autre.
- Port est le point de connexion de l'interface interaction ou de l'interaction.
- Attachement : le lien établi entre le port de l'interface interaction et le port de l'interaction.
- Message : les informations envoyé d'un "processlets" à un autre.

3.2.3 Les approches de réutilisation de procédés logiciels centrées configurations

3.2.3.1 Environnement pour la modélisation et la simulation d'architecture de procédé d'acquisition [32]

Cet environnement supporte la modélisation et la simulation des procédés d'acquisition en utilisant les ALs. Les procédés d'acquisitions sont décrits comme larges et complexes, requièrent de multiples entreprises intervenantes et de multiples procédés.

Il offre un support de distribution et d'exécutions concurrentes de plusieurs procédés simulés en utilisant HLA (Hight Level Architecture) combiné à l'infrastructure (RTI) Run Time Infrastructure. HLA est un standard IEEE proposé pour structurer la simulation des systèmes distribués et à exécution concurrente. Les composants sont des objets qui encapsulent des programmes d'application. Par contre les connecteurs sont des objets qui encapsulent les moyens d'interactions et qui peuvent être des protocoles, des applications, des middlewares...etc. Les Interfaces spécifient les ressources fournies ou requises par les composants ou les connecteurs.

3.2.3.2 Environnement de Modélisation de procédés évolutifs [42]

C'est un Framework qui permet de décrire des PLs évolutifs ; les PLs sont à base d'architectures, ainsi :

Les composants sont des composants évolutifs décrits avec les réseaux de Pétri, les sommets représentent les activités avec seulement une entrée et une sortie entre deux sommets (activités). Les interactions sont prises en charge par des connecteurs prédéfinis (concurrent, sélection et précédence) et décrites avec un arc entre les sommets.

Le composant est encapsulé et la structure globale du procédé est décrite à l'aide du langage EPCDL (Evolution Process Component Description language) développé dans ce but. Il permet l'adaptation et la composition semi assistées des composants PLs.

3.3 Cadre de comparaison pour les solutions de réutilisation de modèles de procédés logiciels à base d'architectures logicielles

Comme cité précédemment, il est indispensable de mettre en place un cadre de comparaison pour pouvoir analyser les solutions de réutilisation de PLs à base d'ALs. Les solutions de modélisation de PLs à base d'ALs ont pour particularité d'être à l'intersection de deux domaines de recherches : l'ingénierie des PLs et l'ingénierie des ALs.

Les architectures de PLs sont avant tout des modèles de PLs, de ce fait, ont les mêmes caractéristiques et objectifs que les modèles de PLs traditionnels. Par ailleurs, ils sont modélisés en exploitant les concepts des ALs (composant, connecteur, configuration...etc.) ; ils sont considérés comme des ALs, ils ont donc, les mêmes caractéristiques et objectifs que les ALs.

Notre cadre de comparaison doit prendre en compte au moins ces deux axes : Axe des PLs et axe des ALs. Ces deux axes définissent l'aspect technique des solutions de réutilisation de PLs à base d'ALs, ils se focalisent sur le côté quantitatif de l'architecture de PL (Figure- 3.1-).

L'aspect qualitatif doit aussi être évalué, ainsi, le troisième axe de notre cadre de comparaison concernera l'aspect qualité de l'architecture de PL. Nous définissons un seul axe pour l'évaluation de la qualité des architectures de PLs résultats des approches de réutilisation, car nous évaluons l'architecture de PL en tant qu'une seule entité qui a un certain nombre de critères de qualité. De plus, la plupart des critères qualitatifs des deux domaines (AL et PL) se rejoignent et se ressemblent.

Ainsi, nous définissons les critères d'évaluation selon les axes suivants :

- Les critères d'évaluation selon l'axe PL : ces critères permettent d'évaluer les concepts et langages de PLs utilisés pour décrire le PL indépendamment de la vision architecture.
- Les critères d'évaluation selon l'axe AL : contrairement aux caractéristiques précédentes, ces critères permettent d'évaluer les concepts et langages exploités pour décrire la structure abstraite de l'architecture de PL indépendamment de l'aspect procédé.
- Les critères d'évaluation qualitative : ces critères permettent d'évaluer la qualité des modèles de PLs à base d'ALs.

Nous détaillons ces critères dans les sections suivantes.

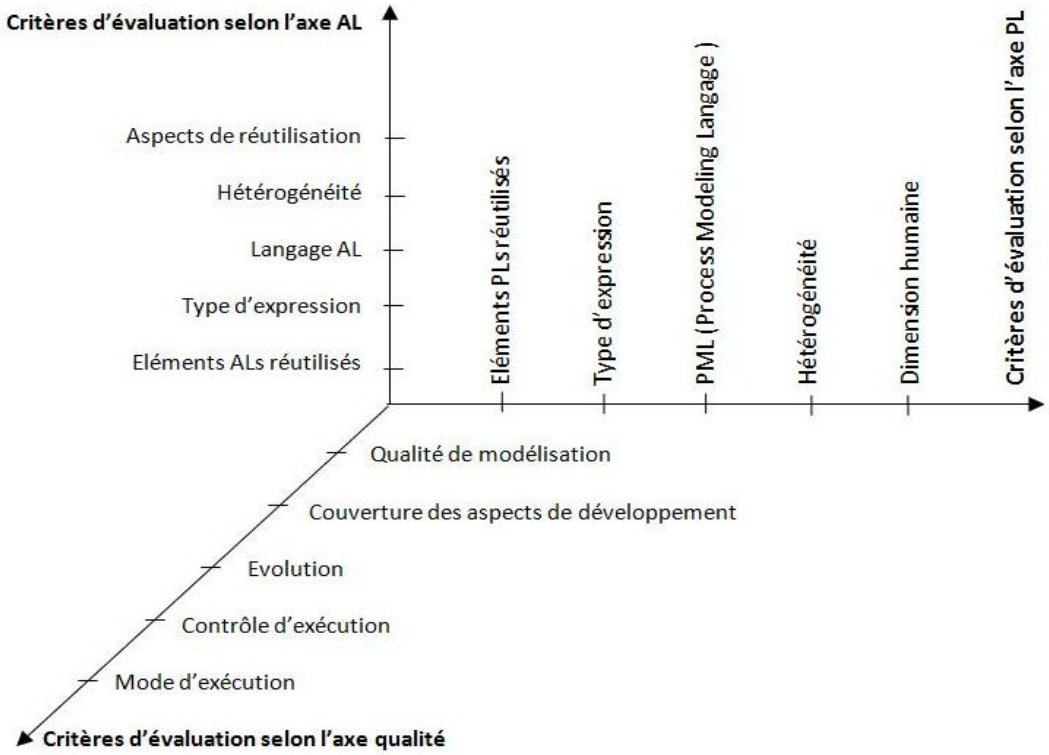


Figure 3.1 Les axes d'évaluation des approches de réutilisation de PLs à base d'ALs.

3.3.1 Les critères d'évaluation selon l'axe procédé logiciel

Ces critères définissent les caractéristiques des modèles de PLs indépendamment de la vision AL. Les PLs à base d'ALs sont en premier lieu des PLs, il est légitime de prendre en compte les classifications

et les cadres de comparaison déjà établis dans ce domaine.

Éléments PLs réutilisés - De base (activité, Rôle et Produit) - Secondaire (ressource, acteur,...)
Types d'expression - Non formel - Semi formel - Formel
PMLs (Process Modeling Languages) - Orienté objet - Réseau de Pétri - A base de règles - Fonctionnel
Hétérogénéité - Syntaxique - Sémantique - Plateforme d'exécution - Homogène
Dimension humaine - Assistance à l'exécution - Contrôle de l'interaction humaine

Table 3.1Les critères d'évaluation selon l'axe PL.

Le tableau -3.1- résume les critères sélectionnés pour évaluer l'aspect technique des solutions de réutilisation selon l'axe PL. Ces critères sont inspirés des classifications de Longchamps [73], Accuna et al [2] et Conradi et al [48], zamli [124] [123], ainsi nous évaluons :

3.3.1.1 Les éléments PLs réutilisés

Comme cité précédemment le noyau conceptuel du PL est constitué des concepts Activité (Unité de travail), Produit de travail et rôle. D'autres éléments dits secondaires peuvent être utilisés pour décrire d'autres facettes du PL. Ainsi, nous regroupons les éléments PLs réutilisés en deux groupes :

- **De base :** Les éléments "activité" "rôle" "produit" sont considérés comme des concepts de base dans les PLs. Ces concepts restent les éléments de base pour les PLs à base d'ALs.
- **Secondaires :** Ces éléments décrivent les préoccupations et les objectifs spécifiques à certains modèles de PLs, les PLs peuvent être dédiés à la gestion de configuration, la gestion du temps ou des ressources de développement [49]. Des concepts additionnels peuvent être intégrés pour prendre en compte ces préoccupations, les concepts "Ressources", "outils", "stratégie" ou "guidance" sont quelques exemples de concepts PLs secondaires.

Par la définition de ce critère, notre but n'est pas d'identifier les concepts exacts manipulés par les approches étudiées mais de vérifier si la modélisation à base d'ALs requière des concepts additionnels ou spécifiques pour prendre en charge l'aspect architecture.

3.3.1.2 Le type d'expression

Ce critère permet de spécifier le degré d'abstraction des formalismes utilisés lors de la modélisation des PLs à base d'ALs. Comme pour les PLs, selon l'objectif et la complexité du modèle de PL, nous distinguons plusieurs types d'expression. Dans notre cadre de comparaison, nous classons les modèles de PLs selon trois types de formalismes.

- **Non formel :** Les modèles de PLs sont décrits sans utiliser un formalisme strict mais plutôt des formalismes à base de langage naturel. L'objectif est la compréhension du PL.
- **Semi formel :** Les modèles de PLs sont décrits graphiquement, l'objectif est la structuration du PL.
- **Formel :** Les langages utilisés sont des langages qui ont une syntaxe et une sémantique rigoureuses, l'objectif est de permettre l'exploitation effective par exécution du modèle de PL.

3.3.1.3 Le PML (Process Modeling Language)

Un PML (Process Modeling Language) est un langage qui permet de modéliser les PLs, un nombre important de PMLs ont été définis, exploitant différents paradigmes [124]. Pour la comparaison des modèles de PLs à base de concepts architecturaux, nous utilisons les différentes catégories de formalismes définis dans la classification d'Ambriola et al [48]. Ainsi, nous utilisons les catégories suivantes :

- **PMLs Orientés Objet :** où le PML se base sur les concepts et les mécanismes orientés objets.
- **Réseau de Pétri :** où le PML utilise les réseaux de pétri [124].
- **A base de règles :** où le PML utilise des techniques de panification en se basant sur des langages à base de règles.
- **Langage de programmation :** où le PML définit le modèle de PL comme un programme en exploitant les langages de programmation connus.

3.3.1.4 L'hétérogénéité

L'hétérogénéité peut être évaluée dans les modèles de PLs principalement distribués. En effet, les différents fragments du modèle de PL peuvent être hétérogènes selon trois niveaux :

- **Syntaxique :** les fragments de modèles de PLs sont modélisés en exploitant différents PMLs.
- **Sémantique :** les fragments de modèles de PLs ne respectent pas le même métamodèle.
- **Plateforme d'exécution :** les fragments de PLs peuvent s'exécuter sur des plateformes d'exécution différentes.
- **Homogène (pas d'hétérogénéité) :** les fragments de modèles de PLs respectent la même syntaxe, même sémantique et s'exécutent sur la même plate-forme d'exécution.

3.3.1.5 La dimension humaine

Les PLs sont orientés humain ; les tâches de réflexion, d'analyse et de création sont des tâches de développement typiquement humaine. L'influence des pratiques sociales et culturelles qui sont humaines et non techniques sur la modélisation et l'exécution du PL a été déjà mise en évidence [110]. La prise en compte de cette dimension lors de la modélisation des architectures de PLs est primordiale. Pour cela, nous définissons deux critères :

- **Assistance à l'exécution :** l'approche doit permettre l'identification des tâches typiquement humaines et offrir l'assistance adéquate au réalisateur de cette tâche lors de l'exécution du modèle de PL.

– **Contrôle de l'interaction humaine :** l'interaction humaine peut déterminer la réussite du modèle de PL. Ce critère permet d'évaluer l'importance donnée au contrôle des interactions humaines.

3.3.2 Les critères d'évaluation selon l'axe architecture logicielle

Dans leur Framework, Ambrilola et al [48] ont consacré une catégorie à la "réutilisation" pour la description des modèles de PLs ; ils considèrent que la réutilisabilité est un critère important qui fait partie des critères de qualité d'un PL.

Dans notre travail, nous traitons des PLs à base d'ALs, ces modèles de PLs exploitent des concepts réutilisables tels que le composant, le connecteur ou la configuration. La réutilisabilité est l'essence même des ALs, et par transition l'essence des PLs à base d'ALs. De ce fait, la présence de la réutilisabilité, dans les modèles de PLs à base d'ALs n'est pas remise en cause mais plutôt détaillée. Ainsi, dans cette section, nous identifions les critères qui permettent d'évaluer le type et le degré de réutilisabilité des modèles de PLs à base d'ALs.

De la même manière que pour l'identification des critères d'évaluation de l'axe PL, pour identifier les critères dévaluation selon l'axe AL, nous nous inspirons des Frameworks [1] [78] [16] [80] mis en place dans le domaine des ALs.

Éléments ALs réutilisés - Composant - Connecteur - Configuration - Style
Type d'expression - Implicite - Prédéfini - Explicite
Langages d'AL - Langage orienté objet - ADL spécifique aux PLs - Autre ADL
Hétérogénéité - Homogène - Hétérogène (composant, connecteur, configuration)
Aspect de réutilisation - Mécanismes de réutilisation - Espace de stockage adopté - Portée de la réutilisation : Interne ou externe au système - Déploiement et génération de code

Table 3.2Les critères d'évaluation selon l'axe AL.

Le tableau -3.2 résume les critères adoptées pour l'évaluation des approches de réutilisation de PLs selon l'axe AL.

3.3.2.1 Éléments ALs réutilisés

Comme définie précédemment, une AL décrit le système comme un ensemble de composants (unités de calcul ou de stockage) qui communiquent entre eux par l'intermédiaire de connecteurs (unités d'interaction). La plupart des travaux existants considèrent les concepts "composant", "connecteur" et "configuration" comme concepts de base de toute AL. Ce constat, reste valable pour les modèles de PLs à base d'ALs.

L'importance de ces concepts varie d'une approche de réutilisation de PLs à une autre, cette variation dépend de l'interprétation que peut donner l'auteur aux entités réutilisables.

Dans notre travail, nous évaluons l'utilisation et l'interprétation des éléments architecturaux de base à savoir : composant, connecteur et configuration. Néanmoins, nous introduisons le concept "style" car nous pensons que la notion de style est très importante pour la réutilisation des PLs, notre objectif est d'évaluer l'importance donnée à ce concept dans les approches de réutilisation de PLs à base d'ALs.

Même si les concepts architecturaux évalués sont spécifiques aux PLs, ils gardent les mêmes définitions adoptées par la communauté des ALs, ainsi :

- **Composant :** est une entité qui fournit des fonctionnalités de calcul et de stockage spécifiques aux PLs, le composant interagit avec les autres composants pour réaliser un ou plusieurs objectifs de l'architecture de PL.
- **Connecteur :** est un bloc de construction architectural utilisé pour modéliser les interactions entre les composants PLs et pour spécifier les règles qui régissent ces interactions.
- **Configuration :** représente un graphe de composants et de connecteurs et définit la façon dont ils sont reliés entre eux.
- **Style :** permet de capturer les caractéristiques des structures et des comportements récurrents des PLs.

3.3.2.2 Le type d'expression

Le type d'expression des éléments réutilisés peut varier de :

- **Implicite :** exploité mais de manière non formelle sans utiliser de langage de modélisation, ou bien sans avoir d'existence propre.
- **Prédéfini :** Les instances des éléments prédéfinis sont de structures et de fonctionnalités connues, stockées et réutilisées.
- **Explicite :** Les instances des éléments ont une syntaxe et une sémantique rigoureuses, des instances peuvent être modifiées, et d'autres instances peuvent être rajoutées.

3.3.2.3 Les langages d'ALs

Les ADLs (Architecture Description Langages) ou langages de description d'architecture sont des formalismes spécifiques aux ALs, ils sont utilisés pour décrire la structure d'un système comme un assemblage d'éléments logiciels. Pour la modélisation des concepts architecturaux des modèles de PLs, nous désirons identifier les types de langages utilisés. Pour cela, nous utilisons trois catégories :

- **Les Langages orientées objets :** certaines architectures sont décrites à l'aide de langages orientés objets et exploitent les mécanismes objet tels que l'instanciation, l'agrégation et l'héritage.
- **ADLs spécifiques aux PLs :** sont des ADLs mais spécifiques à la description des architectures de PLs.
- **ADLs standards (Architectural Description Languages) :** sont les langages destinés à décrire des ALs mais qui sont non destinés au domaine des PLs.

3.3.2.4 L'hétérogénéité

Les modèles de PLs à base d'AL peuvent être :
- **Hétérogènes :** les composants ou les connecteurs de l'architecture de PL peuvent être hétérogènes syntaxiquement ou sémantiquement.
- **Homogènes :** le modèle de PL à base d'AL exploite les mêmes concepts, et le même langage lors de la modélisation.

3.3.2.5 Les aspects de réutilisation

Ce critère permet d'évaluer certains aspects que nous jugeons importants pour faciliter la réutilisation des PLs, ainsi, nous évaluons :
- **Les mécanismes utilisés pour la réutilisation effective** tels que la composition et certains mécanismes orientés objets tels que l'héritage, l'instanciation...etc.
- **L'espace de stockage et les stratégies de stockages adoptées :** les espaces de stockage peuvent être des bases de données, des fichiers ou des ontologies...etc.
- **Portée de la réutilisation [97] :** nous identifions deux niveaux : la portée de réutilisation interne où les blocs sont réutilisés par le système qui les a créés seulement, et la portée de réutilisation externe où les blocs (composants, connecteurs) peuvent être exportés et réutilisés par d'autres systèmes [97].
- **Le déploiement et la génération du modèle de PL :** ce critère permet d'évaluer les mécanismes mis en place pour permettre la génération du modèle de PL final.

3.3.3 Les critères d'évaluation selon l'axe qualité

Les solutions de réutilisation doivent permettre la modélisation des PLs à base d'ALs qui ont une certaine qualité.

Afin d'évaluer la qualité des PLs à base d'ALs modélisés, nous identifions un certain nombre de critères de qualité. La plupart des critères de qualité sont communs aux modèles de PLs et aux ALs. En effet, la facilité de modélisation, la compréhension, ainsi que l'évolution sont des caractéristiques qui peuvent être attribuées aux PLs comme aux ALs. Les modes d'exécution possibles pour les architecturées de PLs sont aussi recensés et s'inspirent des modes d'exécution des deux domaines. Néanmoins, nous rajoutons, le critère "contrôle d'exécution" comme critère d'évaluation et qui est spécifique aux PLs. Le tableau-3.3- résume les critères de qualité adoptés afin d'évaluer les PLs à base d'ALs.

3.3.3.1 Qualité de modélisation

Ce critère permet de spécifier la qualité de la modélisation, cette qualité peut être évaluée par :
- **La facilité de modélisation :** la modélisation doit être facile et indépendante du langage de modélisation de PL utilisé ; des outils d'assistance et de guidance doivent être intégrés pour faciliter la modélisation du PL.
- **La compréhension :** le résultat de modélisation doit être facilement compréhensible.
- **La cohérence et la persistance du résultat :** la solution de réutilisation doit permettre la vérification du résultat de modélisation, cette option est indispensable pour tous les PLs et particulièrement pour les architectures de PLs, car la modélisation de PL à base d'ALs prend en compte deux aspects : l'aspect structure et l'aspect contenu.

Qualités de modélisation - Facilité de modélisation - Compréhension - Cohérence et persistance
Mode d'exécution - Simulation - Distribué - Hétérogène - Dynamique - Incrémental - Itératif - Standard
Contrôle de l'exécution
Évolution - Statique - Dynamique

Table 3.3Les critères d'évaluation selon l'axe qualité.

3.3.3.2 Mode d'exécution

Ce critère permet de spécifier les modes d'exécution possibles pour une architecture de PL.

Nous constatons que la plupart des modes d'exécution des deux domaines (AL et PL) sont similaires. Aussi, afin de prendre en compte les modes d'exécution des processus agiles, nous introduisons les modes d'exécution itératifs et incrémentaux qui sont considérés comme des types particuliers de l'exécution dynamique. Les modes d'exécution identifiés se résument comme suit :

- **Simulation :** la simulation de l'exécution d'un modèle de PL est une exécution virtuelle du modèle de PL. L'objectif de la simulation est d'étudier le comportement de certains types de modèles de PLs dont l'exécution réelle est confrontée à des difficultés tels que leur complexité, faisabilité, coût ou le temps d'exécution.
- **Distribué :** l'exécution se fait sur des plateformes ou des environnements de travail distribués, l'intérêt et de permettre le travail d'équipe de développeurs sur des sites géographiquement distants.
- **Hétérogène :** l'exécution hétérogène est souvent associée à l'exécution distribuée, cela signifie que les modèles de PLs s'exécutent sur des plateformes ou des moteurs d'exécution hétérogènes.
- **Dynamique :** les caractéristiques de cette exécution est qu'il est possible d'effectuer Les modifications sans interrompre l'exécution du modèle de PL.
- **Incrémental :** est considéré comme un type particulier d'exécution dynamique, le PL est modélisé puis exécuté partie par partie (incrément), la partie initiale du modèle de PL concerne les activités à suivre pour implémenter les fonctionnalités noyau. Des couches de modèle de PLs permettant d'implémenter d'autres fonctionnalités sont rajoutées. L'objectif de ce type d'exécution est d'avoir un résultat opérationnel le plutôt possible dans le cycle de développement.
- **Itératif :** est considéré aussi comme un type particulier d'exécution dynamique où certains enchainements du modèles de PLs sont repris et ré-exécutés.
- **Standard :** l'exécution est lancée une seule fois, elle est standard et n'appartient à aucune des catégories précédentes.

3.3.3.3 Contrôle de l'exécution

Le contrôle d'exécution est un aspect important dans le modèle de PL. La solution proposée doit permettre le contrôle d'exécution du modèle de PL et gérer les adaptations effectuées.

3.3.3.4 Évolution

Comme cité dans les chapitres précédents, les PLs comme les ALs sont par nature évolutifs. Ainsi, ce critère permet d'évaluer les mécanismes mis en place pour promouvoir les différents types d'évolution du PL.

En général, deux types d'évolution sont possibles :

- **Statique :** où l'évolution se fait lorsque le modèle de PL ne s'exécute pas.
- **Dynamique :** où l'évolution se fait lors de l'exécution du modèle de PL.

3.4 Évaluation des approches de modélisation et d'exécution de PLs à base d'ALs

3.4.1 Résumé des approches de réutilisation de procédés logiciels à base d'architectures logicielles

Les approches étudiées (figure-3.2 -) sont des approches de modélisation et d'exécution de PLs qui exploitent les éléments architecturaux pour promouvoir la réutilisation des PLs. En faisant une première comparaison, nous remarquons que ces approches ne sont pas uniformes ; chacune manipule un certain nombre de concepts dans le but de traiter des problèmes spécifiques. Le tableau -3.4- résume ces approches et avance leurs objectifs et leurs points forts.

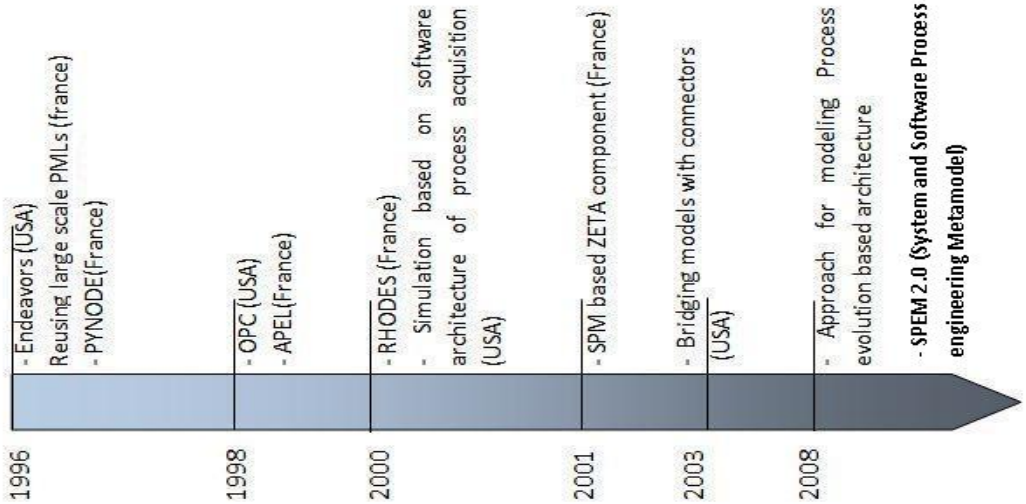


Figure 3.2 Approches de réutilisation de PLs à base d'ALs étudiées.

D'après le tableau -3.4- les approches n'offrent pas de solution générique ; les solutions proposées traitent des problèmes particuliers et proposent des solutions particulières ce qui limite la réutilisation des PLs. Néanmoins, nous pouvons faire une première classification des ces approches. En effet, nous pouvons constaté que les approches les plus anciennes sont influencées par la communauté industrielle, ainsi,

les architectures de PLs sont décrites à l'aide de langages orientés objets (OO). Par contre, les approches les plus récentes sont plutôt influencées par la communauté académique, de ce fait, les architectures de PLs sont décrites en exploitant les langages de description d'architectures (ADLs).

Approche	**Objectifs**	**Points forts**	**For. Mod.**
Endeavors	Traite les procédés WorkFlow, spécifique au travail distribué et dynamique	- Distribution via le web. - Indépendant des plateformes d'exécution. - représentation graphique du modèle de PL.	OO
PYNODE	Réutilisation dynamique de composants hétérogènes	-Exécution dynamique. -Hétérogénéité des fragments PLs.	OO
PLs à base de gestion de configuration	Réutilisation des PLs par versionnement des composants.	- Réutilisation par versionnement. - Contrôle de la cohérence du résultat.	OO
OPC	Réutilisation dynamique de composants hétérogènes	-Exécution dynamique. -Hétérogénéité des fragments PLs.	OO
APEL	Exécution distribuée de procédé hétérogène et indépendamment des moteurs d'exécution.	- Exécution distribuée - Indépendant des moteurs d'exécution. - Modèles de PLs hétérogènes.	OO
RHODES	Réutilisation des connaissances de développement, assistance à la modélisation de PLs.	- Assistance à la modélisation. - Vérification de la cohérence du modèle.	OO
App. d'assemblage de modèles basée connecteurs	Définition de différents connecteurs de modèles de données des différentes phases de développement.	- Identification de règles, pour le passage d'un modèle de données à un autre. - Identification de propriétés communes entre les connecteurs PLs.	ADL
ADL ZETA pour la modélisation des interactions PLs	Prise en charge des différentes interactions des modèles de PLs.	Définition de modèles d'interactions et de modèles d'interactions types.	ADL
Simulation d'architectures de procédés d'acquisition	Simulation, exécution concurrente et distribuée des modèles PLs d'acquisition.	Simulation de l'exécution de modèles de PLs d'acquisition (procédés complexes).	ADL
Modélisation de PL évolutifs	Réutilisation de composants évolutifs.	Définition d'un langage pour la description de les architectures de PLs.	ADL
SPEM	Réutilisation de composants PLs.	Définition de métamodèle pour la réutilisation des PLs à base de composants PLs.	OO
Abréviations : For. Mod. : Formalisme de Modélisation., **App. :** Application			

Table 3.4 Résumé des approches de réutilisation de PLs à base d'ALs étudiées.

3.4.2 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe PL

Le tableau-3.5- résume les caractéristiques des approches de réutilisation de PLs étudiées selon l'axe PL.

En analysant les informations collectées à partir des solutions de réutilisation étudiées, nous dressons les remarques suivantes :

- Les concepts des modèles de PLs modélisés sont généralement de base et se limitent pour la plupart aux concepts "activité" et "produit", nous remarquons aussi que les approches étudiées n'ont pas exploité de concepts additionnels afin d'introduire les concepts architecturaux, exceptée l'approche de Belkhatir et al [19] qui exploitent le "versionnement" dans la réutilisation de ses composants.
- La plupart des approches offrent des solutions qui modélisent des PLs formels, l'objectif est souvent de fournir des modèles de PLs exécutables. Ces modélisations formelles sont souvent combinées à des modélisations semi-formelles qui se traduisent par des représentations graphiques du modèle de PL. Les représentations graphiques ont pour objectif de fournir une aide aux développeurs et aux utilisateurs des modèles de PLs.
- En analysant les langages de modélisation de PLs exploités, nous remarquons qu'il n'y a pas de tendances particulières et aucun langage n'est mentionné comme langage facilitant la réutilisation des modèles de PLs, excepté le langage PBOOL+ de l'environnement RHODES [38] qui intègre la notion de composant procédé et de réutilisation de PLs.
  Il est à noter aussi que dans certaines solutions, le langage de modélisation de PLs n'a pas d'influence et les composants sont des boites noires comme dans l'approche de Alloui et Oquendo [6] et l'approche de Medvidovic et al [76], et dans certains cas le langage de modélisation de PL n'est même pas mentionné comme dans l'approche de Choi et Scatcchi [32].
- Concernant l'interaction humaine, aucune assistance particulière n'est fournie, l'interaction humaine se fait de manière implicite et n'est pas contrôlée par aucun mécanisme spécifique ce qui peut être considéré comme un inconvénient particulièrement pour les modèles de PLs à forte interaction humaine. Excepté l'environnement RHODES qui fournit une assistance sous forme de "sketchs" et de guidances textuelles [39].
- En analysant l'approche SPEM, nous remarquons que sa particularité d'être un métamodèle standard lui confère la possibilité de couvrir la modélisation d'un large éventail de PLs, ainsi, les concepts additionnels, les types d'expressions ainsi que la valeur des autres critères dépendront de l'environnement qui respecte le métamodèle SPEM, ce qui lui confère l'aptitude à modéliser tout type de PLs.

Approche	Éléments PLs réutilisés		T.E.	PML	Hétérogénéité	Dimension humaine	
	De base	Additionnel				Assistance	C.I.
Endeavors	Activité, Produit	Ressource	Formel, Semi formel	ObjV, Lisp (OO)	Plate-forme d'exécution	Représentation graphique	-
PYNODE	Activité, Produit, Rôle	-	Formel, Semi formel	quelconque	Syntaxique	Représentation graphique	-
PLs à base de ges. de conf.	Activité, Produit	version	Formel, Semi formel	dépend du modèle	Hétérogène	Vérification de la cohérence	-
OPC	Activité, Produit, Rôle	-	Formel, Semi Formel	quelconque	Syntaxique	Représentation graphique	-
APEL	Activité, Produit	Agent	Formel, Semi Formel	Dépend du modèle	Syntaxique Sémantique	Représentation graphique	-
RHODES	Activité, Produit, Rôle	Stratégie	Formel , Semi formel	PBOOL+ (OO)	Homogène	représentation graphique, Détection des incohérences	-
Simulation des Procédés d'acquisition	Dépend du modèle					Représentation graphique	-
Assemblage de modèles basée connecteurs	Activité, Produit	Phase	Non formel	-	-	-	-
Interactions PLs basées ADL ZETA	Activité Produit	-	formel	Dépend du modèle	Syntaxique Sémantique	Représentation graphique	-
Modélisation de procédés évolutifs	Activité, Produit	-	Semi Formel	réseau de pétri	homogène	représentation graphique	-
SPEM	Activité, Produit, Rôle	Dépend de l'environnement respectant SPEM					

Abréviations : T.E. : Type d'Expression, **ges. de conf. :** gestion de configuration, **C.I. :** Contrôle d'Interactions.

Table 3.5 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe PL.

3.4.3 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe AL

En analysant les caractéristiques des solutions de réutilisation étudiées selon d'axe AL, nous constatons que :

- La brique de base pour la réutilisation des PLs est le composant. La plupart des approches étudiées définissent le composant PL comme une unité de travail ou un enchainement d'unité de travail. Ainsi, le composant PL peut être un fragment de PL ou un modèle de PL. Le composant PL est dans la plupart des cas explicite, exceptée l'approche de Medvidovic et al [76] qui se focalise sur les connecteurs et n'a pas explicité ses composants. Aussi, même si les détails concernant les composants des approches de Alloui et Oquendo [6] et de Choi et Scatcchi [32] ne sont pas fournis, ils sont explicites.
- Concernant le concept connecteur, nous remarquons qu'il est implicite dans les approches les moins récentes ; il est considéré comme une transmission de données ou un flux d'exécution, ces transmissions n'ont pas eu beaucoup d'attention, leur prise en charge est intégrée dans le PL et dépend du langage et de la nature du PL. Par contre, les solutions les plus récentes donnent plus d'importance aux connecteurs qui sont soit prédéfinis ou explicites.
  Cependant, il faut noter que les définitions adoptées concernant ces connecteurs ne peuvent être généralisées, en effet, dépendant du langage de modélisation du PL et de l'interprétation de l'auteur, ces connecteurs restent spécifiques et la portée de réutilisation reste limitée à l'environnement d'origine.
- Aussi, dans la plupart des solutions, la notion de configuration est implicite. Dans les anciennes approches, la configuration implicite se traduit par l'exploitation de la représentation graphique du modèle de PL mais sans traitement ou analyse particulière au niveau structure. La configuration est devenue explicite avec l'utilisation d'ADLs qui permettent de décrire ses éléments et de spécifier les contraintes d'assemblage qui la régissent.
- La notion de style architectural n'est pas exploitée dans les approches étudiées, exceptée l'approche de Alloui et Oquendo [6] qui définie des connecteurs types (Interaction Type) et des configurations types (Container type) qui sont des éléments de base d'un style architectural. Néanmoins, selon cette approche, le composant n'a pas de type car il est considéré comme une boite noire. La description de ces éléments types est possible grâce à l'utilisation de L'ADL ZETA qui est un outil de l'ingénierie des ALs.
- En analysant la colonne des langages d'ALs, nous remarquons que ces langages sont généralement orientés objet ; la réutilisation est réalisée en exploitant les opérations offertes par le paradigme objet (instanciation, héritage, héritage multiple, composition...etc.). L'utilisation d'ADLs n'a fait son apparition que dans les approches les plus récentes, ce qui a permis à ces approches d'expliciter la totalité des éléments architecturaux.
- Bien que la plupart des solutions offrent des dépôts de composants (base de composants, librairies...etc.) pour permettre la réutilisation des composants PLs, le détail de l'organisation ou de la classification des composants PLs n'est pas présenté. Ce que nous retenons c'est qu'aucune approche n'a exploré la possibilité d'exploiter une ontologie de domaine. Ce choix est justifié par l'orientation adoptée par les solutions proposées, en effet, pour la réutilisation des PLs, les approches étudiées exploitent les concepts architecturaux, l'exploitation d'une ontologie de domaine exige une autre vision et d'autres mécanismes de réutilisation (connaissances procédé, inférence, raisonnement...), ce qui n'est pas l'objectif de ces solutions.
- Lors de la modélisation du PL à base d'AL, les approches étudiées implémentent des ALs homogènes, en d'autres termes, elles utilisent le même langage pour tous les composants et les connec-

Approche	**Éléments ALs réutilisés**				**Langage d'AL**	**Aspects de réutilisation**	
	Com.	Conn.	Conf.	Style		Mécanismes	Espace de stockage
Endeavors	Exp.	Imp.	Imp.	-	Orienté objet	Instanciation, Composition, Héritage Multiple	fichiers ascii
PYNODE	Exp.	Imp.	Imp.	-	Orienté objet	Instanciation, Composition dynamique, Héritage	BD de composants
PLs à base de gestion de conf.	Exp.	Imp.	Imp.	-	PIL (orienté objet)	Instanciation, Composition, Héritage, Versionnement	DBMS base de donnée de composants
OPC	Exp.	Imp.	Imp.	-	Orienté objet	Instanciation, Composition, Dynamique, Héritage	Base de composants
APEL	Exp.	Imp.	Imp.	-	Orienté objet	-	Pas de stockage de composants
RHODES	Exp.	Imp.	Imp.	-	PBOOL+ (orienté objet)	Instanciation, Composition, Héritage modulaire, sous typage	Base de composants gérée par le SGBD OO jasmin
Simulation de procédés d'acquisition	Exp.	Exp.	Exp.	-	HLA (ADL)	Composition	bibliothèque de modèles d'acquisitions
Assemblage de modèles basé connecteurs	Imp.	pré.	Imp.	-	-	-	-
Interactions PLs basées ADL ZETA	Exp.	Exp.	Exp.	I.T.	ZETA (ADL)	Instantiation	pas de stockage
Modélisation de procédés évolutifs	Exp.	pré.	Exp.	-	EPCDL (ADL pour PL)	Composition	base de composants
SPEM	Exp.	Imp.	Imp.	-	Orienté objet	Composition, instanciation	-
Autres caractéristiques : Toutes les approches sont **homogènes**, ont une portée de réutilisation **interne** et elles n'implémentent **pas de mécanismes de déploiement d'ALs.**							
Abréviations : Com. : Composant, **Conn. :** Connecteur, **Conf. :** Configuration, **Imp. :**Implicite, **Exp. :** Explicite, **Pré. :** Prédéfini,**I.T. :** Interaction Type, **PIL :** Process Interconnection Language **HLA :** Hight Level Architecture, **EPCDL :** Evolution Process Component Description Language.							

Table 3.6 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe AL.

teurs, exceptée l'approche de Medvidovic et al [76] où les connecteurs peuvent être décris en langage naturel ou en langage de programmation.
- La portée de la réutilisation de toutes les approches étudiées est limitée à la réutilisation interne, les solutions proposées traitent les préoccupations internes, et ne se focalisent pas sur la généricité et la portabilité des éléments architecturaux.
- Les approches étudiées ne proposent pas le déploiement d'ALs, en effet, la plupart des solutions sont centrées sur le concept composant ; le concept configuration est implicite ce qui ne permet pas de proposer un déploiement d'architectures.

3.4.3.1 Autres interprétations des termes des concepts architecturaux

Certaines approches utilisent les termes architecture/ connecteur/ configuration/ familly (qui est utilisé pour designer un style dans l'environnement ACME studio) mais pas pour décrire des éléments architecturaux ; leurs interprétations dépendent des solutions proposées, ainsi pour les concepts :
- **Composant :** Dans APEL [43] le composant est un composant produit et non un composant procédé, il représente une partie de l'environnement de modélisation et d'exécution du PL, l'objectif et d'avoir des modélisations et des exécutions distribuées de plusieurs modèles de PLs locaux.
- **Connecteur :**
  APEL [43] utilisent le terme connecteur pour décrire les liens (Data flow et Control flow) entre les activités PLs alors que le PL n'est pas décrit en orienté composant.
- **Architecture :** Borsoi et al [30] utilisent le terme architecture pour décrire la structure générale du PL non en terme de composants mais en termes de "phase" et "d'activité" qui seront raffinées au cours d'un processus de modélisation défini dans sa solution de modélisation du PL.
- **Familly :** Belkhatir et al [19] définissent le concept familly (qui peut laisser penser aux styles architecturaux) pour décrire un process unit (un composant) et ses versions.

SPEM [88] utilise le terme "configuration" pour décrire l'ensemble des éléments - qui ne sont pas forcément des éléments architecturaux- permettant de décrire un PL -qui n'est pas forcément une configuration- .

3.4.4 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe qualité

Le tableau -3.7- résume les caractéristiques qualitatives des approches étudiées. En analysant ces informations, nous constatons que :
- Les solutions proposées se focalisent sur la modélisation de PLs exécutables. De plus, l'utilisation de la modélisation à base d'ALs a permis de mettre en place des modèles de PLs distribués, hétérogènes, et d'offrir des possibilités d'exécutions dynamiques. Les caractéristiques de distribution, d'hétérogénéité et d'évolution sont considérées comme indispensables aux PLs de qualité. Ces caractéristiques ont pu être fournis en exploitant les concepts ALs.
- Les solutions étudiées n'offrent pas d'exécution incrémentale ou itérative, la nouveauté des méthodes agiles (qui sont incrémentales et itératives) est une justification de la non prise en charge de ces caractéristiques.
- Le contrôle d'exécution est sous la responsabilité du gestionnaire de PL. A part les représentations graphiques utilisées lors du contrôle d'exécution, aucun mécanisme spécifique à l'exécution centrée humain n'est explicité, ce qui augmente la dépendance de la qualité d'exécution des capacités humaines.

- Les ALs contribuent à améliorer la compréhension de modèles de PLs et à augmenter la facilité de modélisation. En effet, la formalisation puis l'utilisation de la structure abstraite du modèle de PL, en plus de la séparation entre interaction et traitement améliorent la lisibilité, et par conséquent, la compréhension et la facilité de modélisation du PL.
  Ces deux critères (Compréhension et facilité de modélisation) n'ont pas été évaluées de manière détaillée, car nous pensons que la contribution des ALs à leur amélioration est la même pour toutes les approches étudiées. D'autre part, ces deux critères sont soumis à d'autres influences telles que le type du langage de modélisation de PL utilisé.
- Concernant le critère de couverture des aspects de développement, les ALs n'influent pas sur ce critère car ce dernier dépend exclusivement du langage de modélisation de PL utilisé.
- La plupart des approches étudiées qui modélisent des PLs formels implémentent des mécanismes d'évolution (dynamique ou statique). Cependant, ces mécanismes sont divers et varient d'une approche à une autre. De plus, les évolutions offertes se situent toutes au niveau contenu et non au niveau structure, alors qu'il est possible d'exploiter l'évolution du modèle de PL au niveau structure(vision architecturale du modèle de PL).

Approche	Qualité de la modélisation	Mode d'exécution	Contrôle d'exé.	Évolution	
				Statique	Dynamique
Endeavors	-	Dynamique, Hétérogène, Distribué	-	Mécanismes d'intégration flexibles pour l'évolution incrémental du support	Déclarations dynamiques, modifications et extensions des champs, états, variables et interfaces objets.
PYNODE	Protocole spécifique à la composition dynamique	Dynamique, Hétérogène, Distribué	-	-	-
PLs à base de gestion de configuration	Contrôle de la compatibilité, consistance lors de la sélection des composants	Distribué	-	versionnement	-
OPC	-	Dynamique, Hétérogène, Distribué	Diags. état transition	-	Composition dynamique Changement dans les diagrammes d'état transitions des composants
APEL	-	Dynamique, Distribué	-	-	-
RHODES	Détecter les incohérences, gérer l'indéterminisme	Standard	-	mécanisme de polymorphisme et de sous typage	-
Simulation de procédé d'acquisition	-	Simulation	-	-	-
Assemblage de modèles basée connecteurs	-	-	-	-	-
Interactions PL basés ADL ZETA	-	standard	modèles d'interactions	-	-
Modélisation de PLs évolutifs	-	standard	réseau de pétri	-	intégré dans le composant : des composants évolutions
SPEM	-	non défini	orienté objet	classe Variability	-

Table 3.7 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe qualité.

3.4.5 Bilan et discussions

D'après l'étude des ces approches selon l'axe PL, nous remarquons qu'il n'y a pas de tendance particulière dans le domaine de la réutilisation des PLs à base d'ALs : les concepts manipulés sont souvent les concepts noyau du PL, et dans la plupart des cas, il n'y a pas de concepts spécifiques à la réutilisation des PLs. Aussi, les langages de modélisation de PLs sont des langages connus dans le domaine de l'ingénierie des PLs et non spécifiques à la réutilisation des PLs.

D'autre part, nous constatons que l'une des spécificités de base des PLs qui est la dimension humaine n'a pas eu beaucoup d'attention, l'aide aux développeurs et surtout le contrôle des interactions humaines n'ont pas été pris en charge de manière efficace.

D'autre part, d'après l'étude de ces approches selon l'axe AL, nous remarquons que les concepts architecturaux tels qu'ils sont définis dans le domaine des ALs ne sont pas bien exploités pour la réutilisation des PLs. Les concepts connecteur, configuration et style sont mal ou dans certains cas, pas du tout exploités. L'interprétation des connecteurs reste liée au langage de modélisation du PL utilisé, ce qui ne permet pas d'avoir des connecteurs fortement réutilisables. L'exploitation du concept configuration reste implicite en utilisant des représentations graphiques sans réflexion ni raisonnement à ce niveau lors de la phase de modélisation.

Les styles architecturaux de PLs n'ont pas encore fait leur apparition. Les solutions proposées restent intuitives et se basent plutôt sur un besoin de réutilisation d'un certain type de PL que sur la définition d'une méthodologie pertinente pour la réutilisation globale des PLs. La nouveauté de ce domaine (les architectures de PLs) est une justification de ces insuffisances, les solutions proposées sont souvent des solutions de modélisation à base de composants ce qui explique la faiblesse des concepts connecteur, configuration et style dans ces approches.

Il faut aussi noter qu'aucune approche n'offre des mécanismes de déploiement et de génération de code, ce qui limite l'efficacité de ces solutions.

En revanche, coté qualitatif, l'apport des ALs à la modélisation des PLs est considérable : L'exploitation même des ALs pour la modélisation des PLs contribue à la compréhension et à faciliter la modélisation des PLs. Aussi, les ALs offrent des mécanismes qui permettent de modéliser de manière plus facile des PLs distribués, hétérogènes ou dynamiques.

3.5 La réutilisation de PLs à base d'ontologies de domaine

Dans cette section, nous présentons l'étude des approches de modélisation de PLs à base d'ontologies de domaine. Comme mentionné dans l'introduction, cette étude a pour but, d'une part, d'identifier les possibilités de manipulation d'ontologies de domaine dans le domaine des PLs, d'autre part, elle permettra d'identifier une ontologie existante qui peut être exploité par notre approche.

3.5.1 Les approches de réutilisation à base d'ontologies de domaine

3.5.1.1 Environnement centré procédé à base d'ontologies

Dans [120], un environnement centré PLs est construit sur la base de deux ontologies [121] :

- Une ontologie objet qui spécifie tous les objets manipulés dans le modèle de PL : documents, tables, spécifications, conceptions,...etc. Les caractéristiques de base de ces objets sont définies de manière à pouvoir raisonner sur le type des objets manipulés par les PLs.
- Une ontologie procédés qui décrit les activités du PL. Les activités sont considérées comme des opérations effectuées sur des produits en entrée pour avoir des produits en sortie.

Ces deux ontologies sont connectées en respectant le schéma du procédé prédéfini et qui prend en considération les éléments de base d'un modèle de PL (activité, produit, rôle, ressource) et leurs relations. L'objectif de cet environnement est de générer un plan procédé logiciel (Software Process Plan -SPP-). Ce plan est généré après plusieurs étapes puis validé en utilisant les critères initialement prédéfinis du plan recherché.

3.5.1.2 Managing SP knowledge [41]

ProKnowHow est un environnement qui a pour objectif la gestion des connaissances procédé, il a pour but de :

- Supporter la modélisation de PLs standards.
- Collecter des connaissances PLs à partir d'expériences d'exécutions précédentes.
- Supporter la personnalisation des PL à partir de retours d'expériences capitalisés.

Cette solution se base sur l'exploitation d'une ontologie PL. Cette ontologie est développée pour supporter l'acquisition, l'organisation, la réutilisation et le partage des connaissances procédé. Elle est utilisée pour établir un vocabulaire en commun et faciliter le partage et la recherche des connaissances.

Les connaissances PLs sont organisées en deux parties :

- Une partie formelle qui permet de capitaliser les PLs.
- Une partie informelle qui permet de sauvegarder les retours d'expériences des projets réalisés.

3.5.1.3 SPO (Software Process Ontologie) [72]

SPO est une ontologie OWL utilisée pour décrire les PLs au niveau conceptuel. L'intérêt de cette ontologie est d'avoir une description formelle des modèles de PLs afin de permettre de faire des correspondances entre ceux de même type. Un modèle de pratique atomique (Atomoic Practice Model) est défini pour régler le problème de la granularité qui n'est pas forcément la même pour tous les PLs.

Les concepts de base de cette ontologie décrivent les éléments PLs tels que produit, activité et rôle.

3.5.1.4 Approche basée logique descriptive pour la maintenance logicielle [102]

Dans cette approche, une représentation ontologique à base de logique descriptive est utilisée pour présenter un modèle de PL formel pour la compréhension et la maintenance logicielle.

L'approche fournit un modèle qui intègre les informations concernant les ressources, les produits et les interactions entre eux. Son objectif est de fournir une assistance active lors de la maintenance logicielle. L'environnement fournit aussi la flexibilité et l'extensibilité requises pour supporter l'évolution, l'adaptation dynamique aux changements de l'environnement et la modélisation incrémentale du modèle de procédé de maintenance.

3.5.1.5 Le framework OnSSPKR (Ontology Supported Software process knowledge Representation) [61]

Ce Framework Permet de composer et de représenter les connaissances du PL, mais aussi, il permet de fournir un support pour gérer de nouveaux projets ou en cours de réalisation. Son ontologie regroupe trois types de connaissances qui couvrent les éléments de base du PL :

- Expériences du procédé qui couvre l'élément Unité de travail,
- Connaissances sur les produits couvre l'élément produit
- Aptitude du personnel qui couvre l'élément rôle.

Les concepts de base de l'ontologie sont tirés des concepts partagés entre les modèles de procédés : CMM, CMMI, ISO/IEC15504, ISO9001.

3.5.1.6 OWL Ontology pour le modèle CMMI- SW [111]

Comme son nom l'indique, cette approche fournit une ontologie OWL qui regroupe les concepts de base du modèle CMMI-SW, l'objectif est de permettre le raisonnement sur le niveau de maturité des entreprises en prenant en compte les informations fournies par celles-ci.

3.5.2 Analyse des approches de réutilisation à base d'ontologies de domaine

Approche	**Objectifs**	**Type ontologie**
Environnement centré procédé à base ontologies	Génération de plans de PLs	Ontologie objets qui décrit les produits manipulés, Ontologie activité qui décrit les activités du PL.
Managing SP knowledge	un vocabulaire en commun pour faciliter le partage et le recherche des connaissances	Ontologie contient des concepts décrivant le vocabulaire et terminologie.
SPO (Software Process Ontologie)	Décrit le PLs de manière formelle pour supporter son évaluation, peut être étendu pour décrire des procédés spécifiques (CMM, CMMI).	Concepts de base des procédés.
OWL Ontology pour le modèle CMMI- SW	Décrit les modèles CMMI-SW	Permet d'évaluer les entreprises, selon ce modèle.
Approche basée logique descriptive	Framework pour la maintenance logicielle	Concepts qui affectent la maintenance logicielle
Le framework OnSSPKR	Représentation contrôlée et partagée pour éliminer les confusions terminologiques et conceptuelles.(modèles CMM, CMMI, ISO/IEC15504, ISO9001,)	- Process experiences Ontology, - Personal skills ontology, - Knowledge artifacts ontology

Table 3.8 Les ontologies de domaine utilisées dans les approches de réutilisation de PLs à base d'ontologies.

Le tableau -3.8- résume les objectifs et les types d'ontologies des approches de réutilisation de PLs à base d'ontologies.

Nous avons présenté ces solutions de modélisation de PLs à base d'ontologies afin d'analyser les ontologies proposées pour la modélisation des PLs. Notre objectif est d'étudier la possibilité de réutiliser une des ontologies existantes.

Dans chaque approche, l'ontologie proposée contient soit un type de modèle de PL [72], soit des PLs de types similaires [61], soit des plans de PLs [120].

Dans la plupart des ces approches, l'utilisation de l'ontologie (ou de plusieurs ontologies) a pour intérêt le raisonnement sur un type particulier de PL plus que la réutilisation des modèles de PLs. Notre on-

tologie est conçue pour d'autres objectifs ; notre but est de regrouper les concepts utilisés dans différents modèles de procédés logiciels dans une seule ontologie de domaine. L'ontologie permettra l'exploitation des connaissances de différents modèles de PLs, en permettant tous type de raisonnement, inférence et d'analyse concernant ce domaine. Ces ontologies ne peuvent pas prendre en charge des PLs hétérogènes, par conséquent, elles ne peuvent pas être exploitées dans notre solution.

3.6 Conclusion

Nous avons abordé dans ce chapitre les approches de réutilisation de PLs à base d'ALs. Dans un premier temps nous avons présenté les approches les plus significatives qui ont apporté des solutions aussi bien diverses qu'intéressantes.

Afin de cerner les caractéristiques de ces approches nous avons défini notre propre cadre de comparaison. Notre cadre de comparaison s'inspire des cadres de comparaison établies dans le domaine des PLs et des cadres de comparaison établies dans le domaine des ALs en même temps.

Notre cadre de comparaison a été organisé comme suit :

- Un axe décrivant les aspects techniques des solutions de réutilisation selon la vue PL.
- Un axe décrivant les aspects techniques des solutions de réutilisation selon la vue AL.
- Un axe décrivant l'aspect qualitatif des solutions de réutilisation.

Ainsi, en se basant sur les critères de ce cadre de comparaison, nous avons étudié un ensemble de travaux issus du domaine de la réutilisation des PLs à base d'ALs. Nous avons conclu cette étude en apportant notre propre évaluation de ces approches de réutilisation. En se basant sur cette dernière, nous avons pu dresser un bilan récapitulatif. Ce que nous retenons essentiellement de ce bilan, comme cela a été mis en évidence à la fin du chapitre -2-, l'exploitation des ALs est considérée comme une solution très prometteuse pour la modélisation de PLs de qualité, elle permet non seulement d'augmenter la réutilisabilité des modèles de PLs, mais aussi d'augmenter d'autres critères de qualité tels que la compréhension et la facilité de modélisation. Aussi, cette approche(modélisation de PLs à base d'ALs) peut s'appliquer à différents PLs en utilisant différents concepts ou différents langages de modélisation ce qui augmente son efficacité en tant que solution de réutilisation.

Néanmoins, Il faut noter que dans les approches étudiées l'exploitation des concepts architecturaux n'a pas été optimale. Certains concepts tels que les styles architecturaux ne sont pas exploités, rares sont les solutions qui ont utilisé tous les éléments architecturaux en même temps, ce qui nous permet d'exploiter cette insuffisance pour proposer notre solution.

Dans ce chapitre, nous avons aussi présenté les approches les plus significatives qui exploitent une ontologie de domaine pour capitaliser les connaissances PLs. En étudiant ces solutions, nous remarquons d'aucune ontologie ne répond à notre cahier de charge principalement la capitalisation de connaissances de sources hétérogènes.

Dans le chapitre suivant, nous présentons notre propre approche de réutilisation de PLs à base d'ALs : l'approche AoSP (Architecture oriented Software Process).

CHAPITRE 4

L'approche AoSP (Architecture oriented Software Process)

4.1 Introduction

Malgré l'apport considérable des ALs, force est de constater que les approches de modélisation de PLs à base d'ALs étudiées sous exploitent les opportunités de réutilisation offertes par les ALs. De plus, la majorité de ces approches ne tirent pas avantage de l'expérience humaine et du savoir-faire capitalisés dans le domaine de l'ingénierie des PLs.

Dans ce chapitre nous introduisons notre proposition : AoSP (Architecture oriented Software Process), cette nouvelle approche de modélisation de PLs à base d'ALs repose sur le principe d'exploiter au mieux les concepts architecturaux afin de promouvoir la réutilisation des PLs.

AoSP couvre l'ingénierie "pour" et "par" la réutilisation des PLs, notre chapitre est donc organisé de manière à détailler les solutions adoptées pour couvrir ces deux ingénieries de réutilisation de PLs.

4.2 L'approche Architecture oriented Software Process (AoSP)

Architecture oriented Software Process (AoSP) est une approche pour la modélisation et l'exécution des PLs à base d'ALs. L'approche que nous proposons est bien originale du fait qu'elle exploite toute les opportunités afin d'améliorer la réutilisation des PLs, et cela, en mettant deux domaines de recherche prônant la réutilisation à large échelle (les architectures logicielles et les ontologies de domaine) au service de la réutilisation des PLs.

A travers l'approche AoSP nous nous fixons les objectifs suivants [11] :

- Proposer une nouvelle définition de la notion d'architecture de PL en exploitant au mieux les concepts, les techniques et les outils du domaine des ALs.
- Proposer une solution générale qui couvre la modélisation et l'exécution d'un large éventail de modèles de PLs.
- Améliorer la réutilisation des modèles de PLs existants en s'affranchissant des obstacles rencontrés lors de la réutilisation des PLs tel que l'hétérogénéité qui peut être celle des langages de modélisation de PLs, des concepts manipulés dans les modèles de PLs, ou bien, de la terminologie utilisée par les développeurs.
- Augmenter la réutilisabilité des nouveaux modèles de PLs en décrivant des architectures de PLs dont le déploiement peut être effectué avec différents langages de modélisation de PLs.

– Améliorer la qualité des modèles de PLs, et surtout combiner au mieux les caractéristiques difficilement conciliables telles que l'agilité et le contrôle d'exécution.

L'approche AoSP couvre les deux ingénieries de réutilisation [9] :

– "Pour" la réutilisation de PLs, en utilisant une ontologie de domaine qui capitalise les expériences positives des modélisations et des exécutions d'anciens modèles de PLs.
– "Par" la réutilisation de PLs, en permettant la description d'architectures de PLs (à partir des connaissances de l'ontologie), puis son déploiement pour la génération du modèle de PL final.

Modéliser les PLs en exploitant les ALs modifie le processus de modélisation, ainsi, au lieu de modéliser le PL en une seule étape, la modélisation du PL est réorganisée en deux étapes :

– La pré-modélisation : consiste à décrire l'architecture de PL et les éléments permettant son déploiement. Cela revient à modéliser séparément les différents modules réutilisables du PL (structure, interaction et traitement). Séparer les préoccupations permet non seulement d'améliorer la réutilisabilité des PLs, mais aussi, d'augmenter leur compréhension et leur facilité de modélisation. Aussi, cette séparation des préoccupations a un impact direct sur l'analyse et le contrôle d'exécution des modèles de PLs. Les différents éléments architecturaux de l'architecture de PL sont récupérés à partir de l'ontologie de domaine.
– La modélisation finale : consiste à faire le déploiement de l'architecture de PL. Le déploiement doit être de manière automatique. Cette possibilité, donne à l'approche un aspect général et lui confère la possibilité d'être étendue à la génération de différents modèles de PLs. L'extension de l'approche est possible en développant d'autres programmes de déploiement d'architectures de PLs.

4.3 Ingénierie pour la réutilisation de procédés logiciels

Cette étape tente de pallier la faible réutilisabilité des éléments architecturaux et de tirer avantage de la maturité du domaine des PLs en matière d'expérience et de meilleures pratiques adoptées par les développeurs.

Pour capitaliser les connaissances du domaine des PLs, la solution envisagée est l'utilisation d'une ontologie de domaine regroupant les concepts spécifiques aux PLs. L'ontologie de domaine constitue alors un socle qui permettra de contenir les connaissances PLs, et permettra ainsi, la réutilisation de modèles de PLs hétérogènes indépendamment de leurs environnements d'origine. Elle permettra aussi, de réutiliser des modèles de PLs qui ne sont pas initialement dédiés à la réutilisation formelle (non orientés composants par exemple) et qui sont réutilisés de manière informelle à travers la consultation manuelle des rapports d'exécution et des comptes rendus [26]. L'ontologie instanciée permettra aussi de récupérer les connaissances nécessaires à la description d'architectures de PLs (figure -4.1-). L'autre avantage de l'utilisation d'une ontologie au lieu d'une autre structure de stockage est la possibilité de raisonner, d'analyser et d'inférer de nouvelles connaissances qui assureront l'émergence de nouvelles solutions de modélisation de PLs.

Afin de capitaliser l'expérience du domaine des PLs, l'instanciation de cette ontologie se fait à partir de modèles de PLs existants. L'acquisition de ces connaissances passe nécessairement par une phase de ré-ingénierie, en effet, des programmes d'instanciation (instanciateurs) à partir de modèles de PLs doivent être développés à cet effet. Chaque langage de modélisation de PLs doit avoir son propre instanciateur qui permettra la capture des connaissances réutilisables. Une instanciation pertinente est celle qui permet d'identifier de manière unique chaque instance de l'ontologie. L'instanciation à partir de plusieurs modèles de PLs se heurte au problème de l'hétérogénéité du vocabulaire et des concepts utilisés,

ainsi, il est important de définir une stratégie pour gérer cette hétérogénéité.

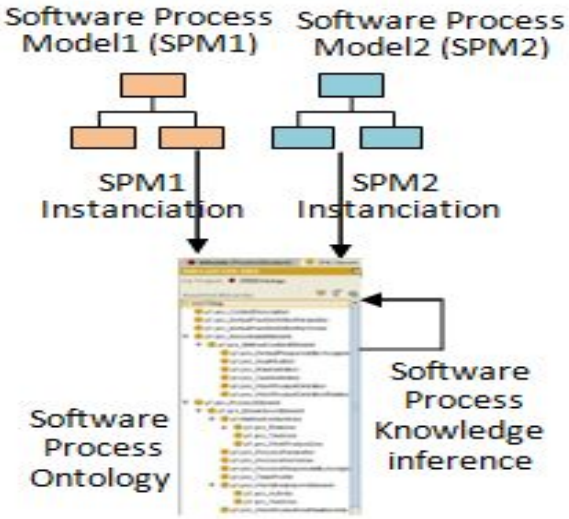


Figure 4.1 Les étapes de l'ingénierie "pour" la réutilisation de PLs.

4.3.1 La sémantique adoptée pour la notion d'architectures de procédés logiciels

Afin de détailler l'étape de l'ingénierie pour la réutilisation, la définition de la notion d'architecture de PL est indispensable, en effet, les approches étudiées n'offrent pas de définition précise et normalisée de la notion d'architecture de PL, chaque solution avance sa propre définition concernant les éléments architecturaux manipulés. Notre interprétation des éléments architecturaux est gouvernée par deux points essentiels :

- La nécessité d'exploiter de manière optimale les concepts architecturaux.
- La nécessite de respecter les caractéristiques des PLs telles que l'importance de la dimension humaine et les exécutions diverses et correctes du même modèle de PLs.

Afin de proposer la sémantique adéquate nous explorons les métamodèles d'ADLs (Architectures Description Laguages) existants décris en UML [99] [115] [125] [7], l'objectif est de s'inspirer des métamodèles formels déjà mis en place pour proposer notre propre métamodèle.

L'identification des concepts architecturaux pour les PLs se base sur l'approche ADL, car les ADLs ont une charge sémantique plus riche que les approches classiques, ils introduisent des concepts architecturaux, des techniques et des outils qui permettent de décrire rigoureusement des ALs de manière explicites. Cette étape a comme résultat l'introduction des concepts architecturaux manquants, le raffinement de la sémantique des concepts architecturaux existants, puis la formalisation de notre métamodèle d'architecture de PL.

L'importance des interactions dans les PLs [6], la diversité des flux circulant dans les PLs [43] nous suggèrent de définir des connecteurs de PLs qui peuvent prendre en charge ces différentes interactions. Dans notre approche le connecteur PL est considéré comme une entité de première classe.

Nous définissons notre connecteur PL comme une activité qui permet de "faciliter et contrôler" les transitions entre les activités PL. Contrairement, au "composant PL" le "connecteur PL" ne crée pas de nouveaux produits, mais "adapte et contrôle" les produits existants.

La distinction entre les activités de "création" de produits (qui constitueront les composants PL) et les activités "d'adaptation et de contrôle" de flux (qui constitueront les connecteurs PL) modifie la

sémantique des concepts identifiés dans les approches étudiées. Ainsi, notre interprétation des concepts architecturaux des PLs sera comme suit (tableau-4.1-) :

Concept Procédé Logiciel.	**Interprétation selon les concepts architecturaux**
Activité de **"création"** de nouveaux produits logiciels.	Composant PL.
Donnée (en entrée /en sortie) d'une activité de création.	Port PL : peut être un Port flux de données ou un Port flux de contrôle.
Structure procédé logiciel.	Configuration PL : ensemble de composants PL et connecteurs PL respectant des contraintes d'assemblage.
Cycle de vie du logiciel.	Stye PL : introduit formellement avec les concepts "Types", les invariants et les contraintes.
Données fournies ou requises (données en entrée ou en sortie) d'une activité d'adaptation.	Rôle connecteur PL : peut être un rôle connecteur "Data Flow" ou un rôle connecteur "Control Flow".
L'activité **"d'adaptation ou de contrôle"** des données.	Connecteurs explicites : taxonomie de connecteurs prédéfinies.
Lien de précédence entre une activité de création et une activité d'adaptation.	Attachement : le lien entre un port PL et un rôle connecteur PL de même type.
Lien de délégation entre activités.	Binding : un lien entre les ports PL ou entre les rôles connecteur PL de même type.

Table 4.1 Les correspondances adoptées entre les concepts de PLs et les concepts ALs.

- **Composant PL (SP Component) :** décrit un traitement réalisé sur des produits en entrée pour "la création" de nouveaux produits en sortie.

- **Port PL (SP Port) :** l'interface d'un composant est un ensemble de points d'interactions du composant PL ; elle spécifie les services fournis ou requis nécessaires à l'exécution du composant PL. L'interface du composant PL est un ensemble de "ports PL". Les ports requis correspondent aux "flux en entrée" nécessaires à l'exécution du composant PL. Les ports fournis correspondent aux "flux en sortie" du composant PL. Deux types de ports PL sont définis :
  - Ports Data Flow (Ports flux de données) : sont des points d'interactions spécifiques à la transmission des produits des composants PL. Selon le sens des ports PL, ils permettent le transfert des produits résultats du composant PL ou bien le transfert des produits requis par un composant PL.
  - Ports Control flow (Ports flux de contrôle) : sont des points spécifiques aux flux d'exécution des modèles de PLs, ils permettent d'identifier l'ordre et l'état d'exécution des composants PL.

- **Connecteur PL (SP Connector) :** Décrit le traitement à réaliser sur des produits en entrée afin de "les adapter ou les contrôler" pour les besoins du composant PL connecté.

- **Rôle connecteur PL (SP Role) :** L'interface d'un connecteur PL est représentée par les "Rôles connecteur PL". De la même manière que les ports PL, ils représentent les points d'interaction pour la transmission des flux (produit ou flux d'exécution) requis ou fournis par les connecteurs PL. Comme pour les ports PL deux types de Rôles connecteur PL sont définis : les rôles connec-

teur Data Flow et les rôles connecteur Control Flow.

- **Binding :** une configuration peut être composée hiérarchiquement d'un ou de plusieurs sous-composants. Les connexions entre les ports PL d'une configuration PL avec les ports PL de ses sous composants sont appelées des "Bindings". Le binding permet de formaliser la délégation du flux de données (respectivement flux d'exécution) aux éléments internes de la configuration PL. De la même manière, les connecteurs PL peuvent être composés de sous-connecteurs. Le binding décrit aussi la connexion entre le rôle d'un connecteur PL composite et le rôle de son sous-connecteur PL. Les bindings sont possibles qu'entre des ports PL de même type, ou entre des rôles Connecteur PL de même type (Data Flow ou Control Flow).

- **Attachement (Attachment) :** est le lien entre le port PL d'un composant PL et un Rôle connecteur PL d'un connecteur PL. L'attachement se fait entre le port PL et rôle connecteur PL de même type (Data Flow ou Contrôle Flow). Selon le type du port PL et du rôle connecteur PL connectés (Data Flow ou Control Flow), l'attachement permet de formaliser soit la transmission de données ou bien l'ordre d'exécution entre composant PL et connecteur PL.

- **Configuration PL (SP Configuration) :** elle décrit l'agencement logique des composants PL et des connecteurs PL en déterminant explicitement les contraintes d'assemblage de la structure du modèle de PL. Une configuration PL peut respecter un ou plusieurs styles PL.

- **Style PL (SP Style) :** permet de formaliser les caractéristiques (contraintes d'assemblages, types des éléments architecturaux...etc.) des structures des PLs et des stratégies d'exécution récurrentes. Les styles de PLs permettent de capitaliser l'expérience et le savoir-faire de la modélisation et de l'exécution des PLs.

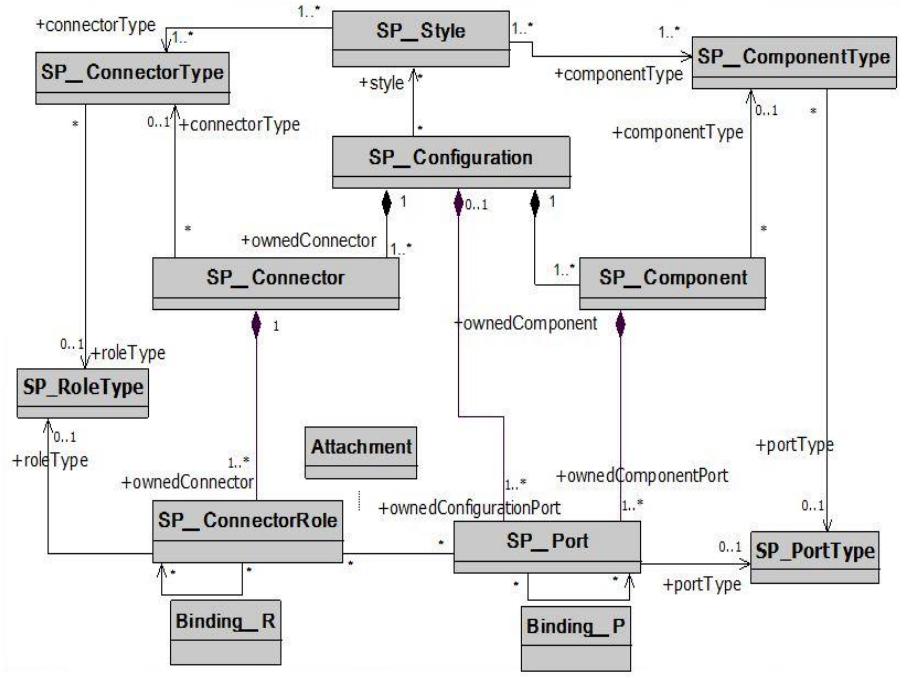


Figure 4.2 Le métamodèle d'architecture de procédé logiciel.

La figure -4.2- représente le métamodèle regroupant les concepts architecturaux identifiés pour la description des architectures de PLs.

Ainsi, comme pour les ALs, une "SP configuration" est un assemblage de "SP Component" connectés à travers des "SP Connectors". La description de la structure interne des "SP Connectors" et des "SP Components" se fait à travers les "Binding". Le "SP Style" est décrit à travers des associations aux classes "Type" des différents concepts.

Nous introduisons les concepts : "SP Component Type", "SP Connector Type", "SP Role Type" et "SP Port Type" qui correspondent respectivement à "Type Composant PL", "Type connecteur PL", "Type Role PL" et "Type Port PL" pour décrire formellement les styles architecturaux des PLs. Pour l'instant, ces concepts sont définis indépendamment des concepts PL, la correspondance directe entre ces concepts type et les concepts PLs n'est pas évidente à ce niveau, néanmoins, ces concepts aurons une interprétation plus formelle dans les sections suivantes.

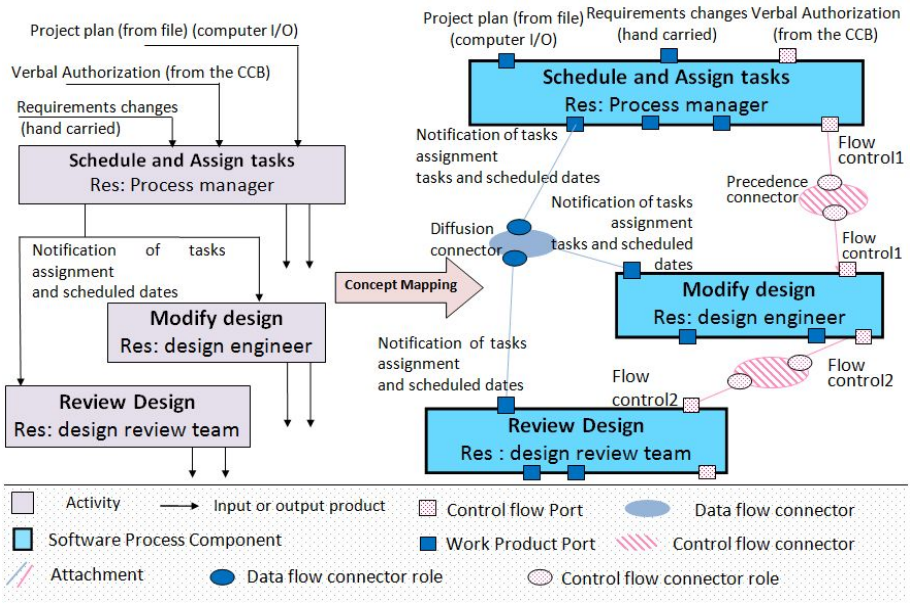


Figure 4.3 L'architecture de PL selon la sémantique adoptée.

La figure-4.3- illustre bien la vue architecturale du modèle de PL, elle présente l'interprétation de l'exemple ISPW-6 [70] en architecture de PL selon la sémantique adoptée. Nous constatons que les flux d'exécution (représentés par les connecteurs "Precedence") sont explicites. Aussi, la diffusion des données (représentée par le connecteur "Diffusion") est aussi explicite, ce qui n'est pas le cas dans la représentation standard.

Afin d'exploiter formellement la sémantique adoptée, il est important d'utiliser un outil qui permet d'expliciter l'architecture de PL. Comme cité dans le chapitre -2-, nous avons opté pour l'utilisation de l'ADL ACME.

La figure -4.4- illustre une vue partielle du code ACME de l'architecture de PL décrite dans la figure précédente (exemple ISPW-6). L'utilisation de l'ADL ACME pour la description d'architectures de PL est adéquate et n'a posé aucun problème particulier. Ce constat est logique, car la sémantique des architectures de PLs se base sur la sémantique des ADLs.

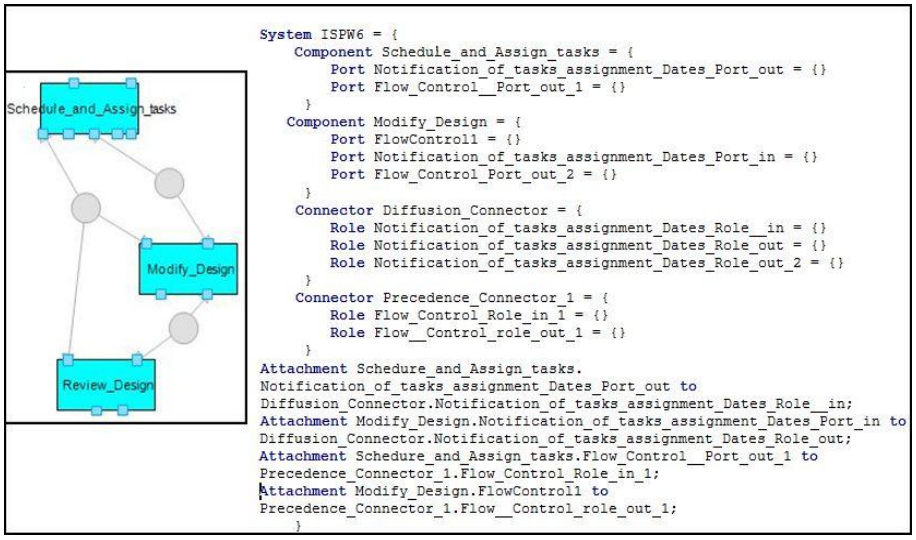


Figure 4.4 Architecture de PL décrite avec l'ADL ACME (Vue partielle exemple ISPW).

4.3.2 Les connecteurs de Procédés Logiciels

Comme préconisé par la sémantique adoptée, nos connecteurs PLs sont des entités de première classe et peuvent être définis explicitement. En analysant la structure des modèles de PLs, et en prenant en compte les propositions des approches étudiées [88] [43] [42], nous définissons deux types de connecteurs PL :

4.3.2.1 Les connecteurs Data Flow

En analysant les activités des modèles de PLs, nous remarquons que certaines activités de transmission et d'adaptation de données sont récurrentes [43]. Des activités de transmission telles que : la diffusion, la fragmentation ou la conversion de données permettent la gestion des transferts de données, et de plus, elles sont indépendantes du type du modèle de PL. Nous nous inspirons des activités d'adaptation de l'environnement APEL [43] pour définir nos connecteurs Data Flow.

Nos connecteurs Data Flow sont comme suit :

- **Le connecteur Transmission :** C'est le connecteur PL de base de cette catégorie, il s'occupe de la transmission des données (produits) d'un composant PL à un autre. Sa caractéristique principale est qu'il possède un seul Port Data Flow en entrée et un seul Port Data Flow en sortie.

- **Le connecteur Adaptation :** Permet d'adapter ou de convertir puis transmettre le produit en entrée en un produit en sortie selon les exigences du composant PL connecté. Sa caractéristique principale est qu'il possède un Port Data Flow en entrée et un seul Port Data Flow en sortie.

- **Le connecteur Fusion :** Permet de fusionner un ensemble de produits rentrants on un seul produit sortant. Sa caractéristique principale est qu'il possède plusieurs Ports Data Flow en entrée et un seul Port Data Flow en sortie.

- **Le connecteur Diffusion :** Permet de récupérer le produit d'un composant PL, puis de les diffuser sur l'ensemble de composants PL connectés. Sa caractéristique principale est qu'il possède un Port

Data Flow en entrée et plusieurs Ports Data Flow en sortie.

- **Le connecteur Fragmentation :** Permet de fragmenter un produit en entrée en un ensemble de sous produits en sortie ; de les transmettre aux composants PLs connectés. Sa caractéristique principale est qu'il possède un Port Data Flow en entrée et plusieurs Ports Data Flow en sortie.

4.3.2.2 Les connecteurs Contrôle Flow

Ces connecteurs correspondent à des activités de transfert et de contrôle de flux d'exécution. ces connecteurs permettent d'assurer l'ordre d'exécution des composants PLs, certains d'entre eux permettent d'analyser la qualité de l'exécution de l'architecture de PL.

Les activités correspondantes à ces connecteurs appartiennent au domaine de la gestion de projets logiciels. La qualité de l'exécution peut être analysée selon trois aspects : le temps de l'exécution, le coût de l'exécution et la qualité du produit (résultat de l'exécution).

En se basant sur ces principes, nous définissons trois catégories de connecteurs Control Flow :

- **Les connecteurs ordre d'exécution (Execution Order connectors) :** Formalise l'ordre d'exécution des composants PL, il assure l'exécution standard sans évaluation ou traitement particulier sur le flux d'exécution. Ces connecteurs explicitent les ordres d'exécution traditionnels des modèles de PLs et qui dépendent du début et de la fin des activités : (fin-début, début-début, début-fin et début-début) [29]. Dans notre travail, nous utilisons le connecteur Precedence (fin-début) comme connecteur de base pour décrire l'ordre d'exécution des composants PL. Le connecteur Précédence (fin-début) précise que la "fin" d'exécution d'un composant PL déclenche le début d'exécution du composant PL connecté.

- **Les connecteurs Évaluation (Evaluation connectors) :** Ces connecteurs sont définis pour évaluer l'exécution des composants PL. Ces connecteurs peuvent être exploités dans la modélisation de PL à exécution dynamique centré humain, en d'autres termes, des exécutions de modèles de PLs où le gestionnaire du PL humain décide (après consultation des rapports d'exécution émis par les connecteurs Evaluation) de la progression de l'exécution du modèle de PL.
  Selon l'orientation de l'exécution du modèle de PL, nous distinguons trois types de connecteurs :
  - **Le connecteur ToE (Time oriented Evaluation) :** Évalue les paramètres qui concernent le temps d'exécution du composant PL.
  - **Le connecteur CoE (Cost oriented Evaluation) :** Évalue les paramètres qui concernent le coût de l'exécution du composant PL.
  - **Le connecteur QoE (Quality oriented Evaluation) :** Évalue les paramètres qui concernent la qualité du résultat de l'exécution du composant PL.
- **Les connecteurs Décision (Decision connectors) :** Sont définis pour évaluer puis émettre des décisions concernant la progression de l'exécution du modèle de PL. Ce type de connecteurs PLs est adéquat aux exécutions dynamiques centrés procédés, ces connecteurs sont les mêmes que les connecteurs "Evaluation", cependant, après évaluation des paramètres adéquats, ils émettent des décisions de progression d'exécution, qui, après validation du gestionnaire du PL (humain) sont appliquées automatiquement. Ces connecteurs sont : ToD (Time oriented Decision), CoD (Cost oriented Decision) and QoD (Quality oriented Decision) (figure-4.5-).

La liste des connecteurs PLs que nous avons défini n'est pas exhaustive, en effet, la définition du connecteur PL telle qu'elle est adoptée (activité d'adaptation) permet d'identifier d'autres connecteurs PLs.

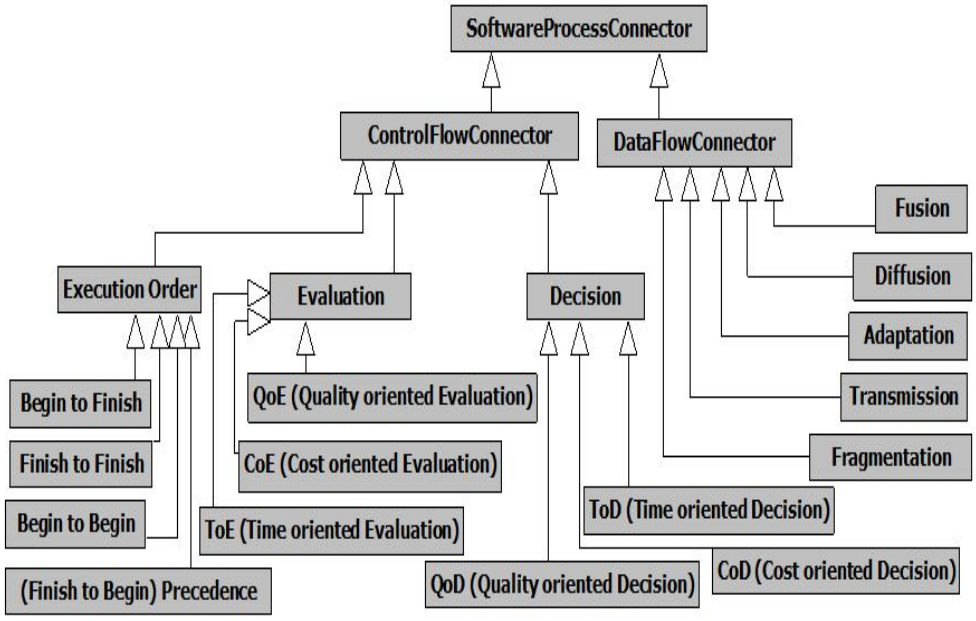


Figure 4.5 La taxonomie des connecteurs de procédés logiciels.

4.3.2.3 La propriété Execution_Mode du connecteur PL

Afin de distinguer les interactions humaines des interactions automatiques, les connecteurs PL sont munis d'une propriété Execution_Mode qui détermine le type de l'exécution du connecteur PL, cette propriété peut avoir la valeur :

- Automatique : le connecteur s'exécute automatiquement sans intervention humaine, par exemple : le connecteur Diffusion.
- Manuelle : le connecteur s'exécute avec une intervention humaine, par exemple : le connecteur Fragmentation.
- Verbale : le connecteur représente une interaction entre humain qui n'a pas de support (humain) ; par exemple : certains connecteurs Precedence où l'ordre d'exécution est donné oralement.

4.3.3 Les styles architecturaux de PLs

En définissant les concepts architecturaux pour la description des architectures de PLs, nous pouvons envisager de définir des styles architecturaux spécifiques aux architectures de PLs.

Capturer les enchainements et les structures récurrentes est l'objectif principal d'un style architectural. Dans le domaine des PLs, cette idée n'est pas récente, en effet, la plupart des structures récurrentes des PLs ont été déjà identifiées et formalisées, et cela, par la définition des cycles de vie de logiciel [28], des patrons de PLs [117] et des meilleures pratiques de développement [65] [18].

Le rapprochement entre le cycle de vie de logiciel et le style architectural a été déjà abordé par Boehm [25], en ce basant sur ces constatations, décrire les cycles de vie de logiciel, les partons de PLs comme des styles de PLs est incontestable. Ainsi, un style de PL doit permettre de capturer les spécificités des structures récurrentes dans les modèles de PLs.

D'un autre coté, les PLs ont une caractéristique que les logiciels n'ont pas : un modèle de PL peut s'exécuter de manière différente selon les conditions du développement. En effet, lors de l'exécution du

modèle de PL, le chef de projet peut donner la priorité au temps, au coût ou à la qualité de réalisation. La politique d'exécution se base souvent sur le type du projet, l'expérience du chef de projet ainsi que sur les conditions d'exécution du modèle de PL. D'autre part, des évènements non planifiés tels que l'absence de ressources ou la difficulté de réalisation de certaines tâches peuvent surgir et dévier l'exécution du modèle de PL.

Les priorités de développement et les imprévus d'exécution influent directement sur l'exécution du modèle de PL. Par conséquent, une instance d'un modèle de PL peut avoir plusieurs instances d'exécutions sans pouvoir différencier la bonne de la mauvaise exécution. N'ayant pas de vision explicite de l'exécution souhaitée, il est difficile de contrôler l'exécution et de prendre les bonnes décisions d'adaptation, ce qui peut induire à l'échec du projet de développement. De plus, les approches de modélisation de PLs existantes n'ont pas envisagé d'exploiter les expériences d'exécution de manière formelle. En effet, les stratégies d'exécution récurrentes ne sont ni capitalisées ni réutilisées de manière formelle. Les stratégies d'exécution dépendent entièrement des connaissances tacites du gestionnaire du PL, ce qui est considéré comme un grand inconvénient.

Dans notre travail, un style architectural de PL doit permettre non seulement la capture des caractéristiques des structures et des enchainements récurrents ; mais aussi, la capture des caractéristiques des politiques d'exécution récurrentes.

4.3.3.1 Les éléments de base du style d'architecture de procédé logiciel

Un style architectural est défini selon ses composants types, ses connecteurs types et ses invariants qui décrivent ses caractéristiques intrinsèques [55]. Dans notre cas, cette définition reste valable, les invariants permettent la description des règles qui régissent les enchainements prédéfinis des composants PL Types et des connecteurs PL types.

Afin d'identifier les stratégies d'exécution récurrentes, et de les introduire explicitement, nous définissons un style architectural de PL comme une combinaison d'un style structurel qui capture les caractéristiques de la structure du PL et d'un style d'exécution qui capture les caractéristiques de la stratégie d'exécution du PL (tabeau-4.2-). Cette combinaison permet de dissocier le style de l'exécution du style de la structure PL ce qui nous permettra une meilleure visibilité et un meilleur contrôle de l'exécution du modèle de PL.

Catégorie du style	**Style structurel**	**style d'exécution**
Objectif	Capture les structures récurrentes.	Capture les stratégie d'exécutions récurrentes.
Composant PL Types	component PL types selon le style structurel	–
Connecteur PL Types	connecteur Precedence, connecteur DataFlow	connecteur Control Flow
Invariants	décrivent les composants Types et les connecteurs Types séquences, cardinalités et compositions.	

Table 4.2 Les caractéristiques des styles architecturaux de PLs.

4.3.3.2 Les styles structurels pour les architectures de procédés logiciels

Le style structurel de PL est caractérisé par :

– Les composants PL Types varient selon le style structurel définit. Ces composants PL Types peuvent être des phases de développement telles que : conception, réalisation, test...etc., des itérations telles que : itération, sprint...etc. ou des étapes et des activités connues telles que correction code, Audit, réunion...etc.

– Les connecteurs PL Types sont de catégorie Data Flow. Cette catégorie de styles exploite aussi le connecteur "Precedence" ; ces styles se focalisent principalement sur les transmissions des produits, L'ordre d'exécution est standard et est formalisé par le connecteur "Precedence".

– Les contraintes qui régissent les enchainements de ses composants PL Types et de ses connecteurs PL Types. Ces contraintes sont classées en trois catégories : contraintes sur le nombre d'instances, contraintes sur les enchainements et contraintes sur les compositions.

Nous nous inspirons des travaux réalisés concernant les cycles de vie de logiciels, les processus agiles et les patrons de PLs pour proposer deux exemples de styles structurels (styles structurel en V et le style structurel UP). Néanmoins, il faut souligner qu'il est possible de capturer les caractéristiques de structures récurrentes et qui ne sont pas connues.

4.3.3.3 Les styles d'exécution pour les architectures de procédés logiciels

Définir des styles d'exécution permet de capitaliser puis réutiliser les expériences et les stratégies d'exécution des modèles de PLs. La définition de la politique d'exécution fait partie de la discipline de gestion de projets logiciels. En effet, l'exécution du PL est régie par les paramètres qui sont évalués dans des activités de gestion de projets logiciels. Ces paramètres expriment souvent l'état des trois aspects essentiels : temps, coût et qualité. La définition de nos styles d'exécution de PLs repose sur ces trois aspects et leur évaluation. Ayant définit des connecteurs Control Flow qui permettent l'évaluation de l'exécution d'un composant PL selon ces aspects, nous exploitons ces connecteurs pour définir nos styles d'exécution de PLs.

En conséquence, un style d'exécution est caractérisé par des connecteurs Control Flow sans contraintes particulières sur les types des composants. Dans notre cas, les types des composants n'ont pas d'impact sur le type d'exécution du PL. Les instances des connecteurs Control Flow sont programmées et paramétrées selon les besoins de l'exécution du modèle de PL.

Dans notre travail nous définissons deux catégories de styles d'exécution :

– **Les styles d'exécution dynamiques centrés humains :** ces styles sont définis pour capturer les caractéristiques des exécutions dynamiques centrés humain où les décisions de modification sont prises par le chef de projet et non pas automatiquement par le modèle de PL. Ces styles d'exécution exploitent les connecteurs "Evaluation", ce qui nous permet de définir les trois styles suivants :
  - Le style d'exécution évaluation centrée temps : Décrit par le connecteur Type ToE (Temps oriented Evaluation).
  - Le style d'exécution évaluation centrée coût : Décrit par le connecteur Type CoE (Cost oriented Evaluation).
  - Le style d'exécution évaluée centrée qualité : Décrit par le connecteur Type QoE (Quality oriented Evaluation).

– **Les styles exécution dynamiques centrés procédés :** ces styles sont définis pour décrire les exécutions des modèles de PLs qui supportent des modifications dynamiques. Dynamique dans le

sens ou le connecteur Control Flow évalue l'exécution selon des critères prédéfinies, le connecteur Control Flow émet une décision de progression d'exécution et éventuellement une proposition de modification qui sont validées (ou pas) par le chef de projet. Ces styles d'exécution exploitent les connecteurs "Decision", nous définissons, donc, les trois styles suivants :
- Le style d'exécution décision centrée temps : où le connecteur type est le connecteur ToD (Time oriented Decision).
- Le style d'exécution décision centrée coût : où le connecteur type est le connecteur CoD (Cost oriented Decision).
- Le style d'exécution décision centrée qualité : où le connecteur type est le connecteur QoD (Quality oriented Decision).

Le style d'exécution peut être appliqué de deux manières :

– En association avec un style structurel : où le connecteur Précédence du style structurel est remplacé par le connecteur Control Flow du style d'exécution. Cette combinaison confère au style structurel un style d'exécution particulier.
– Seul sans style structurel particulier : Dans ce cas c'est juste les connecteurs Control Flow de type Evaluation ou Decision qui sont utilisés dans la configuration PL défini.

4.3.3.4 Les propriétés types des styles de procédés logiciels

Afin de compléter la description des styles architecturaux des PLs, nous définissons des propriétés permettant de spécifier les éléments types des styles PL. Ces propriétés s'inspirent du domaine de la gestion de projets logiciels et dépendent des méthodes et techniques suivies dans ce domaine, ces propriétés sont classées en trois catégories :

– **Propriétés décrivant le temps d'exécution :** Ces propriétés permettent de décrire les paramètres concernant l'aspect temporel de l'élément architectural. Si nous prenons comme exemple les estimations de temps de réalisation selon les techniques du diagramme de Gantt [29], nous pouvons définir les propriétés suivantes : durée de l'exécution, date début exécution estimée, date de début exécution au plus tôt, date de début exécution plus tard, marge de retard libre, date de réalisation effective, retard effectué...etc.

– **Propriétés décrivant le coût de l'exécution :** Ces propriétés permettent de décrire les paramètres concernant le coût de l'exécution de l'élément architectural. Elles respectent généralement un modèle d'estimation et d'évaluation des coûts particulier. Si nous prenons comme exemple le modèle d'estimation des coûts COCOMO [29], les propriétés peuvent être : l'effort (Homme-mois) estimé, l'effort effectif, nombre de développeurs, productivité...etc.

– **Propriétés décrivant la qualité du produit de l'exécution :** Ces propriétés permettent de décrire les paramètres concernant l'aspect qualitative des résultats de l'élément architectural. Ces propriétés peuvent être des métriques d'évaluation de la qualité ou des critères d'un modèle d'évaluation de qualité. Elles dépendent généralement du type du produit résultat (code source, diagramme, spécification, documentation...etc.) et du modèle de qualité adopté. Par exemple, si le produit est un code source, les propriétés définies pourrons servir à évaluer sa complexité architecturale (nombre de fonctions, nombre d'appels de fonctions, nombre d'appels imbriqués...etc.) ou à évaluer sa complexité structurelle (nombre d'instructions imbriquées, le nombre d'exécutions possibles...etc.).

4.3.3.5 Exemple 1 : Le style structurel en V

Afin de montrer la possibilité de décrire les styles architecturaux de PLs en respectant les définitions présentées ci-dessus. Nous présentons deux exemples de styles structurels : le style en V qui s'inspire du cycle de vie en V et le style UP qui s'inspire du processus UP.

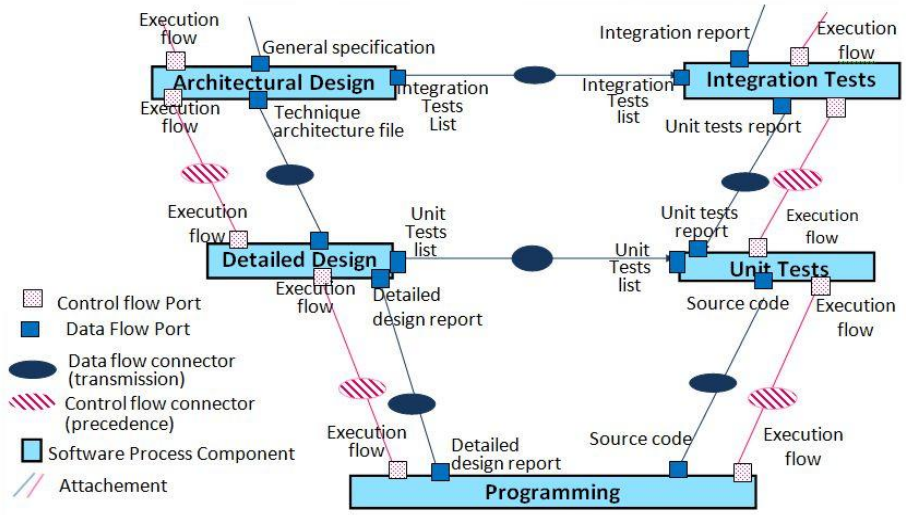


Figure 4.6 Le style structurel en V.

Le cycle de vie d'un logiciel est considéré comme un modèle descriptif qui permet de décrire les étapes et les enchainements connus et reconnus pour de la réalisation d'un produit logiciel. Le cycle de vie en V est décrit comme un modèle en cascade dans lequel les étapes de développement et la mise en place des tests sont effectuées de manière synchrone.

La figure-4.6- illustre une vue partielle du style en V ; le tableau-4.3-résume les caractéristiques de ce style :

SP Component Type	SP Connector Type
Needs Specification, Anlysis, Design, Programming, Test.	Transmission, Precedence
Cardinalités : - Une instance de : Needs Specification, Anlysis, Design, Programming, - Plusieurs instances de : Test, Precedence, Transmission. **contraintes d'assemblage :** -Chaque composant Type est relié à au moins deux -Chaque composant Type est relié à au plus à trois composants types. - Chaque composant Type qui est différent de composant type "Test" est relié une fois à des composants type "Test" à travers un connecteur Data Flow. - Le composant type "Test" est relié à au plus deux composants de type "Test". - Un composant PL type "Analyse" et "Spécification des besoins" ne peuvent pas être relié à un composant de type "Programming".	

Table 4.3 Description du style structurel en V.

Nous constatons que la définition de ce style est assez simple, les contraintes sur les cardinalités et sur les enchainements sont facilement identifiables. La structure interne des composants types n'est pas

définit dans cet exemple, cependant, il est possible de rajouter d'autres contraintes et d'autres composants types pour décrire la structure interne du style.

4.3.3.6 Exemple 2 : Le style structurel UP (Unified Process)

UP Unified Process est une démarche de développement générique utilisée pour définir un large types de PLs respectant certaines caractéristiques ; c'est un processus de développement logiciel itératif et incrémental, centré sur l'architecture et piloté par les cas d'utilisation. UP est décrit comme générique et exploité comme patron pour décrire différents types de PLs itératifs et incrémentaux. Nous définissons le style structurel UP qui permettra de décrire puis déployer des architectures de PLs itératives et incrémentales.

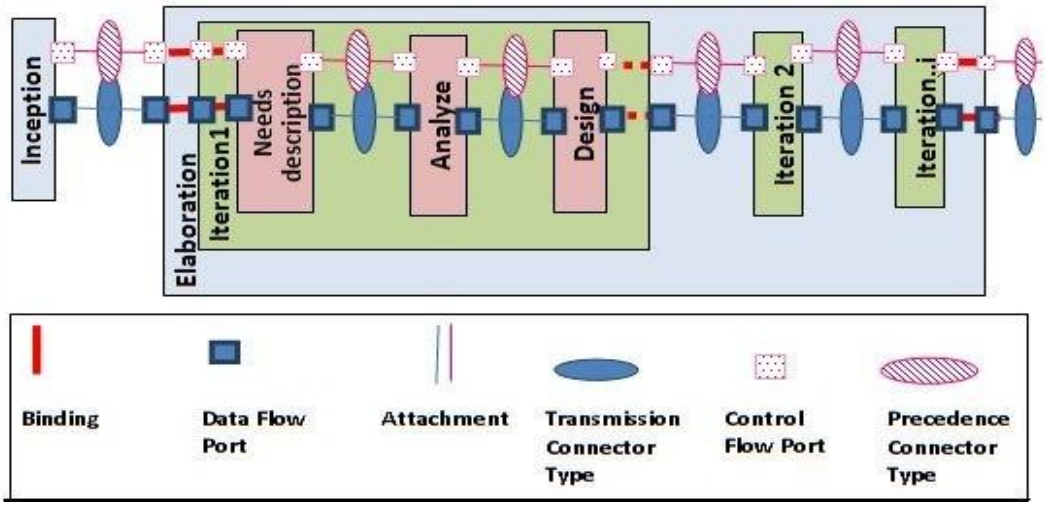


Figure 4.7 Le style structurel UP.

La figure -4.7- illustre l'architecture de PL selon le style UP. Le tableau -4.4- détaille les éléments de ce style, le style UP est constitué des composants types décrivant ses phases et ses itérations. Ses connecteurs PLs sont de type "Transmission" et "Precedence". Dans ce style, la structure interne de chaque composant est détaillée, ainsi, nous remarquons la définition de contraintes de composition.

Comme le style en V, ce style peut être décrit à l'aide de l'ADL ACME, la figure- 4.8 illustre le style structurel UP. Toutes les contraintes définies peuvent être décrites à l'aide de cet ADL.

SP Component Type	SP Connector Type
Inception, Elaboration, Construction, Transition, Planning, Needs specification, Analysis, Design, programming, Test	Transmission Precedence
Cardinalitiés : - Une instance de Inception, Elaboration, Construction, Transition. - Plusieurs instances de : iteration, planning, Needs specification,Conception, Analysis Coding, test. - Plusieurs instances de : Communication, Precedence. **Contraintes d'assemblage :** -Une sequence de : Inception, Elaboration, Construction, Transition. - une ou plusieurs séquences des activités : Planning, Needs Expression, Analysis, Design, Coding, Test and Validation. **Contraintes de composition :** - Une phase est composé de plusieurs itérations successives. - une itération est composé d'une sequence d'activités.	

Table 4.4 La description du style structurel UP.

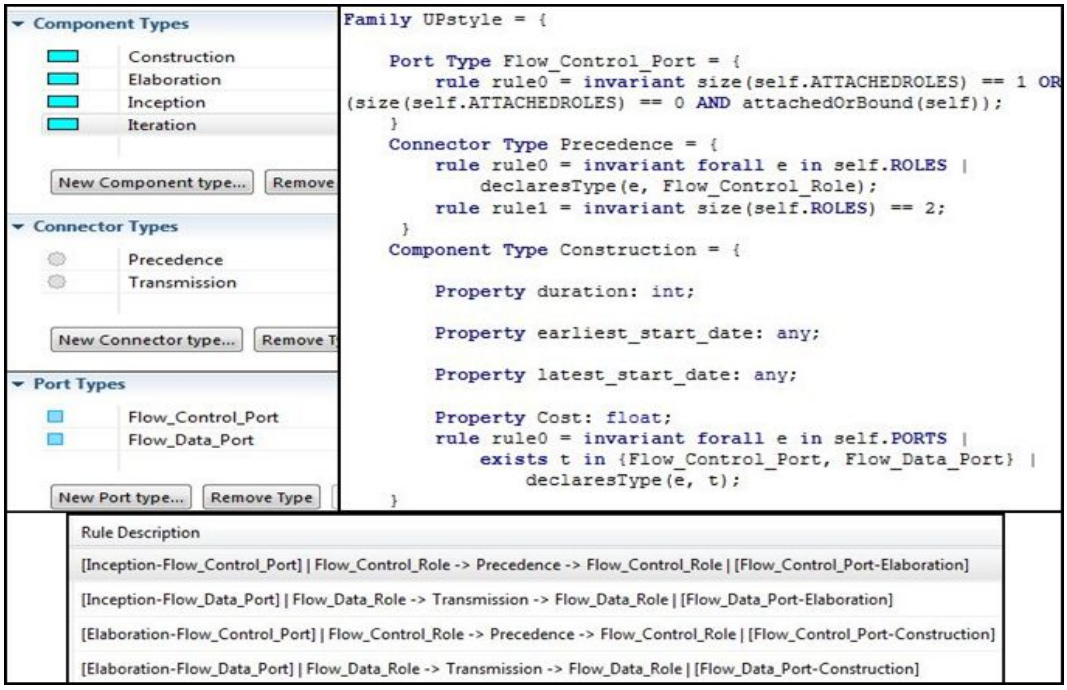


Figure 4.8 Le style structurel UP decrit avec l'ADL ACME.

4.3.4 La conception de l'ontologie des procédés logiciels

Afin de capitaliser les connaissances PLs qui sont réutilisables, notre solution se base sur la création d'une ontologie de domaine. Notre ontologie PL doit :

- Être cohérente, non ambiguë, et surtout, communément acceptée,
- Permettre la capitalisation des connaissances PLs à partir de modèles de PLs hétérogènes.

- Représenter explicitement les architectures de PLs, en incluant les concepts et les règles nécessaires à cet effet.
- Permettre d'extraire les connaissances nécessaires à la description puis au déploiement d'architectures de PLs.
- Gérer l'hétérogénéité et la cohérence des connaissances provenant de différentes sources.
- Extraire des connaissances compréhensibles par l'utilisateur final.
- Offrir la possibilité d'inférer de nouvelles connaissances et permettre le raisonnement et l'émergence de nouvelles solutions.

Pour collecter les concepts de notre ontologie de PLs, deux solutions ont été explorées :

- Étudier les ontologies de domaine spécifiques aux PLs et étudier la possibilité de leur réutilisation : l'étude des approches de réutilisation de PLs à base d'ontologies nous a permis de conclure que les ontologies proposées dans la littérature ne peuvent pas répondre à nos exigences, car chacune d'entre elles se limite à la modélisation d'un type particulier de PLs. De plus, elles ne traitent pas la capitalisation de connaissances à partir de sources hétérogènes, ce qui n'est pas notre cas.

- Exploiter des métamodèles existants regroupant les concepts PLs et étudier la possibilité de projection de ces concepts en concepts d'ontologie : la plupart des métamodèles de PLs existants sont spécifiques et représentent les concepts d'un environnement particulier ; construire une ontologie à partir de ces métamodèles n'est pas suffisant, l'ontologie sera spécifique à un type de PLs ce qui est contraire à nos aspirations.

Cependant, il existe un métamodèle qui répond à la plupart de nos exigences, c'est le métamodèle SPEM, ainsi :

- SPEM regroupe des concepts concernant un large ventail de PLs, il est général et ne se focalise pas sur un type de PLs particulier. L'utilisation de SPEM permet de collecter un maximum de concepts pertinents concernant le domaine des PLs.
- SPEM est un standard adopté par l'OMG et un métamodèle référence dans le domaine de la modélisation des PLs. Notre intérêt est d'exploiter une conceptualisation déjà adoptée et acceptée par la communauté de l'ingénierie des PLs.
- Des règles de correspondances entre différents métamodèles on été définies par l'OMG [93], dans le cadre de l'exploitation du métamodèle SPEM nous nous intéressons, entre autres, aux règles concernant la correspondance modèle UML/modèle OWL.
- SPEM est un profil UML, donc conforme au métamodèle UML2.0 [94] ; il est possible d'exploiter les différents outils, Frameworks développés dans le cadre de la MDA afin d'exploiter la conceptualisation de SPEM.
- A travers les sept packages de SPEM, plusieurs points de vue concernant les PLs sont offerts. Ainsi, il est possible de raisonner et d'inférer différents types de connaissances PLs qui permettent de répondre non seulement à nos préoccupations (réutilisation à large échelle des modèles de PLs), mais aussi, de répondre à d'autres préoccupations telles que les préoccupations des méthodes situationnelles [36], d'assistance ou documentation de PLs [38].

Ainsi, notre ontologie de domaine se base sur les concepts et la logique du métamodèle SPEM.

4.3.4.1 Les insuffisances de SPEM en concepts architecturaux

SPEM traite la réutilisation des PLs à base de composants à travers le profil "Method Plugin". Ce profil introduit la notion de "Composant procédé" et de réutilisation à base de composants à travers la définition de stéréotypes et de classes dédiés à cet effet (figure-4.9-).

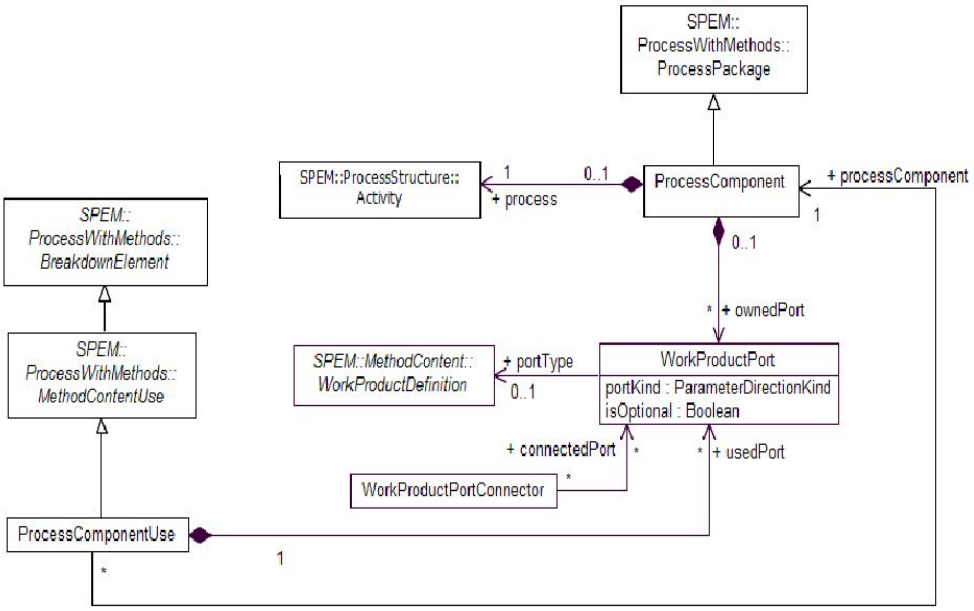


Figure 4.9 Les concepts architecturaux du métamodèle SPEM [88].

Pour la modélisation de PLs à base de composants, des difficultés d'assemblage de composants procédés ont déjà été constatées dans la spécification SPEM, les problèmes soulevés relèvent de :

- L'hétérogénéité de la terminologie utilisée pour les "Work Product Ports", car l'assemblage est fait manuellement en faisant une correspondance direct entre les "Work Product Ports" des composants à connecter [88].
- La difficulté de gestion des nombres de "Work Product Ports" des composants procédés à connecter, particulièrement pour les composants où le nombre de ports est différents [88].
- L'assemblage des modèles de PLs à base de composants est fait manuellement, ce qui augmente la dépendance du résultat aux connaissances et à l'expérience du développeur procédé.

Ces difficultés sont dues à l'absence de certains concepts architecturaux, ainsi, nous remarquons :

- L'absence de connecteurs prédéfinis ou explicites : le connecteur "Work Product Port Connector" est un connecteur implicite, c'est un simple "lien" entre des ports "Work Product Ports" (figure -4.9-). Il est défini pour décrire des liens de précédence ou de délégation sans distinction.
  Aucun mécanisme de facilitation ou d'adaptation d'assemblage n'est intégré ; son rôle se limite à assurer l'enchainement entre les composants procédés. Aussi, les autres propriétés décrivant le connecteur logiciel (sémantique, évolution, propriétés non fonctionnels ...) en tant que concept architectural ne sont pas prises en compte [80].
- L'absence du concept "Connector Role" : ce qui explique la connexion directe entre "Work Product Ports" à travers des "Work Product Port Conncetors".
- L'absence de contraintes d'assemblage : selon les cardinalités de SPEM, un connecteur peut connecter plusieurs ports sans aucune contrainte.
- L'absence d'abstraction architecturale : la notion de configuration procédé est absente dans SPEM. Les contraintes d'assemblage ne sont pas prises en compte, ce qui ouvre la porte à des modélisations non cohérentes. Le concept "Configuration Method" est défini comme une "classe", il

n'est pas considéré ni comme un élément architectural ni comme élément du modèle PL, c'est un ensemble logique d'éléments "Process Package" et d'éléments "Method Content" [88] sans contraintes d'assemblage.
- L'absence de styles architecturaux pour les PLs : la formalisation des structures récurrentes des PLs (telles que les cycles de vie de logiciel) pour la réutilisation à base de composants n'a pas été prise en compte.

4.3.4.2 L'extension du métamodèle SPEM en concept architecturaux manquants

L'absence des concepts architecturaux permettant de décrire les architectures de PLs selon la sémantique que nous avons définie, suggère l'extension du métamodèle SPEM. Nous étendons le Profil Method Plugin en introduisons les concepts identifiés précédemment. Deux stéréotypes abstraits sont définis [10] :
- "Process Architectural Element" qui décrit le comportement en commun des stéréotypes décrivant les constituants de la configuration PL (figure -4.10- à droite). Les stéréotypes "SP Component", "SP Connector" et "SP Configuration" décrivent les traitements effectués (création ou adaptation) pour produire de nouveaux flux de données."SP Port" et "SP Role Connector" sont considérés comme leur interface. "Attachement" et "Binding" sont les liens permettant l'assemblage des éléments architecturaux "SP Components" et "SP Connectors".
- "Method Content Architectural Element" qui décrit le comportement en commun des stéréotypes décrivant les constituants de "SP Style" (figure -4.10- à gauche). Puisque "SP Component" et "SP Connector" sont des "Activity", nous introduisons le stéréotype "Activity Definition" afin de décrire leurs types. "Default Activity Defintion Parameter" décrit la direction et les assemblages par défaut des "Activity Defintion".

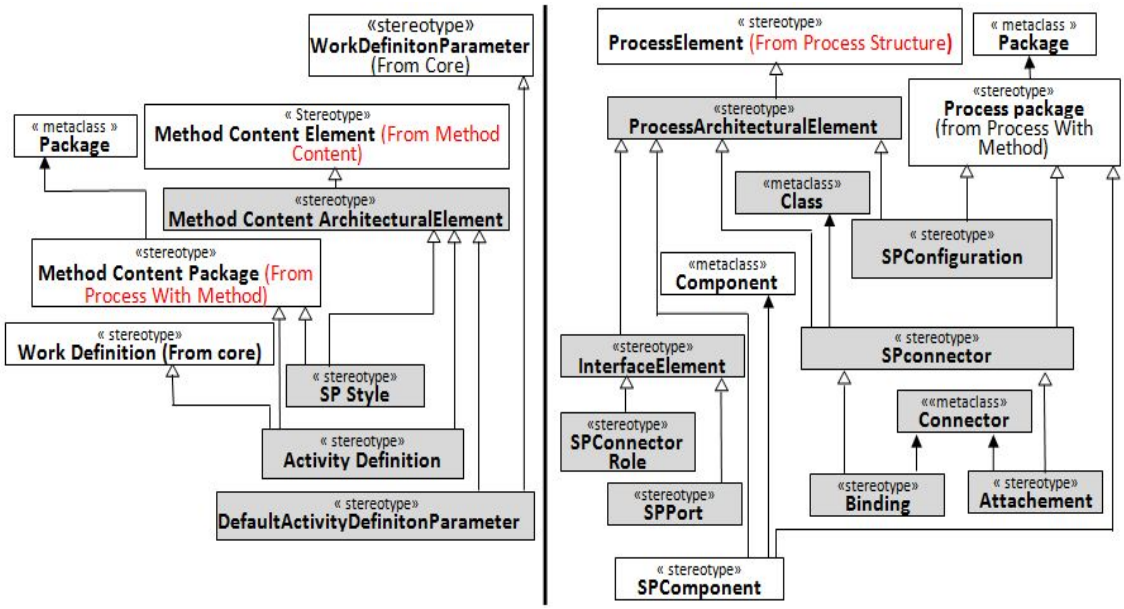


Figure 4.10 Extension du profil Method Plugin avec les concepts architecturaux manquants [10].

La figure-4.11- illustre les classes qui représentent les concepts architecturaux entrants dans l'extension du modèle SPEM . Ainsi,nous remarquons que "Activity Definition" joue le rôle du type de connecteur PL et du type du composant PL en même temps. Le composant PL et le connecteur PL

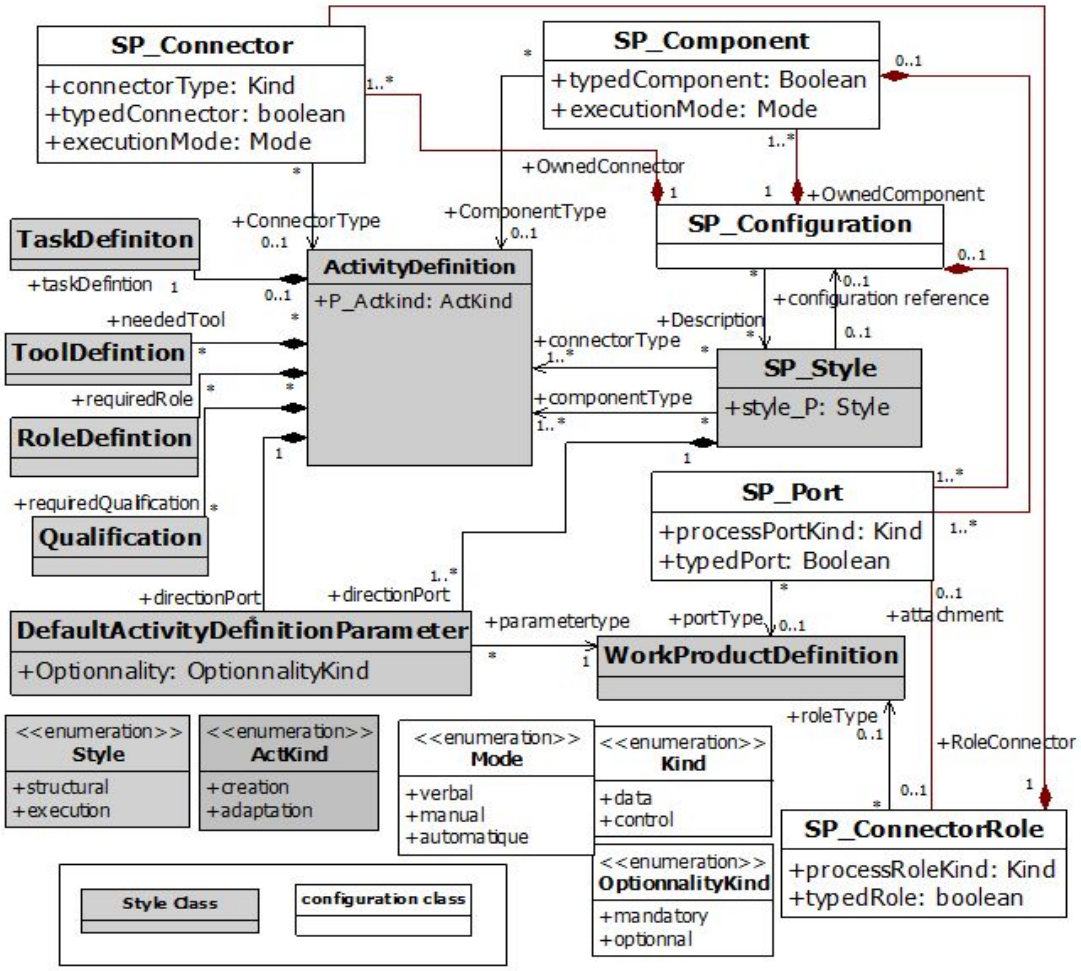


Figure 4.11 Extension du modèle SPEM avec les concepts architecturaux manquants.

sont des activités, en d'autres termes, des fragments de PLs composés des éléments de base d'un PL (Task Use, Role use, Tool Use et used qualification). Les types des composants PL et des connecteurs PL doivent exprimer tous ces concepts, ce qui explique que "Activity Definition" est composée de Task definition, Role Definition, Tool Definition et required qualification.

Nous remarquons aussi que "Work Product Definition" joue le rôle de type des ports PL. Ce rôle a été seulement repris partir de la spécification SPEM.

4.3.4.3 La capitalisation des connaissances procédés logiciels

L'acquisition des connaissances d'un domaine est toujours un défi. Quel que soit le domaine concerné, cette opération repose souvent sur des experts humains et sur un processus d'explicitation de ces connaissances. Le domaine des PLs n'échappe pas à la règle ; l'expert humain et le processus d'explicitation des connaissances sont essentiels pour l'acquisition et l'utilisation cohérentes des connaissances PLs.

L'objectif de cette étape est de capitaliser les connaissances à partir de différents modèles de PLs indépendamment de leur type, syntaxe ou sémantique. L'extraction de connaissances à partir de sources différentes crée de nombreux problèmes d'hétérogénéité et qui, bien évidemment, doivent être pris en

charge.

L'hétérogénéité au niveau concept ne pose pas de problème car la conceptualisation de notre ontologie respecte un métamodèle standardisé, accepté par la communauté. De plus, la plupart des modèles de PLs existants sont conformes au métamodèle SPEM, ou du moins, des correspondances entre concepts SPEM et concepts de l'ontologie peuvent être effectuées.

Cependant, le problème d'hétérogénéité se pose au niveau instance :

- Comment gérer l'hétérogénéité des termes utilisés pour décrire les connaissances PLs ? car plusieurs instances de sources hétérogènes peuvent décrire la même connaissance.
- Quel est le vocabulaire qui doit être utilisé pour décrire les connaissances résultats des recherches pour la modélisation du PL final ? : le résultat des recherches doit être compréhensible par l'utilisateur final du modèle de PL.
- Comment différencier les connaissances architectures de PLs des connaissances capitalisées des modèles de PLs existants ?
- Peut-on ajouter des règles d'inférence afin d'enrichir les connaissances de notre ontologie ?

Toutes ces questions trouvent réponses dans la structure de l'ontology qui respecte le métamodèle SPEM (figure-4.12-). En effet, grâce à la structure du métamodèle SPEM (organisée en plusieurs packages), nous pouvons faire la distinction entre :

- Les connaissances utilisées (Used Knowledge) : extraites à partir des modèles de PLs existants. Les concepts du métamodèle Process Structure et du métamodèle Process With Method sont utilisés pour stocker ces connaissances (figure-4.12-). Dans SPEM, ces métamodèles sont destinés à la description des utilisations effectives des méthodes de développement définies dans le métamodèle "Method Content".
- Les connaissances architecture de PLs (Architectural Elements) : après extension du métamodèle SPEM en concepts architecturaux, l'ontologie permet de capitaliser les connaissances des architectures de PLs.
- Le vocabulaire référence (Vocabulary Reference) : les connaissances du vocabulaire référence sont capitalisées par l'instanciation des concepts du métamodèle "Method Content". Dans SPEM ce métamodèle est dédié à la définition des méthodes de développement indépendamment de leur utilisation.
- La gestion de l'hétérogénéité des instances PLs est faite par l'exploitation des associations existantes entre les concepts du métamodèle "Method Content" avec les concepts des métamodèles "Process Structure" et le métamodèle "Process With Method". Ainsi, chaque connaissance utilisée aura son vocabulaire référence.
- L'enrichissement sémantique est possible en utilisant des règles d'inférence, ces règles permettront de préserver la cohérence des connaissances de l'ontologie et de combler les connaissances manquantes.

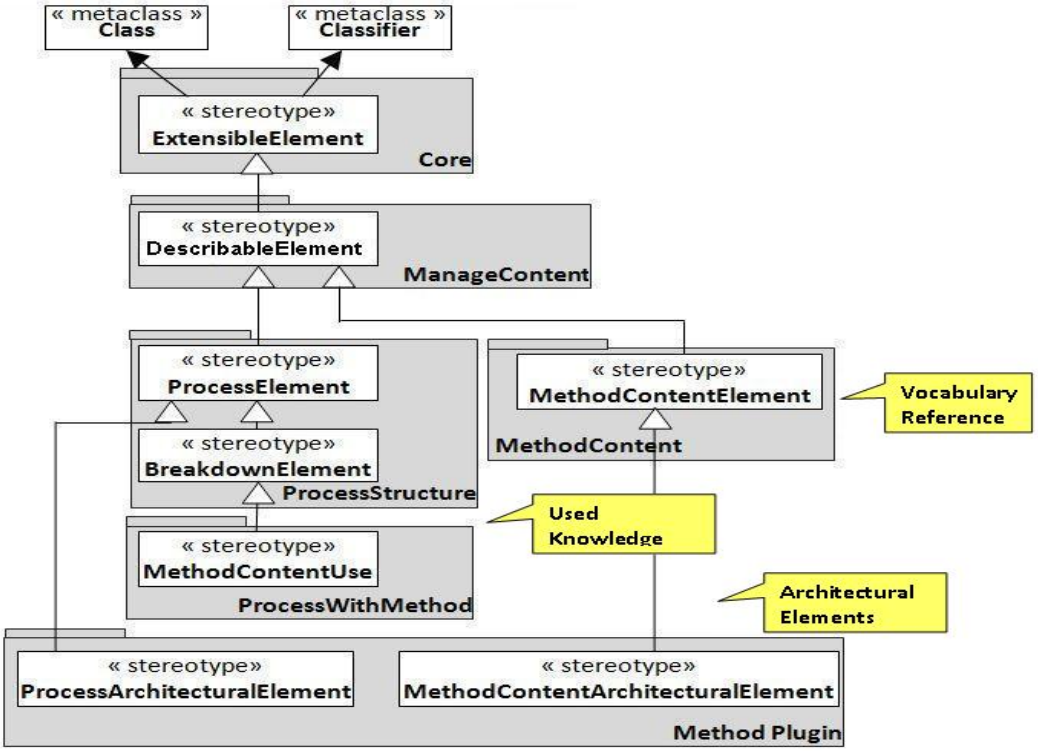


Figure 4.12 Organisation de la capitalisation des connaissances PLs.

4.3.4.4 L'enrichissement sémantique des connaissances PLs capitalisées

L'inférence est l'opération qui consiste à admettre une proposition en raison de son lien avec une proposition préalable tenue pour vraie. C'est un terme général dont les mots raisonnement, déduction, induction, etc., sont des cas spéciaux.

Dans notre approche, la définition de règles d'inférence est indispensable. En effet, la capitalisation de connaissances à partir de sources différentes nous donne l'opportunité de déduire de nouvelles connaissances qui peuvent être bénéfiques à la réutilisation des modèles de PLs. Dans notre approche, la création de nouvelles connaissances signifie la création de nouveaux enchainements entre des activités de sources différentes, ces nouveaux enchainements permettrons d'explorer de nouvelles solutions qui n'étaient pas possibles avant l'enrichissement de l'ontologie.

4.4 Ingénierie par la réutilisation des procédés logiciels

Après avoir conçu, étendu, généré, instancié et enrichi l'ontologie, l'étape suivante est la réutilisation effective des connaissances capitalisées.

L'ingénierie par la réutilisation, concerne l'utilisation réelle des connaissances capitalisées lors de la phase de l'ingénierie pour la réutilisation. Cette phase permettra de générer le modèle de PL opérationnel, elle passe par deux étapes principales (figure-4.13-) :

- La recherche et l'extraction de connaissances nécessaires pour la description de l'architecture de PL à partir des connaissances capitalisées.
- Le déploiement et génération du modèle de PL final.

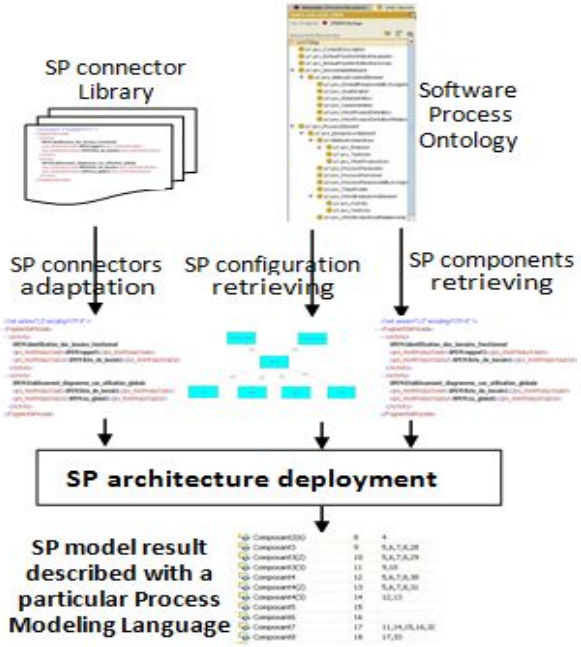


Figure 4.13 Les étapes ingénierie "pour" la réutilisation des PLs.

4.4.1 La recherche et acquisition des connaissances de l'architecture de PL

Cette étape permet de récupérer les connaissances nécessaires au déploiement de l'architecture de PL.

La figure-4.14- résume les étapes de recherche des connaissances de l'architecture PL. Ces étapes sont comme suit :

4.4.1.1 Le recherche et extraction de connaissances de la configuration PL

La recherche des connaissances de la configuration PL se fait à partir des concepts architecturaux rajoutés par l'extension du profil SPEM, deux types de recherches sont possibles :

- Recherche d'une configuration PL ad-hoc : les requêtes ont comme paramètres les produits en entrée et les produits en sortie de la configuration recherchée.
  Lors de la recherche, ces données correspondent aux ports PL de type Data Flow en entrée et en sortie de la configuration recherchée (Data Flow Ports In et Data Flow Ports Out).
- Recherche d'une configuration PL qui respecte un ou plusieurs styles PL : En plus des produits en entrée et des produits en sortie de la configuration recherchée, les requêtes ont comme paramètres le ou les styles que doit respecter la configuration résultat.

Les connaissances des architectures de PLs sont propres à l'utilisateur final et ne sont pas de sources hétérogènes, de ce fait, le problème de l'hétérogénéité de la terminologie utilisée ne se pose pas. De plus, les configurations saisies reflètent les stratégies de développement adoptées par l'entreprise, ainsi, nos requêtes donne la priorité à l'utilisation des configurations existantes avant de penser à créer de nouvelles configurations PL.

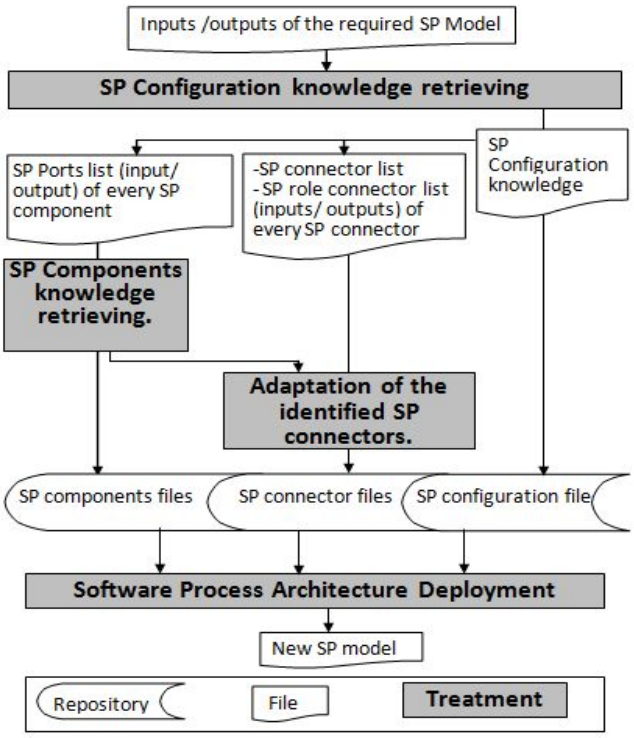


Figure 4.14 L'organigramme global suivi pour l'extraction des connaissances architecture de PL.

4.4.1.2 Le recherche et extraction des connaissances des composants PL

Afin de rechercher les connaissances des composants PL nécessaires au déploiement de l'architecture PL, nous procédons comme suit : pour chaque composant PL que l'on veut rechercher nous récupérons à partir de la configuration PL trouvées (résultat de l'étape précédente) la liste des Ports Data Flow In et Ports Data Flow Out (figure -4.14-).

La recherche de connaissances composant PL consiste à rechercher un enchainement d'activités qui a comme entrée les Ports Data Flow In et comme sortie les Ports Data Flow Out. Ces derniers correspondent au nom des produits en entrée et au nom des produits en sortie de l'enchainement d'activité recherché.

La recherche des connaissances composant PL se fait à partir des connaissances extraites des modèles de PLs existants (concepts de Process With Method et Process Structure packages), par conséquent, le problème de l'hétérogénéité des termes utilisés doit être pris en compte. Nous exploitons les correspondances (établies lors de l'instanciation) entre le "vocabulaire référence" et les "connaissances utilisées" afin d'expliciter le résultat de la recherche et rendre l'enchainement d'activité trouvé compréhensible par l'utilisateur final.

4.4.1.3 L'adaptation des connaissances des connecteurs PL

Les connaissances de chaque connecteur PL ne sont pas recherchées mais adaptées. Contrairement aux composants PL, les activités qui constituent les connecteurs PL sont connues. Que ce soit un des connecteurs de notre taxonomie ou un autre connecteur définit de manière indépendante, c'est toujours une activité d'adaptation connues par le développeur du modèle de PL. Les connaissances connecteur

PL sont adaptées en prenant en compte :

- Les rôles connecteur Data Flow (Data Flow Connector Role) qui sont récupérés à partir de la configuration PL. De la même manière que pour les ports Data Flow, les Role connecteur Data Flow correspondent aux produits en entrée et produits en sortie de l'activité constituant le connecteur.
- Les connaissances de l'activité (Role, Performer,...etc.) qui sont récupérées des connaissances des composants PLs qui précèdent et succèdent le connecteur PL.

4.4.2 Le déploiement de l'architecture de procédé logiciel

Le verbe "déployer" a pour définition d'étaler dans toute son extension une entité qui était pliée. Parmi ses synonymes, il est possible de trouver les verbes : étaler, déplier, ouvrir, développer et mettre en œuvre. Dans notre travail, déployer une architecture de PL revient à générer le code source du modèle de PL final en exploitant les connaissances récupérées dans la phase précédente.

Il existe un nombre important de modèles de PL, la question qui se pose est quel type de modèle de PL devons-nous ou pouvons-nous générer ?

L'idéal serait qu'un programme de déploiement puisse générer différents types de modèles de PLs à partir des mêmes connaissances de manière à augmenter la réutilisabilité des connaissances architectures de PL.

Cependant, le programme de génération du modèle de PL dépend de la syntaxe du langage de modélisation de PL, et par conséquent, un programme de déploiement ne peut générer qu'un type de PL écrit dans un langage de modélisation de PL. Ainsi, chaque langage de modélisation de PL doit avoir son propre programme de génération de modèles de PLs.

Néanmoins, les connaissances d'architecture de PL sont indépendantes des programmes de génération de codes, ce qui signifie que les mêmes connaissances architecture de PL peuvent être utilisées par différents programmes de déploiement et générer différents types de modèles de PLs ce qui augmente leur réutilisabilité.

Type de déploiement	**Configuration PL est déployée ...**	**Peut être utilisé dans des modèles qui sont...**
statique	Globalement, en une seule fois.	Stable, homogène.
Dynamique (centré humain)	composant par composant, l'humain décide quel composant déployer.	Dynamiques où seulement l'humain qui décide de la progression de l'exécution.
Dynamique (centré procédé)	composant par composant, le PL décide quel composant déployer .	Dynamiques où l'humain valide la décision de progression.
Incrémental	Incrément par incrément.	Les méthodes Agiles.
Itératif	Itération par itération.	Les méthodes Agiles.
Hétérogènes	En utilisant différents PMLs.	Hétérogènes.

Table 4.5 Les types de déploiement offerts par l'approche AoSP.

Le tableau-4.5- résume les types de déploiement mis en place pour implémenter différents types d'exécutions. En général, deux catégories de déploiement sont possibles, le déploiement total et les déploiements partiels.

Les déploiements partiels permettent de modéliser les différentes exécutions dynamiques des modèles de PLs ; elles se basent souvent sur les résultats des connecteurs Control Flow (Evaluation ou

Decision).

Il est possible de combiner ces types de déploiement pour avoir des exécutions particulières qui conviennent aux besoins des utilisateurs.

4.5 Évaluation de l'approche AoSP

4.5.1 Évaluation de l'approche AoSP selon l'axe procédé logiciel

Le tableau-4.6- résume l'évaluation de l'approche AoSP selon l'axe PL.

Notre approche a comme concepts de base les concepts : "Produit" et "Activité", ces concepts sont indispensables car toute opération d'instanciation, de recherche ou de déploiement exploite ces deux concepts. Aussi, aucun concept additionnels n'est indispensable pour assurer la réutilisation des PLs.

D'une autre côté, contrairement aux approches étudiées, notre approche n'est pas spécifique à un type de PLs, les concepts manipulés, l'abstraction, l'hétérogénéité et les langages de modélisation de PLs dépendent du programme de déploiement ; tout type de modèle de PL peut être généré à condition d'avoir le programme de déploiement adéquat.

Aussi, contrairement aux approches existantes, le contrôle des interactions humaines ont eu un traitement particulier ; la définition de connecteurs PL qui décrivent les interactions humaines est un atout important dans notre approche.

Approche	**Éléments PLs**	**Type expression**	**PML**	**Hétérogénéité**	**Dimension humaine**
L'approche AoSP	**De base :** Activité, Produit **Additionnels :** Dépendent du PML	Formel Semi formel	Dépend du déploiement		**Assistance :** Représentation graphique, recherche de solutions, déploiement. **Contrôle d'interaction :** connecteurs centrés humain

Table 4.6 Évaluation de l'approche AoSP selon l'axe PL.

4.5.2 Évaluation de l'approche AoSP selon l'axe architecture logicielle

Le tableau-4.7- résume l'évaluation de l'approche AoSP selon l'axe AL. Contrairement aux approches étudiées, les éléments architecturaux sont tous explicites, y compris le style. En effet, différents styles spécifiques aux architectures de PLs ont été identifiés, ce qui augmente la réutilisation des architectures de PLs.

Aussi, comme langage de description d'architectures de PLs nous avons utilisé un ADL existant et qui est reconnu comme une référence dans le domaine des ALs. L'autre avantage de notre approche est qu'elle permet différents types de déploiements d'architectures de PLs, ce qui, non seulement facilite la modélisation du PLs, mais aussi, augmente la réutilisation des architectures de PLs.

Comme espace de stockage, nous avons opté pour l'utilisation d'une ontologie de domaine, l'intérêt et d'exploiter au mieux les connaissances capitalisées, et cela par l'inférence de nouvelles connaissances, ce qui n'est pas le cas des approches existantes.

La portée de réutilisation des éléments architecturaux dépend de la solution technique adoptée pour la description des connecteurs PL et des composants PL. Pour cela, nous avons opté pour :

– La description des connaissances composant PL et connecteur PL à l'aide du langage XML, ce qui leur permettra d'avoir un aspect standard, portable et réutilisable.
– la définition de balises XML conformes au métamodèle SPEM, ce qui conférera aux connaissances composants PL et connecteurs PL un aspect standard.

Approche	**Concepts AL réutilisés**				**Langage d'AL**	**Aspects de réutilisation**
	Com.	Conn.	Conf.	Style		
L'approche AoSP	**Explicite**				ACME	**Mécanismes :**Instanciation, Composition. **Espace de stockage :** Ontologie de domaine, **portée :** externe, **Déploiement :** Différents types.

Table 4.7 Évaluation de l'approche AoSP selon l'axe AL.

4.5.3 Évaluation de l'approche AoSP selon l'axe qualité

Le tableau-4.8- résume l'évaluation qualitative de l'approche AoSP. Nous pouvons remarquer que notre approche exploite les spécificités des ALs pour augmenter la qualité du modèle de PLs. Ainsi :
– La configuration et sa représentation graphique sont exploitées pour visualiser puis faciliter la compréhension du modèles de PLs lors de la modélisation et de l'exécution du PLs.
– Les styles de PLs sont exploités pour faciliter la modélisation des PLs et pour bénéficier d'un savoir-faire acquis et formalisé.
– Les connecteurs PLs sont exploités pour faciliter le contrôle d'exécution du modèle de PL.
Le mécanisme de déploiement est utilisé pour matérialiser les différents évolutions et adaptations (prévues, non prévues, statiques, dynamiques...) du modèle de PL.

Tous ces aspects n'ont pas été exploités par les approches étudiées ce qui rend notre approche plus efficace pour la modélisation de PLs de qualité.
De plus, l'utilisation d'une ontologie de domaine et d'algorithmes de recherche et d'extraction de connaissances, augmente cette efficacité car, cela nous permet d'exploiter des solutions optimales pour la modélisation des PLs.

Approche	**Qualité de la modélisation**	**Mode d'exécution**	**Contrôle d'exé.**	**Évolution**
L'approche AoSP	**Pertinence :** Recherche des meilleures solutions, **Compréhension :** exploitation de configuration, **facilité de modélisation :** définition de styles PLs	Dynamique, Hétérogène, Distribué	Différents connecteurs explicites	**Statique :** mécanisme de recherche de composants., **Dynamique :** mécanisme de déploiement partiel, incrémental ou itératif.

Table 4.8 Évaluation de l'approche AoSP selon l'axe qualité.

4.6 Conclusion

Dans ce chapitre nous avons présenté notre approche de réutilisation de PLs à base d'ALs.
l'approche AoSP est proposée pour pallier la faible réutilisation des modèles de PLs, elle se base sur les insuffisances des approches existantes pour proposer une solution plus optimale.
Notre approche a pour objectif de proposer une solution facile et économique pour la modélisation de PL de qualité : Facile, dans le sens où l'intervention d'experts PLs est minimisée. Économique telle que nous réutilisons tous les modèles de PLs existants.
Les modèles de PLs sont de qualité, dans le sens où ils répondent aux critères de qualité exigés par le monde du développement logiciel.
La solution de réutilisation conçue couvre les deux cycles de réutilisation : "pour" et "par" la réutilisation de PLs ; elle combine deux domaines de recherche caractérisés par l'importance et la prise en charge de la réutilisation : les architectures logicielles et les ontologies de domaine.
L'exploitation des Als a permis d'augmenter la compréhension et la facilité de modélisation des PLs à travers l'exploitation de la configuration PLs lors de la modélisation. Elle a permis aussi de combiner l'agilité et le contrôle d'exécution, en réservant un traitement particulier aux interactions des PLs ; des connecteurs explicites ont été définis, ces derniers assurent les traitements adéquats afin de prendre les décisions pertinentes de progression de l'exécution du modèle de PL.
Dans le chapitre suivant, nous présentons l'aspect technique de l'approche AoSP ; nous détaillons les solutions adoptées et nous illustrons son fonctionnement par des exemples et des cas d'études.

CHAPITRE 5
Implémentations et expérimentations

5.1 Introduction

Au cours du chapitre précédent, nous avons présenté notre approche de réutilisation de PLs à base d'ALs. Il nous reste à proposer notre réalisation et à mener des expérimentations autour de nos propositions. Par conséquent, dans ce chapitre, nous présentons en premier lieu les étapes que nous avons suivies pour réaliser notre prototype. En prenant en compte la structure de l'approche, ce chapitre est organisé en deux parties : une partie est consacrée à l'implémentation de l'ingénierie "pour" la réutilisation des PLs où nous détaillons les étapes suivies pour générer et peupler notre ontologie de domaine. La deuxième partie est consacrée à l'ingénierie "par" la réutilisation des PLs où nous détaillons l'implémentation qui concerne la description puis de déploiement des architectures de PLs.
Chaque étape de réalisation est accompagnée de cas d'études et d'exemples qui illustrent le fonctionnement de l'application réalisée. Nous terminons ce chapitre par une conclusion qui résume le travail réalisé et énonce les améliorations qui peuvent être apportées au résultat.

5.2 Description des étapes de l'implémentation de l'approche AoSP

La figure-5.1- illustre les étapes suivies pour réaliser notre prototype, nous résumons ces étapes comme suit :

- **Génération de notre ontologie de domaine (SPEMOntology generation) :** SPEMOntology est générée par des techniques de transformation de modèles. A cet effet, nous utilisons le langage de transformation de modèles ATL (Atlas Transformation Modèles) [12].
- **Capitalisation des connaissances PLs (SP knowledge capitalization) :** cette étape permet d'extraire les connaissances pertinentes à partir des modèles de PLs existants. Comme cas d'étude, nous présentons l'extraction des connaissances à partir des modèles PBOOL [39] et à partir des modèles EPF [44].
- **Inférence de connaissances PL (SP knowledge inference) :** la capitalisation de connaissances de sources hétérogènes peut créer des problèmes de cohérence et de consistance des connaissances capitalisés. Des "vides" peuvent être crées par absence de certaines connaissances PL et doivent être gérés.
  Par ailleurs, la déduction de nouvelles solutions est matérialisée par la déduction de nouveaux enchainements entre des activités de sources différentes.
  Des règles d'inférence doivent être mises en place pour, d'une part, combler les connaissances manquantes, et d'autre part, déduire de nouveaux enchainements.

Ces étapes constituent la phase d'ingénierie pour la réutilisation des PLs. Malgré la difficulté de réalisation de ces étapes et la sollicitation des experts PLs, leur avantage est qu'elles sont faites une seule fois. Les mises à jour des connaissances de l'ontologie ne sont pas fréquentes, elles sont faites que dans le cas où nous introduisons de nouveaux modèles de PLs candidats à la réutilisation.

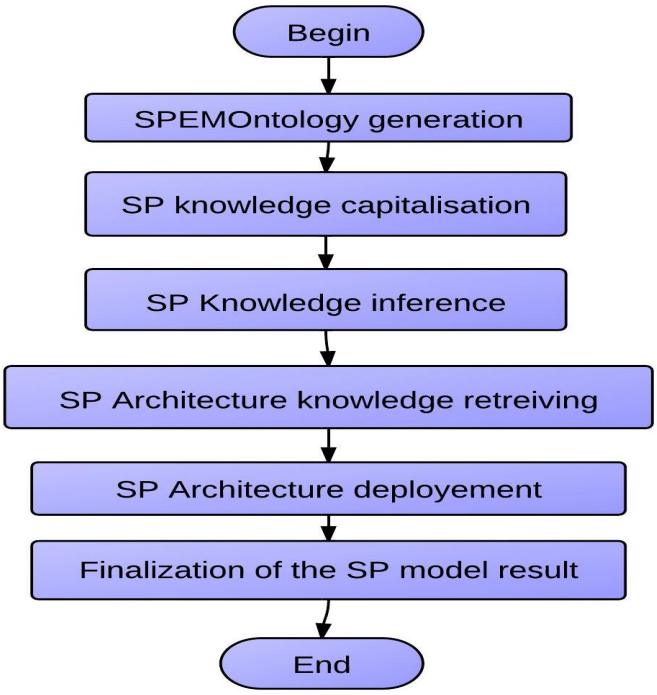


Figure 5.1 Les étapes de réalisation de l'approche AoSP.

- **Extraction de connaissances architecture PL (SP architecture knowledge retreiving) :** cette étape consiste à rechercher et extraire les connaissances de la configuration PL et celles des composants PL et des connecteurs PL correspondants. L'extraction est faite à partir des connaissances capitalisées lors de l'étape précédente. Plusieurs algorithmes de recherche et de parcours ont été implémentés à cet effet.
- **Déploiement de l'architecture de PL (SP architecture deployment) :** cette étape permet de générer le modèle de PL final à partir des connaissances architecture de PL extraite lors de l'étape précédente.
- **Finalisation du résultat (Finalization of the SP model result) :** cette étape permet de combler les informations manquantes telles que l'affectation des développeurs aux activités et la vérification de la cohérence du résultat.

Ces étapes constituent la phase de l'ingénierie par la réutilisation des PLs, ces dernières sont en majorité automatiques et sollicitent rarement le savoir-faire d'experts de PLs.

5.3 Ingénierie pour la réutilisation des PLs : Préparer l'ontologie de domaine.

5.3.1 La génération de SPEMOntology

Dans notre travail, nous générons automatiquement notre ontologie "SPEMOntology" à partir du métamodèle SPEM étendu, pour cela, nous utilisons le langage de transformation de modèles ATL [12]. ATL (Atlas Transformation Language) est un langage de transformation de modèles basé sur le langage de contraintes OCL (Object Constaints Language) [87]. ATL est proposé par l'OMG, il est défini pour effectuer des transformations de modèles dans le cadre de la MDA (Model Driven Architecture).
Une transformation ATL est composée de modules ATL, pour la génération de SPEMOntology nous utilisons trois modules de transformation de modèles : UML2OWL [13] OWL2XML [13] et UML2Copy [13]. Ces modules existent déjà, ils sont opérationnels et peuvent être exploités pour générer une ontologie OWL à partir d'un modèle UML.
UML2OWL et OWL2XML sont des modules de la transformation UML2OWL, ils fournissent les règles permettant la transformation d'un modèle UML en un modèle OWL. Cependant, SPEM est un profil UML, de ce fait, cette transformation n'est pas suffisante, car elle ne permet pas la transformation d'un modèle UML "stéréotypé" conforme à un profil UML en un modèle OWL. En effet, le module UML2OWL ne possède pas de règles qui permettent de transformer des profils, stéréotypes et Tagged Values en éléments UML.
Afin de remédier à ce problème, nous introduisons une transformation ATL (ATL1) qui permet d'appliquer les stéréotypes aux classes UML correspondantes de manière à avoir des classes stéréotypées avec des "Tagged Values" et des "contraintes prédéfinies".

5.3.1.1 Les étapes de la transformation du métamodèle SPEM en une ontologie de domaine "SPEMOntology".

La transformation du métamodèle SPEM en une ontologie de domaine (SPEMOntology) est effectuée par deux transformations : ATL1 et ATL2 (Figure-5.2-) :
ATL1 est une transformation utilisée pour générer le modèle SPEM stéréotypé conforme au profil SPEM. Cette transformation applique le profil SPEM au modèle SPEM pour avoir le métamodèle SPEM (le modèle SPEM stéréotypé). ATL1 est composée de deux modules :
- Le module UML2Copy : copie les éléments du modèle SPEM auxquels aucun stéréotype du profil SPEM n'est appliqué.
- Le module ApplySPEMProfil2SPEMModel : applique les stéréotypes du profil SPEM aux éléments du modèle SPEM. Contrairement aux autres modules, ce module n'est pas réutilisable car il est spécifique au profil SPEM et au modèle SPEM.

La transformation ATL2 consiste à générer SPEMOntology, (une ontologie OWL) à partir du modèle SPEM stéréotypé qui est généré par la transformation ATL1. Elle est composée de deux modules ATL :
- Le module UML2OWL qui est la transformation principale, elle prend en entrée un modèle UML et elle produit en sortie une ontologie OWL conforme au métamodèle OWL défini par l'ODM (Ontology Definition Metamodel) [93].
- Le deuxième module OWL2XML est un extracteur OWL XML qui traduit l'ontologie générée par le premier module en XML suivi d'une projection pour générer un document XML conforme

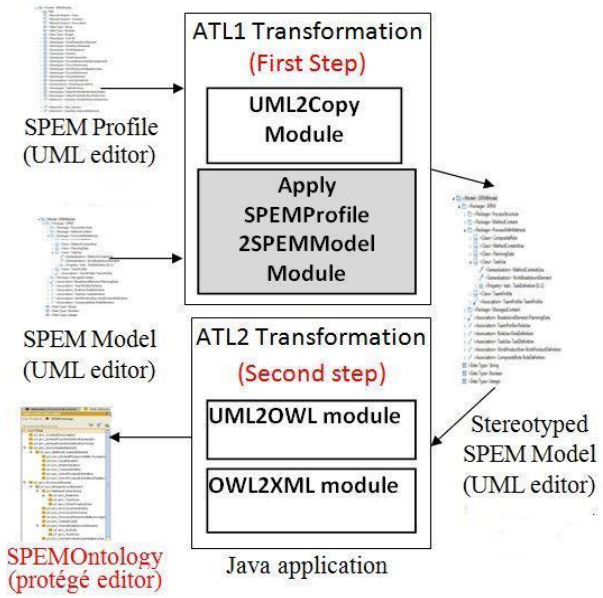


Figure 5.2 Les transformations ATLs appliquées pour la génération de SPEMOntology.

à la syntaxe OWL définie par la spécification W3C.

Ces deux modules ainsi que le module UML2copy ne sont pas détaillés car ces des modules existent déjà et le détail de ces transformations peut être consultés dans [13].

Le résultat des deux transformations est une ontologie OWL qui est opérationnelle et qui va être utilisée pour capitaliser les connaissances PLs.

5.3.2 Le module ATL ApplySPEMProfil2SPEMModel

Le module ApplySPEMProfil2SPEMModel a deux modèles en entrée (SPEM Model and SPEM profil) et génère en sortie le modèle SPEM stéréotypé qui représentera le métamodèle SPEM. Tous ces modèles sont conformes au métamodèle UML 2.0 (figure-5.3-).

Une transformation ATL est constituée de plusieurs modules. Tout module ATL est constitué d'un entête, de helpers et de règles (les CalledRules, les MatchedRules et les règles hybrides) [12]. Le module ApplySPEMProfil2SPEMModel est constitué de :

- **L'entête :** indique le nom de la transformation qui est dans ce cas ApplySPEMProfil2SPEMModel. Elle définit aussi les noms des variables des modèles sources et cibles et spécifie leurs métamodèles. Les modèles sources sont introduits par le mot-clé "from", c'est le modèle SPEM (variable IN) qui suit le métamodèle UML, et le profil SPEM (variable PRO) qui est une spécialisation du métamodèle UML. Le mot-clé "create" introduit le modèle en sortie qui est le résultat de l'application du profil SPEM au modèle SPEM (variable OUT) (figure -5.4-).
- **Les helpers :** renvoient les résultats qui vont être utilisés dans les règles de transformation. Les helpers de la figure -5.4- ont été définis pour déclarer des variables globales dont chacune désignera un des packages du modèle SPEM. Dans notre travail nous avons fait la transformation de six packages parmi les sept packages de SPEM. Nous n'avons pas exploité le package "Process Behavior" du métamodèle SPEM, car nous n'avons pas besoins de ses caractéristiques pour la capitalisation des nos connaissances PLs.
- **Les règles :** Les règles de ce module permettent d'appliquer le profil SPEM au modèle SPEM

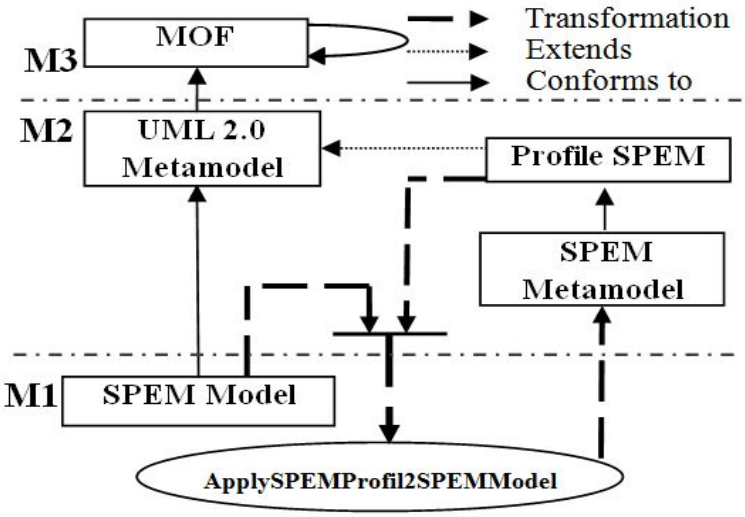


Figure 5.3 Le module de transformation ATL ApplySPEMProfil2SPEMModel.

```
module ApplySPEMProfile2SPEMModel; -- The Header
create OUT : UML2 from IN : UML2, PRO : UML2;

----------------------LES HELPERS---------------------------------

helper def: Global_PackageSPEM : UML2!Package = ' ';
helper def: Global_PackageProcessStructure : UML2!Package = ' ';
helper def: Global_PackageManagedContent : UML2!Package = ' ';
helper def: Global_PackageMethodContent : UML2!Package = ' ';
helper def: Global_PackageProcessWithMethods : UML2!Package = ' ';
helper def: Global_PackageMethodPlugin : UML2!Package = ' ';

helper def: allTypes : Sequence(UML2!DataType) = UML2!DataType.allInstancesFrom('IN');
```

Figure 5.4 Entête et helpers du module ApplySPEMProfil2SPEMModel.

pour générer en sortie le modèle SPEM dont les éléments sont stéréotypés. Chaque élément du modèle SPEM est associé à une matchedRule qui permet d'appliquer le stéréotype correspondant à l'élément. L'élément UML peut être : une classe ou une association.

5.3.2.1 Les étapes d'exécution du module ApplySPEMProfil2SPEMModel.

Les étapes de l'application du profil SPEM au modèle SPEM sont :

- **Définition du profil SPEM et du modèle SPEM :** A l'aide d'un outil UML dans notre cas nous utilisons le Plugin UML2.0. pour java (Eclipse) (figure-5.5-).
- **Application du profil SPEM au modèle SPEM :** cette étape est effectuée en utilisant la règle "ApplyProfil2Model" (figure -5.6-). Cette règle est une matchedRule exécutée quand une instance de l'élément UML2 !Model est reconnue dans le modèle source. Elle permet de copier le

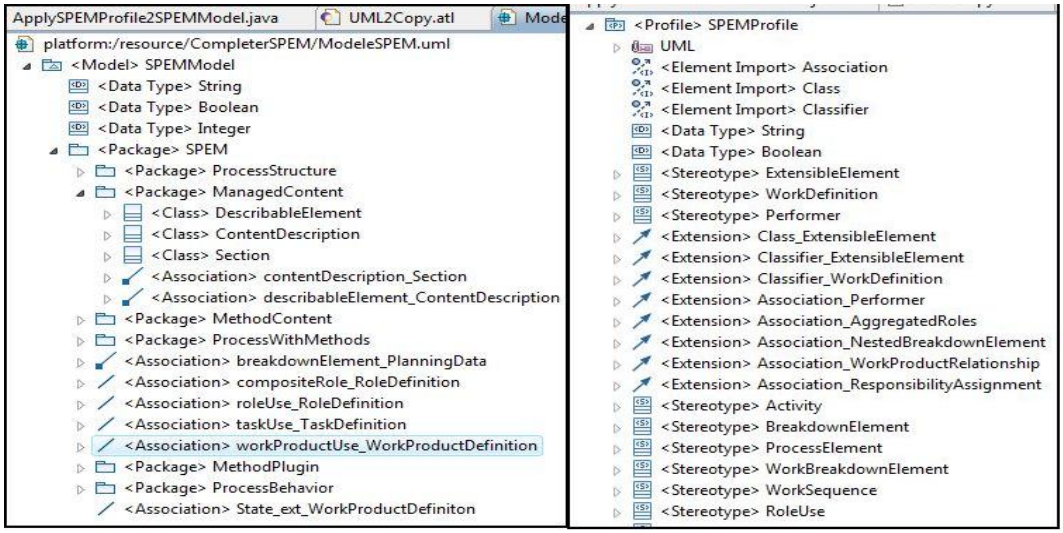


Figure 5.5 Les modèles sources du module ApplySPEMProfil2SPEMModel décrit en UML.

modèle SPEM en copiant toutes ses caractéristiques : son nom, sa visibilité, et les éléments qu'il comporte. La règle comporte une partie impérative introduite par le mot clé "do" dans laquelle le profil SPEM est appliqué au modèle SPEM en utilisant la méthode *applyProfil()*.

```
rule Model {
    from IN : UML2!"uml::Model" (thisModule.inElements->includes(IN))
    to OUT : UML2!"uml::Model" (
        name <- IN.name,
        visibility <- IN.visibility,
        viewpoint <- IN.viewpoint,
        eAnnotations <- IN.eAnnotations,
        ownedComment <- IN.ownedComment,
        clientDependency <- IN.clientDependency,
        nameExpression <- IN.nameExpression,
        elementImport <- IN.elementImport,
        packageImport <- IN.packageImport,
        ownedRule <- IN.ownedRule,
        templateParameter <- IN.templateParameter,
        templateBinding <- IN.templateBinding,
        ownedTemplateSignature <- IN.ownedTemplateSignature,
        packageMerge <- IN.packageMerge,
        packagedElement <- IN.packagedElement,
        profileApplication <- IN.profileApplication)

        do {

                OUT.applyProfile(UML2!Profile.allInstances()->select(e | e.name = 'SPEMProfile').first());
```

Figure 5.6 La méthode *applyProfil()* appliquée dans le module ApplySPEMProfil2SPEMModel.

– **Application des stéréotypes du profil SPEM aux éléments du modèle SPEM (classes et associations).**
Cette étape est effectuée en appliquant la méthode *applyStereotype()*. Pour chacun de ces éléments nous associons une matchedRule qui permet de : copier l'élément et ses propriétés telles que visibilité,...etc. Quant à son nom il est précédé du préfixe "pro" pour distinguer au niveau de l'ontologie les éléments auxquelles un stéréotype du profil SPEM a été appliqué des éléments auxquelles nous n'avons pas appliqué de stéréotypes.
Dans la partie impérative de la règle, l'élément généré est placé dans la copie du package adéquat

puis un stéréotype du profil SPEM lui est appliqué en utilisant la méthode *applyStereotype()*.

```
rule ActivityClass {
from s:UML2!Class (not s.oclIsKindOf(UML2!Stereotype)  and (s.name='Activity'))
to t:UML2!Class (
        name <- 'pro_'.concat(s.name),
        visibility <- s.visibility,
        isLeaf <- s.isLeaf,
        isAbstract <- s.isAbstract,
        isActive <- s.isActive,
        eAnnotations <- s.eAnnotations,
```

```
do {
            --add the class to the package process Structure
            thisModule.Global_PackageProcessStructure.packagedElement <- thisModule.Global_PackageProcessStr

            --Apply the streotype
            t.applyStereotype(UML2!Stereotype.allInstances()->select(st | st.name = 'Activity').first());

            --add the tagged values

            t.ownedAttribute <- t.ownedAttribute ->including(p1);
            t.ownedAttribute <- t.ownedAttribute ->including(p2);
            t.ownedAttribute <- t.ownedAttribute ->including(p3);
```

Figure 5.7 La méthode *applyStereotype()* appliquée dans le module ApplySPEMProfil2SPEMModel.

Les méthodes *applyProfil()* et *applyStereotype()* sont fournies par le plugin UML2.0, ce plugin est indispensable à l'exécution de ces transformations car il fourni aussi le métamodèle UML2.0 en Ecore, auquel les modèles UML (le modèle SPEM, le profil SPEM et le modèle stéréotypé SPEM) sont conformes.

– **Initialisation des Tagged Values des éléments stéréotypés :** à ce niveau les Tagged Values n'ont pas été traitées. Nous détaillons la solution adoptée à leur transformation dans la section suivante.

5.3.2.2 La démarche suivie pour la gestion des Tagged Values

Pour pouvoir manipuler les Tagged Values dans l'ontologie, nous avons constaté que les Tagged Values ne peuvent être que des Data Type Property au niveau de l'ontologie, car un Data Type Property relie un individu d'un concept OWL à un Data Type OWL, ce qui est le cas d'une Tagged Value qui relie une instance UML à un Data Type UML.
Un Data Type Property dans une ontologie OWL est généré à partir d'un attribut d'une classe UML. Par conséquent, nous créons un attribut pour chaque Tagged Value dans la classe stéréotypé. Un stéréotype qui comporte des Tag Definitions, une fois appliqué à une classe, ces dernières (les Tag Definitions) correspondent à des Tagged Values dans la classe. Ainsi, pour chaque classe à laquelle son stéréotype comporte des Tag Definitions, nous créons un attribut pour chaque Tagged Value et nous le rajoutons aux attributs de la classe.
Nous réalisons ceci de la manière suivante :

– Nous associons une matchedRule (figure -5.8-) qui comporte une partie impérative à toute classe dont un stéréotype portant des Tag Definitions lui sera appliqué. La règle génère une copie de la classe et un attribut UML pour chaque Tag Definition du stéréotype.
– Pour chaque attribut UML généré, nous spécifions son nom, son type, sa multiplicité et sa visibilité (figure-5.8-). Dans la partie impérative de la règle, les attributs UML générés sont ajoutés à l'ensemble des attributs de la classe.

Lors de la création de l'attribut UML à partir d'une Tag Definition, le nom, le type, la multiplicité et la visibilité de chaque attribut UML généré sont spécifiés comme suit :

– **Nom :** son nom est le nom du Tag Definition correspondant.

– **Type :** son type est le type du Tag Definition correspondant, qui peut être un UML Data Type (Boolean, integer, string...etc.) ou peut prendre les valeurs d'une Enumeration.

  Dans le cas ou le Tag Definition prend les valeurs d'une Enumeration, le type de l'UML Property généré sera le UML Data Type "String". Et au niveau de l'ontologie lors de l'instanciation, la valeur de l'élément correspondant à cette Property doit être un des littéraux de cette énumération qui ne sont rien d'autre que des chaines de caractères.

– **Visibilité :** la visibilité de toutes des attributs UML dans le modèle SPEM est "Public", donc tout attribut généré aura une visibilité "Public".

– **Multiplicité :** la multiplicité d'un attribut indique le nombre de valeurs qui peuvent être associées à ce dernier. La valeur minimale est lower, et la valeur maximale est upper. Dans notre cas le nombre de valeurs maximum qu'on associe à un attribut est au plus "un". Ainsi, lower et upper de tout attribut générée valent respectivement "zéro" et "un".

```
p2: UML2!Property(
        name <- 'postCondition',
        type  <- thisModule.allTypes -> select(e|e.name = 'String')->at(1),
        lower <- UML2!Class.allInstancesFrom('IN')->select(e|e.name='PlanningData')->first().ownedAttribute->first().lower,
        upper <- UML2!Class.allInstancesFrom('IN')->select(e|e.name='PlanningData')->first().ownedAttribute->first().upper,
        visibility <- UML2!Class.allInstancesFrom('IN')->select(e|e.name='PlanningData')->first().ownedAttribute->first().visibility
),
```

Figure 5.8 Exemples d'instructions ATLs qui permettent d'intégrer les Tagged Values .

5.3.3 La structure de l'ontologie SPEMOntology

SPEMOntology est le résultat de deux transformations ATL successives. Elle est constituée de 56 concepts et un nombre important de Data Types properties et d'Object Type properties. Dans le but de faciliter sa compréhension, il est important de décrire son organisation. SPEM est structuré en sept packages [88], en analysant les packages de SPEM (après extension), nous constatons que chaque package a sa propre classe abstraite qui décrit le comportement commun du reste des classes du package.
La figure -5.9 (à gauche)- décrit les principales classes abstraites du métamodèle SPEM. Nous constatons, l'existence des classes rajoutées lors de l'extension. Toutes les classes UML de SPEM (abstraites ou pas) sont transformées en concepts OWL. Après l'application des transformations ATLs, nous constatons que la structure de SPEMOntlogy (figure-5.9 (a droite)-) respecte la structure du métamodèle SPEM ; la vue "package" est identifiée à travers les concepts OWL générés à partir des classes abstraites de SPEM.

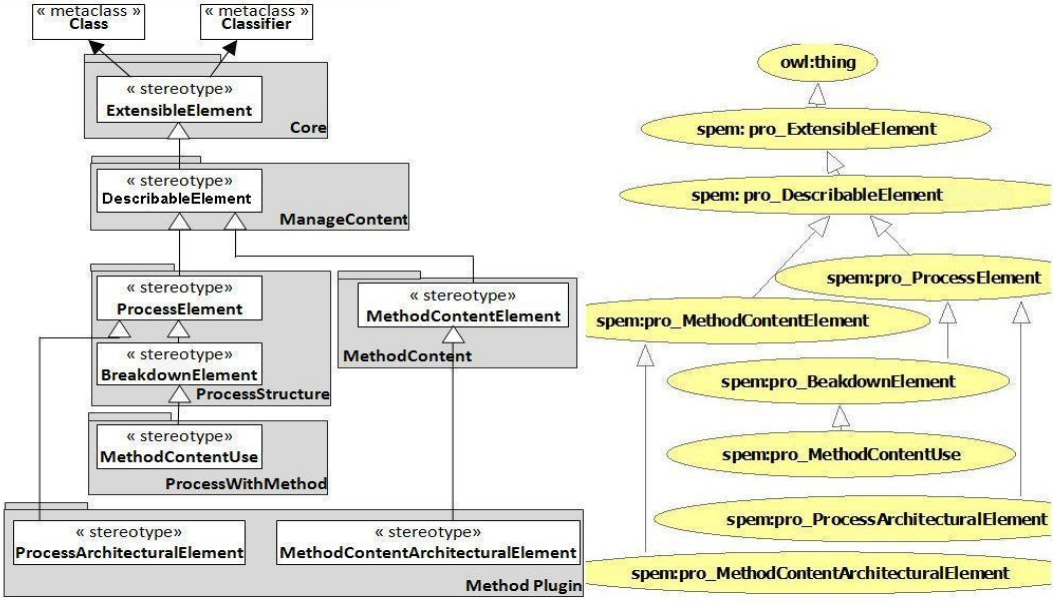


Figure 5.9 La structure de SPEMOntology (vue classe UML vue concept OWL).

5.3.4 La capitalisation des connaissances procédés logiciels

La capitalisation des connaissances se fait par instanciation. L'instanciation de l'ontologie se fait en utilisant la bibliothèque Jena [67], Jena est un Framework écrit en Java, qui permet de manipuler des documents RDF, RDFS et OWL ; il fournit en plus un moteur d'inférences permettant des raisonnements sur les ontologies. Nous utilisons ce Framework pour l'instanciation de notre ontologie. Comme cité précédemment (chapitre-3-) SPEMOntology doit contenir plusieurs types de connaissances, nous instancions quatre types de connaissances :

- Les connaissances des architectures de PLs. Très peu de modèles de PLs à base d'ALs ont été développés, de plus, ceux qui sont développés ne respectent pas notre sémantique. En prenant en compte ces remarques, nous renonçons à la réutilisation de ce type de modèles de PLs. Les connaissances architecturales capitalisées sont propres à la compagnie utilisatrice de SPEMOntology ; ces connaissances décrivent leurs stratégies et politiques de développement. Les concepts concernés par l'instanciation sont les concepts rajoutés lors de l'extension du métamodèle SPEM (figure-5.10-).

- Les connaissances vocabulaire référence : Les connaissances de l'ontologie sont dédiées à l'utilisateur humain, il est très important que ces connaissances soient comprises par l'utilisateur final. Un vocabulaire de référence qui est spécifique à la compagnie utilisatrice est définit. Afin de capitaliser de type de connaissances, les concepts instanciés sont les concepts du métamodèle "Method Content" (figure-5.11-).

Néanmoins, il faut souligner que ces concepts sont sollicités pour décrire différents types de connaissances : des connaissances contenus méthodes (qui est la fonction principale de ce métamodèle), le vocabulaire référence et le type des éléments architecturaux. Afin de différencier ces connaissances, nous rajoutons à la classe génératrice "Method Content Element" un Data Type Property "concepts_kind" ; ce dernier peut avoir une des trois valeurs : "MC" pour Method Content, "VR" pour Vocabulary Reference et "AE" pour Architectural Element.

- Les connaissances utilisées extraites des modèles de PLs existants : Les concepts concernés par cette instanciation sont les concepts des metamodèles "Process structure" et "Process With Method"

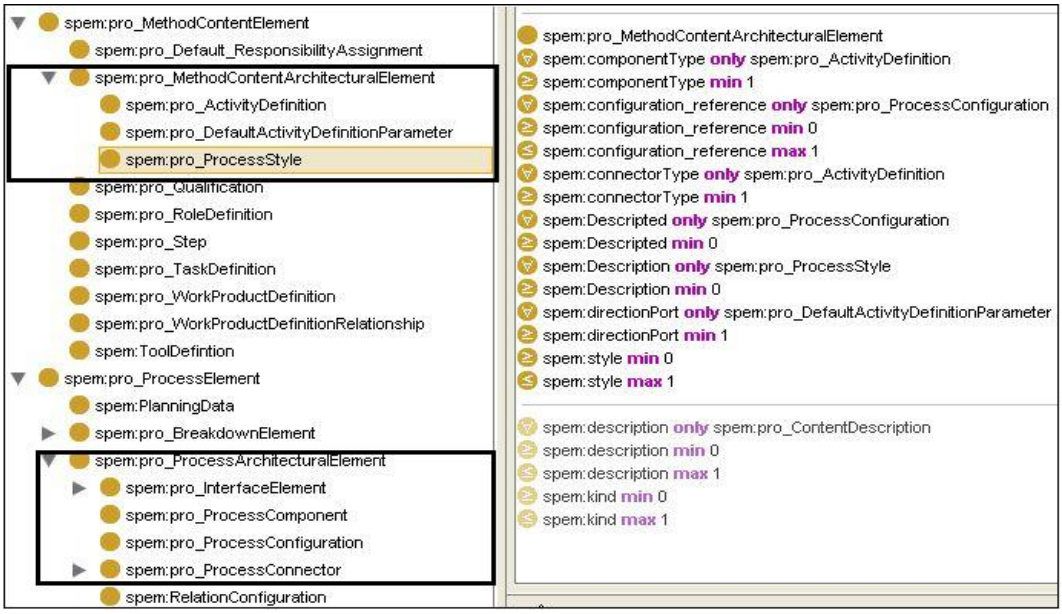


Figure 5.10 Les concepts utilisés pour capitaliser les connaissances des architectures de PLs.

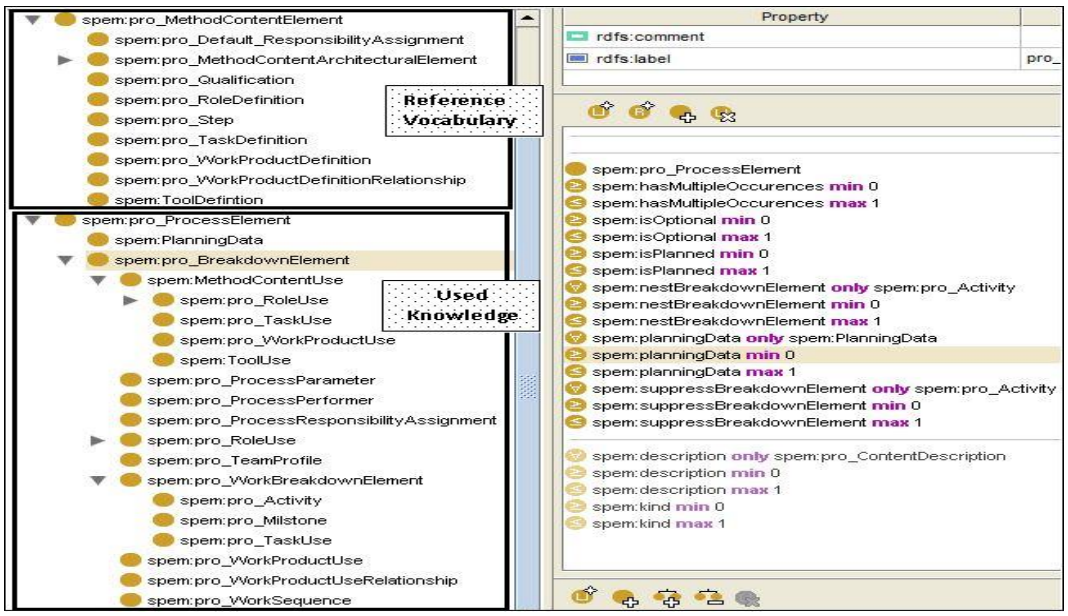


Figure 5.11 Concepts utilisés pour capitaliser le vocabulaire référence.

(figure- 5.11-). Cette étape se fait de manière automatique et va être détaillée dans les sections suivantes.

- La gestion des correspondances entre ces différentes connaissances : Ayant instancié l'ontologie avec le vocabulaire référence et les connaissances des modèles de PLs existants, gérer les instances synonymes est la dernière étape de l'instanciation de SPEMOntology.

Les concepts concernés par cette étape sont généralement des associations entre les concepts du vocabulaire référence et les concepts des connaissances modèles de PLs existants. Ces correspondances sont déjà définies dans le métamodèle SPEM et peuvent être exploitées directement figure-5.12-.

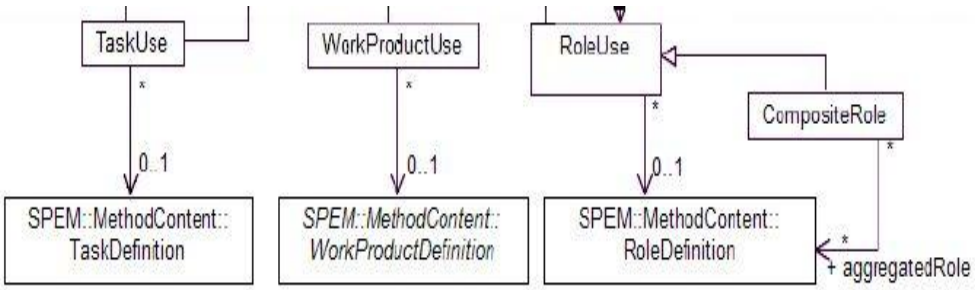


Figure 5.12 Quelques associations SPEM exploitées pour gérer les correspondances entre les connaissances PLs [88].

La capitalisation des connaissances des architectures de PLs, du vocabulaire référence ainsi que l'établissement des correspondances est faite de manière manuelle ce qui est reconnu comme un inconvénient non négligeable. De plus, ces étapes doivent être faites par un expert de la compagnie utilisatrice de SPEMOntology, ce qui est aussi un autre inconvénient (dépendance des connaissances des experts). Néanmoins, il faut noter que ces étapes sont faites une seule fois. De plus, elles contribuent à la constitution de la mémoire et de l'expertise de la compagnie. Après la capitalisation, ces connaissances peuvent être utilisées plusieurs fois sans l'intervention d'experts de PLs.

En revanche, la capitalisation des connaissances à partir des modèles de PLs existants est faite de manière automatique, pour chaque langage de modélisation de PLs, un instanciateur doit être développé. L'instanciateur applique des techniques de rétro-ingénierie en se basant sur la logique du langage de modélisation de PL et sur l'identification des ses mots réservés. Ainsi, il est possible d'identifier les connaissances réutilisables et de les extraire de manière automatique. Dés que l'information est repérée, elle est stockée selon l'élément OWL adéquat (Concept, Data Type, Object Type).

Cette étape est très importante car, nous devons repérer les informations réutilisables ; les informations organisationnelles telles que les noms des développeurs, les plannings...etc. ne doivent pas être prises en compte et doivent être ignorées lors de l'instanciation.

5.3.4.1 La gestion des instanciateurs des procédés logiciels

Afin de faciliter l'étape de capitalisation des connaissances PLs utilisées, nous avons développé un programme qui permet d'intégrer des instanciateurs de différents langages de modélisation de PLs. L'intérêt et de pouvoir intégrer autant d'instanciateurs que de modèles de PLs candidats à la réutilisation et de permettre la réutilisation d'un grand nombre de modèles de PLs.

Le composant "Instantiator Library Manager" du prototype utilise une librairie pour sauvegarder les différentes informations concernant les instanciateurs et leur mode d'utilisation. La figure -5.13- illustre le contenu de la librairie. A partir de l'extension du fichier il est possible d'identifier puis de charger l'instanciateur adéquat.

Il est évident que les instanciateurs doivent être développés au préalable. Afin d'illustrer l'étape de l'instanciation automatique, nous avons développé deux instanciateurs pour deux types de modèles de PLs : les modèles PBOOL [39] et les modèles EPF (Eclipse Process Framwork)[44].

```
library.xml
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <Library>
3  <Instantiator description="" id="instantiator.pbool" name="PBOOL Instantiator">
4  <managedExtension>
5  <extension enabled="true">pbool</extension>
6  </managedExtension>
7  </Instantiator>
8  <Instantiator description="" id="instantiator.spem" name="SPEM Instantiator">
9  <managedExtension>
10 <extension enabled="true">xml</extension>
11 </managedExtension>
12 </Instantiator>
13 </Library>
```

Figure 5.13 Librairie qui permet de sauvegarder les informations des instanciateurs.

5.3.4.2 L'instanciateur PBOOL

PBOOL est un langage de modélisation de PL simple et facile à comprendre. Le langage PBOOL définit quatre modèles qui détaillent les concepts de base du PL : modèle d'activité, modèle de produit, modèle de Rôle et modèle de stratégie [39].

Dans notre travail, nous nous intéressons qu'au modèle d'activité, la figure-5.14- illustre un exemple d'un modèle PBOOL qui décrit un ensemble d'activités. Une activité PBOOL est organisée en plusieurs activités imbriquées. L'imbrication est illustrée par les appels de fonctions et l'ordre d'exécution des activités est illustré par l'ordre des appels de fonctions. Une activité est constituée de produits en entrée, de produits en sortie, du rôle du réalisateur, des appels de fonctions et de l'assistance nécessaires à la réalisation de l'activité.

```
activity Analyse {analyste}

artifact -- Produits manipulés
rapport1 ; liste_des_besoins1; cu_global1;

use -- Produit en entrer
rapport1 ;

define -- Produits en sortie
dictionnaire_de_données ;

require --pre condition
rapport1 available ;

ensure -- post condition
dictionnaire_de_données available,

Refinement sketch General is

call Identification_des_besoins_fonctionnels (rapport1 ; liste_des_besoins1) ;
call Etablissement_diagramme_cas_utilisation_global (liste_des_besoins1 ; cu_global1) ;
call Fragmentation_diagramme_cas_utilisation_global (cu_global1 ; cu1 ; cu2 ; cu3) ;
call ...
call ...
call ...
end ;
```

```
activity Identification_des_besoins_fonctionnels {analyste}
artifact -- Produits manipules
rapport1 ; liste_des_besoins1 ;
use -- Produits en entré
rapport1 ;
define -- Produits en sortie
liste_des_besoins1 ;

require -- pre conditions
rapport1 available,

ensure -- post conditions
liste_des_besoins available,

achievement sketch General do....
end ;
```

Figure 5.14 Exemple de code PBOOL.

Lors de la conception de chaque instanciateur, il est important d'établir les correspondances entre les

concepts modèle de PL et les concepts SPEMOntology. Ces correspondances permettrons d'identifier les connaissances réutilisables et leur emplacement dans SPEMOntology. Le tableau-5.1- illustre les correspondances établies et qui sont suivies par l'instanciateur PBOOL.

PBOOL concept	**SPEMOntology concept**
Activity	Activity
Artefact	WorkProductUse
Use	WorkProduct In
Define	WorkProduct Out
Role	Role Use
Performer	Process perfomer
require	Precondition
ensure	postcondition
Activity	TaskUse
Sequence call	Work Sequence

Table 5.1 Les correspondances adoptées entre les concepts PBOOL et les concepts de SPEMOntology.

Nous remarquons que le langage PBOOL n'utilise pas la notion de Task Use, alors que dans SPEM, une "Activity" atomique est constituée d'au-moins une Task Use. Dans notre correspondance nous supposons que le nom de l'Activity atomique de PBOOL correspond à une Activity SPEM et à sa propre Task Use en même temps.

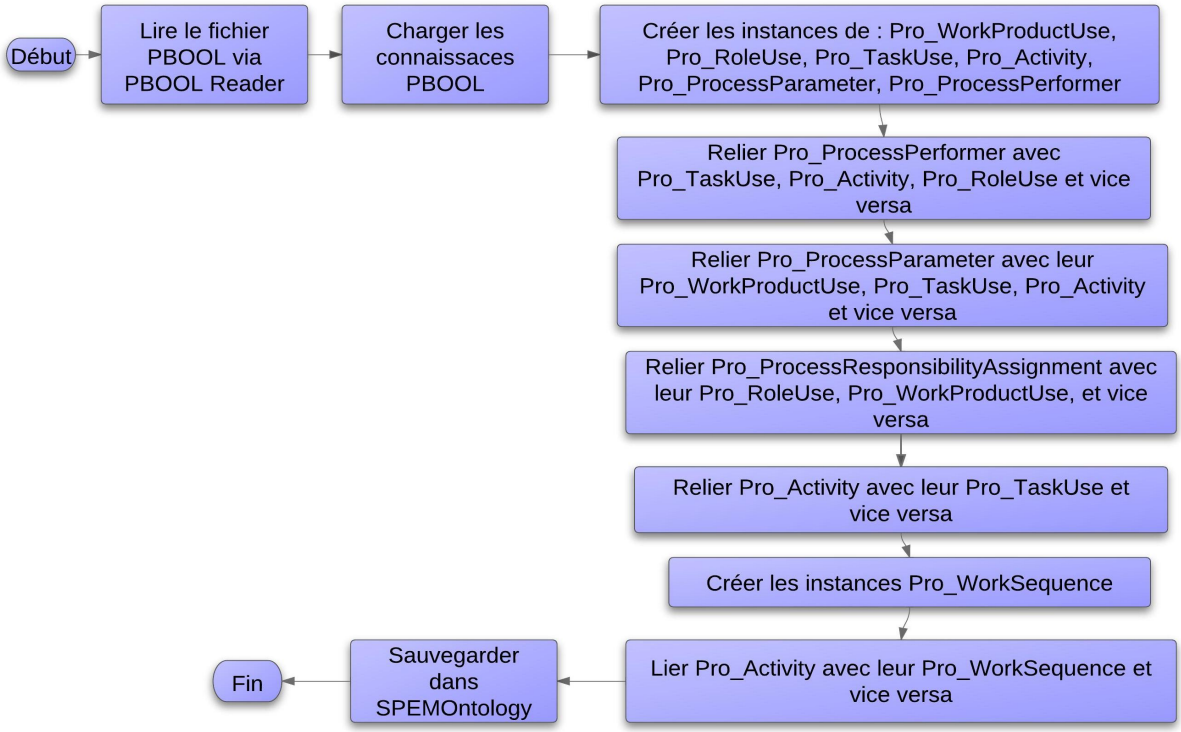


Figure 5.15 Organigramme décrivant les étapes d'extraction des connaissances PL à partir de modèles PBOOL.

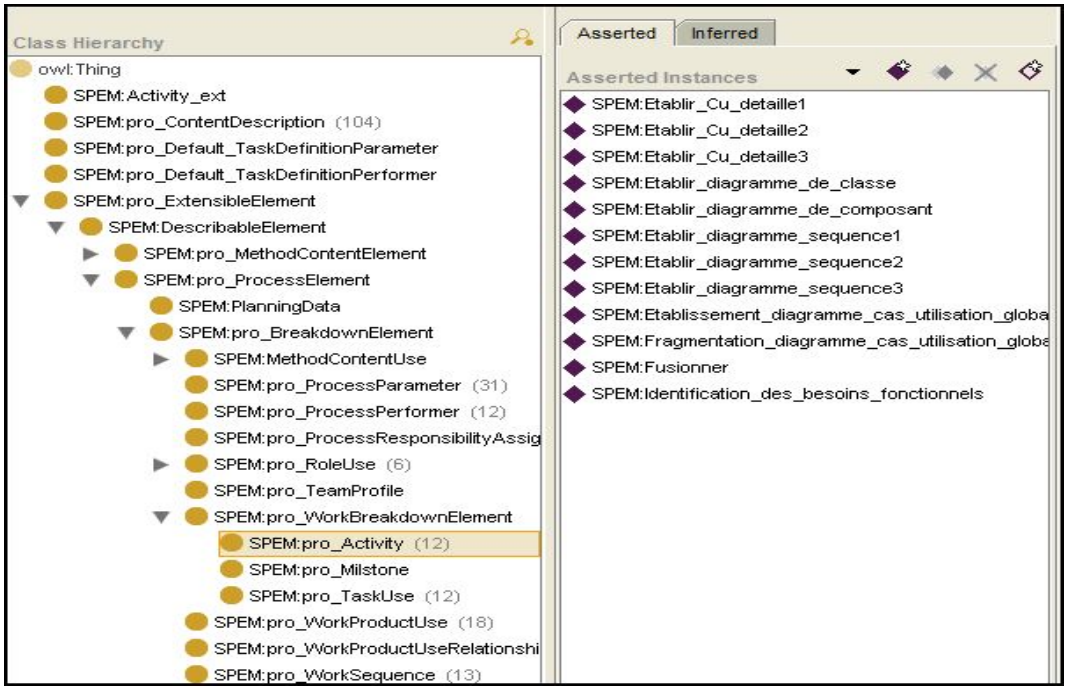


Figure 5.16 Le résultat de l'instanciation de SEMOntologie avec un modèle PBOOL.

L'organigramme présenté par la figure-5.15 décrit les étapes d'extraction des connaissances et d'instanciation de SPEMOntology. Comme première étape, nous chargeons puis et parcourons le modèle PBOOL, et cela dans le but d'identifier les concepts de base de PBOOL (en suivant les mots réservés du langage).

Dans notre travail, nous récupérons que les activités atomiques et leurs enchainements, chaque activité composée (qui est reconnue par l'existence du mot réservé "refinement") est remplacée par ces sous activités jusqu'à arriver aux activités atomiques (reconnues par le mot réservé "achievement"). L'ordre d'exécution est récupéré à partir des ordres des appels de fonctions.

5.3.4.3 L'instanciateur EPF (Eclipse Process Framework)

EPF (Eclipse Process Framework) Composer est un outil développé dans le cadre du sous-projet Eclipse Process Framework (EPF), c'est une application assez riche construite sur le Framework Eclipse et utilisant les plug-ins. Le but d'EPF Composer est d'offrir un outil de modélisation de processus complexes aux chefs de projets, aux directeurs de projets qui ont en charge la maintenance et l'implémentation d'un processus [44].

EPF Composer est un outil permettant à divers intervenants de décrire un processus, de quelque nature qu'il soit, même si l'intention première est de fournir un outil de support méthodologique à des approches telles que RUP, OpenUP, Scrum, ou encore XP. Les modèles EPF sont décris sous fichiers XML (figure-5.17-), mais consultables de manière graphique dans l'EPF composer (figure-5.18-).

De la même manière que pour les modèles PBOOL, l'extraction des connaissances exige l'établissement des correspondances concepts EPF/concepts SPEMOntology ; ces correspondances sont plus faciles à faire car les modèles EPF respectent le métamodèle SPEM et exploitent sa logique, son organisation et sa terminologie.

Figure 5.17 Modèle de PL décrit à l'aide EPF Composer (version XML).

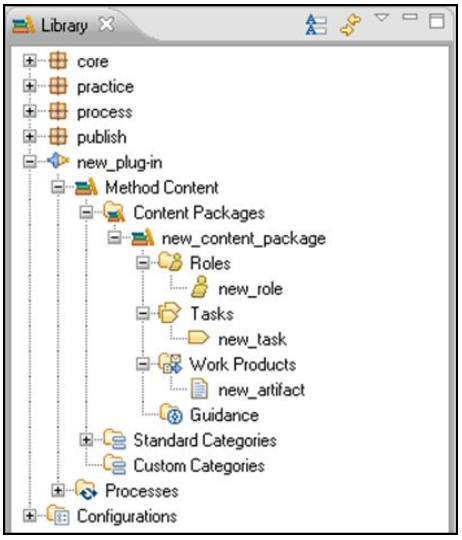


Figure 5.18 Modèle de PL décrit à l'aide EPF Composer (version graphique).

L'organigramme présenté par la figure -5.19- décrit les étapes d'extraction de connaissances et d'instanciation de SPEMOntology à partir de modèles EPF.

Comme première étape nous chargeons et parcourons le modèle EPF. L'identification des concepts de base du modèle EPF (en suivant les balises du langage XML), permet d'identifier les connaissances pertinentes et d'instancier les concepts de SPEMOntology correspondants. L'instanciation se fait de la même manière que pour les modèles PBOOL.

Contrairement au modèle PBOOL, le modèle EPF possède des informations "Method Content", des informations "Process Structure" et des informations "Process With Method" en même temps, ce qui fait que l'instanciation est plus longue car nous récupérons plus de connaissances par rapport à l'instanciation de modèles PBOOL.

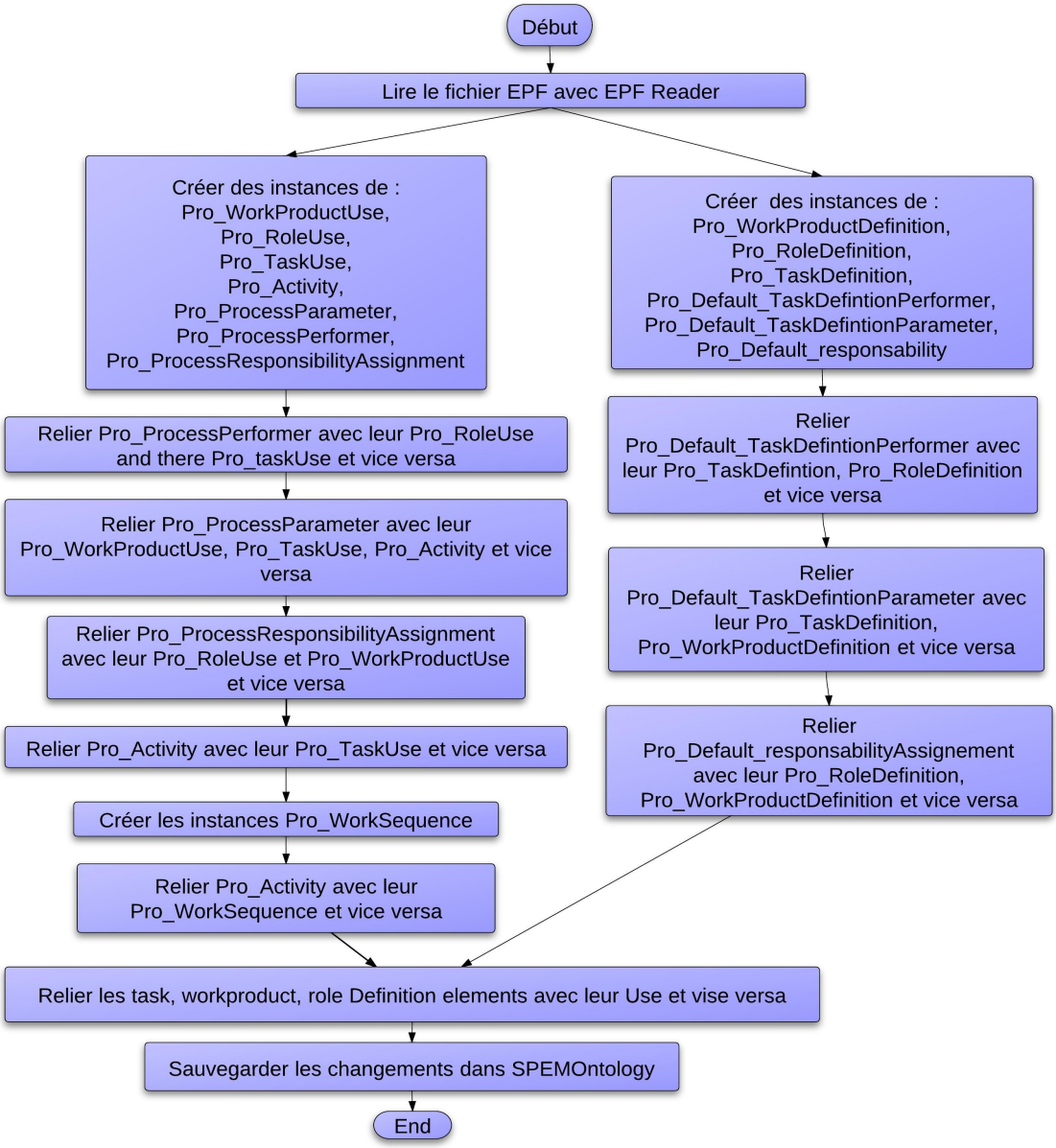


Figure 5.19 Les étapes d'instanciation de SPEMOntology à partir de modèles EPF.

5.3.5 L'inférence et l'enrichissement des connaissances de SPEMOntology

L'enrichissement sémantique des connaissances de SPEMOntology passe par la définition de règles d'inférence qui permettent de déduire de nouvelles connaissances. Outre la possibilité de partager et de réutiliser l'ontologie SPEMOntology, l'inférence constitue un des principaux avantages de l'utilisation d'une ontologie de domaine.

Nos règles d'inférence sont décrites à l'aide du langage SWRL [89], SWRL (Semantic Web Rule Language) est un langage de règle qui enrichit la sémantique d'une ontologie définie en OWL, il permet contrairement au langage OWL, de manipuler des instances par des variables (?x, ?y, ?z...). SWRL ne permet pas de créer des concepts ni des relations, il permet simplement de déduire puis d'ajouter de

nouvelles instances suivant les valeurs des variables et la satisfaction de la règle.

Dans notre approche, deux catégories de règles sont définies.

5.3.5.1 Les règles de cohérence et de persistance

Ces règles préservent la cohérence des connaissances capitalisées. Comme précisé dans les sections précédentes, l'extraction de connaissances de plusieurs sources hétérogènes peut créer des "vides" qu'il faut combler tout en respectant la logique de SPEMOntology. Ces règles sont identifiées après l'étude de chaque type de modèle PL instancié. En effet, chaque type de modèle de PL instancié engendre un certain type de "vides" qu'il faut combler en définissant les règles adéquates.

- **Exemple 1 :** en prenant en compte le modèle PBOOL, nous remarquons que seul les liens de succession sont définis, les liens de précédence sont absents, afin de combler ces vides nous définissons les règles suivantes :

  $SPEM : predecessor(?SPEM : W, ?SPEM : A1) \wedge$
  $SPEM : linktoprodecessor(?SPEM : A2, ?SPEM : W) \rightarrow$
  $SPEM : successor(?SPEM : W, ?SPEM : A2) \wedge$
  $SPEM : linktosuccessor(?SPEM : A1, ?SPEM : W)$

  Si l'activité A1 est le prédécesseur de l'activité A2 alors A2 est le successeur de A1 (W est le Work Sequence qui les relie).

  $SPEM : successor(?SPEM : W, ?SPEM : A2) \wedge$
  $SPEM : linktosuccessor(?SPEM : A1, ?SPEM : W) \rightarrow$
  $SPEM : predecessor(?SPEM : W, ?SPEM : A1) \wedge$
  $SPEM : linktoprodecessor(?SPEM : A2, ?SPEM : W)$
  Si l'activité A2 est le successeur de l'activité A1 alors A1 est le prédécesseur de A2 (W est le Work Sequence qui les relie).
- **Exemple 2 :** Pour synchroniser les connaissances Task Use et son activité nous avons définit la règle suivante :

  $SPEM : perform(?SPEM : Task1, ?SPEM : Performer1) \wedge$
  $SPEM : NestedBreakdownElment(?SPEM : Task1, ?SPEM : Activity1) \rightarrow$
  $SPEM : performs(?SPEM : Activity1, ?SPEM : Performer1)$
  Si le développeur "Performer1" est associé à la Task Use "Task1" et "Task1" est une Task Use de "Activity1" alors "Performer1" est un développeur de "Activity1".
- **Exemple 3 :** PBOOL n'exploite pas le concept Task Use, et même si nous avons supposé que chaque activité atomique a le même nom que sa Task Use, et que nous avons créé la Task Use correspondante lors de la phase instanciation, les Process Performers, Roles et Tools ne sont pas reliés à la Task Use, par conséquent, nous définissons les règles suivantes :

  $SPEM : performs(?SPEM : Activity1, ?SPEM : Performer1) \wedge$
  $SPEM : NestedBreakdownElement(?SPEM : Task1, ?SPEM : Activity1) \wedge$
  $sameas(?SPEM : Task1, ?SPEM : Activity1) \rightarrow$
  $SPEM : perform(?SPEM : Task1, ?SPEM : Performer1)$

Si le "Performer1" est le développeur responsable de "Activity 1" et "Task 1" est une Task Use de "Activity1" alors "Performer1" est responsable de "Task 1".

- **Exemple 4 :** par soucis de cohérence nous avons décidé de ne pas récupérer les informations organisationnelles qui décrivent l'environnement du modèle de PL source. Ainsi, concernant le "Process Performer" nous avons décidé de mettre son "Rôle" à la place du nom du développeur. En d'autres termes, au lieu de garder le nom du développeur nous utilisons son Rôle. La règle SWRL est comme suit :

  $SPEM : linkReoleUsePerformer(?SPEM : Role1, ?SPEM : Performer1) \wedge$
  $Differentfrom(?SPEM : Role1, ?SPEM : Performer1) \rightarrow$
  $SPEM : linkReoleUsePerformer(?SPEM : Role1, ?SPEM : Role1)$
- **Exemple 5 :** pour gérer l'hétérogénéité des termes utilisés nous avons opté pour l'utilisation de la propriété "Content Description" qui aura comme valeur le nom référence de cette instance. Le nom référence sera déduit des concepts "Method Content" reliés et qui permettent d'identifier le vocabulaire référence correspondant. La règle d'inférence est comme suit :

  $SPEM : referenceTask(?SPEM : TU1, ?SPEM : TD1) \wedge$
  $SPEM : Concepts_Kind(?SPEM : TD1, VR) \wedge$
  $SPEM : description(?SPEM : TD1, ?SPEM : Desc1) \rightarrow$
  $SPEM : description(?SPEM : TU1, ?SPEM : Desc1)$
  (TU : Task Use, TD : Task Definition)

  Cette règle permet de copier la description du vocabulaire référence (TD) dans la description de Task Use (TU), ce qui permet de donner aux Task Use une terminologie compréhensible. Nous appliquons le même raisonnement pour les concepts Work Product Use et Role Use.

  $SPEM : referenceWorkProduct(?SPEM : WPU1, ?SPEM : WPD1) \wedge$
  $SPEM : Concepts_Kind(?SPEM : WPD1, VR) \wedge$
  $SPEM : description(?SPEM : WPD1, ?SPEM : Desc1) \rightarrow$
  $SPEM : description(?SPEM : WPU1, ?SPEM : Desc1)$
  (WPU1 : Work Product Use, WPD : Work Product Definition, Desc : Description, VR : Vocabulary Reference).

  $SPEM : referenceRole(?SPEM : RU1, ?SPEM : RD1) \wedge$
  $SPEM : Concepts_Kind(?SPEM : RD1, VR) \wedge$
  $SPEM : description(?SPEM : RD1, ?SPEM : Desc1) \rightarrow$
  $SPEM : description(?SPEM : RU1, ?SPEM : Desc1)$
  (RU : Role Use, RD : Role Definition).

5.3.5.2 Les règles d'émergence de nouvelles solutions

Ces règles permettent la création de nouvelles relations entre des connaissances de différents modèles de PLs. Ces règles permettent de déduire de nouveaux enchainements entre les activités des différents modèles de PLs et la déduction de nouveaux enchainements entre les Task Use de différents modèles de

PLs, ces règles sont comme suit :

Règle 1 : Cette règle permet de créer des enchainements entre les Tasks Uses, son principe est que si les Work Product Use de deux Task Use sont de directions différentes et qu'ils ont la même description (même vocabulaire référence), une Task Use peut être la suivante de l'autre (selon les directions des Work Product Uses)

$SPEM : parameter(?SPEM : WPU1, ?SPEM : ProcessParm1) \wedge$
$SPEM : OunProcessParameter(?SPEM : TU1, ?SPEM : ProcessParm1) \wedge$
$SPEM : direction(?SPEM : ProcessParm1, OUT) \wedge$
$SPEM : parameter(?SPEM : WPU2, ?SPEM : ProcessParm2) \wedge$
$SPEM : OunProcessParameter(?SPEM : TU2, ?SPEM : ProcessParm2) \wedge$
$SPEM : direction(?SPEM : ProcessParm2, IN) \wedge$
$SPEM : description(?SPEM : WPU1, ?SPEM : Desc1) \wedge$
$SPEM : description(?SPEM : WPU2, ?SPEM : Desc2) \wedge$
$SPEM : sameas(?SPEM : Desc1, ?SPEM : Desc2) \rightarrow$
$SPEM : successor(?SPEM : ProcessParm1, ?SPEM : TU1) \wedge$
$SPEM : linktosucccessor(?SPEM : TU2, ?SPEM : ProcessParm1) \wedge$
$SPEM : predecessor(?SPEM : ProcessParm1, ?SPEM : TU2) \wedge$
$SPEM : linktoprodecessor(?SPEM : TU1, ?SPEM : ProcessParm1)$
TU : Task Use, WPU : Work Product Use, Desc : Description, ProcessParam : Process Parameter.

Règle 2 :
$SPEM : parameter(?SPEM : WPU1, ?SPEM : ProcessParm1) \wedge$
$SPEM : OwnedParameter(?SPEM : A1, ?SPEM : ProcessParm1) \wedge$
$SPEM : direction(?SPEM : ProcessParm1, OUT) \wedge$
$SPEM : parameter(?SPEM : WPU2, ?SPEM : ProcessParm2) \wedge$
$SPEM : OwnedParameter(?SPEM : A2, ?SPEM : ProcessParm2) \wedge$
$SPEM : direction(?SPEM : ProcessParm2, IN) \wedge$
$SPEM : description(?SPEM : WPU1, ?SPEM : Desc1) \wedge$
$SPEM : description(?SPEM : WPU2, ?SPEM : Desc2) \wedge$
$sameas(?SPEM : Desc1, ?SPEM : Desc2) \rightarrow$
$SPEM : successor(?SPEM : ProcessParm1, ?SPEM : A1) \wedge$
$SPEM : linktosucccessor(?SPEM : A2, ?SPEM : ProcessParm1) \wedge$
$SPEM : predecessor(?SPEM : ProcessParm1, ?SPEM : A2) \wedge$
$SPEM : linktoprodecessor(?SPEM : A1, ?SPEM : ProcessParm1)$

Cette règle est la même que la précédente, sauf qu'elle est appliquée aux activités/

En prenant en compte le nombre important des concepts de SPEMOntology, il est possible de définir un nombre considérable de règles d'inférence. Cette partie de l'approche n'a pas été explorée dans sa totalité et des travaux plus approfondis doivent être effectués pour l'inférence de connaissances PLs.

5.4 Ingénierie par la réutilisation des PLs : réutilisation effective des connaissances capitalisées

5.4.1 La recherche et l'acquisition des connaissances d'une configuration PL

Comme cité dans le chapitre -4-, deux types de recherches de connaissances configuration PL sont définis :

- **Recherche d'une configuration PL ad-hoc :** les entrées des cette requête sont les produits en entrée de la configuration recherchée (Work products In) et les produits que doit fournir la configuration recherchée (Work Products Out). Ces produits correspondent aux "Ports Data Flow In" et aux "Port Data Flow Out" de la configuration recherchée.
- **Recherche d'une configuration PL qui respecte un ou plusieurs styles :** Les entrées de cette requête sont les "Work Product In" et les "Work Product Out", ainsi que les styles PL que la configuration recherchée doit respecter.

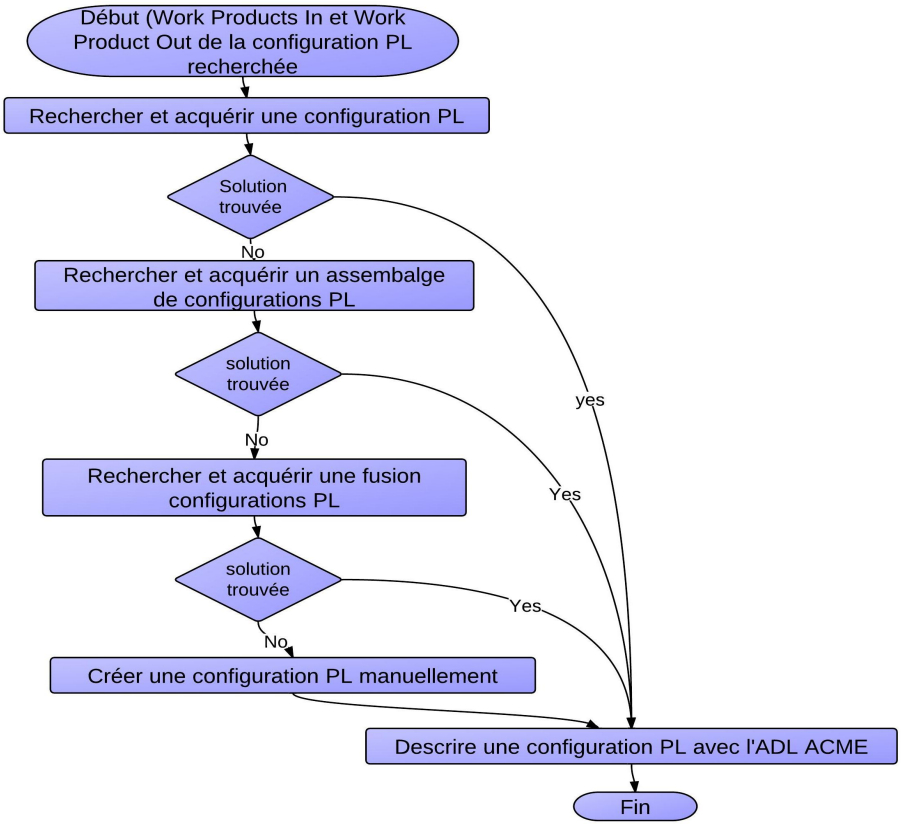


Figure 5.20 Les étapes principales de recherche de connaissances configuration PL.

Notre recherche doit donner la priorité aux configurations déjà établies par la compagnie utilisatrice (figure-5.20-). Ainsi, dans les deux cas, les requêtes de recherche donne la priorité à la recherche d'une seule configuration, si la solution n'est pas trouvée, la recherche est étendue à la recherche d'une combinaison de plusieurs configurations PL (assemblage de configurations PL).

La recherche d'un assemblage de configurations PL se fait en suivant les correspondances des noms des Ports Data Flow des configurations PL, car les configurations PLs n'ont pas d'ordre d'exécution

et par conséquent, pas de connecteur qui les relies. Nous rappelons qu'il n y a pas de problème de d'hétérogénéité de vocabulaire, car les configurations sont décrites à l'aide du vocabulaire référence.

Demandant plus de temps et de ressources, l'accès à l'état interne des configurations (fusion de configurations) est la dernière solution envisagée. La recherche de fusion de configurations PLs se fait en suivant les connecteurs Control Flow afin de pouvoir fusionner plusieurs configurations.

Si toutes ces étapes n'aboutissent pas, la solution n'existe pas dans l'ontologie et le développeur PL est contraint de décrire sa configuration manuellement (figure-5.20-).

La recherche d'une configuration est l'étape la plus simple dans l'algorithme de recherche car elle ne demande pas de combiner différents connaissances PL. Elle correspond a une simple requête de recherche qui doit (si la solution existe) retourner une configuration PL. Si la requête ne retourne pas de solution nous lançons l'algorithme de recherche d'un assemblage de configurations PL.

5.4.1.1 Le principe de recherche d'un assemblage de configurations PL

La recherche d'une configuration PL se fait à partir d'un ensemble de configurations PL. Le principe de l'algorithme suit les étapes suivantes :

- **Étape -1- : Initialisation** Dans cette étape nous vérifions si tous les Work Product In et tous les Work Product Out de la configuration recherchée existent dans l'ensemble des Ports Data Flow de l'ontologie.
  Si un des Ports Data Flow[1] recherchés n'existe pas alors la solution n'existe pas et l'algorithme de recherche n'est même pas lancé (figure-5.22- (Etape-1-)).

- **Étape -2- : Marquage descendent ou marquage des successeurs avec "+"**
  La figure-5.21- résume l'étape -2- et l'étape-3- de l'algorithme suivi pour la recherche des connaissances configuration PL à partir d'un ensemble de configurations PL.
  Pour toutes les configurations existantes dans notre ontologie faire :
  Tant qu'il y a des configurations non marquées par un "+" ou par "vue" faire :
  - Si la configuration ne contient pas exactement un sous ensemble des Ports In recherchés alors nous marquons la configuration comme "vue", ce qui signifie que cette configuration ne répond pas à notre demande.
  Dans la figure (figure-5.22- (Etape-2-)) la configuration "C9" est marquée par "vue" car le produit "P5" n'appartient pas à l'ensemble des Ports In.

  - Si la configuration contient exactement un sous ensemble des Ports In recherchés alors nous marquons la configuration avec un "+", puis nous déclenchons le marquage successif des configurations successeurs de cette configuration.

  - Nous arrêtons le marquage successif si la configuration est déjà marquée par "+" ou si la configuration ne possède pas de successeurs.

  - Nous arrêtons le marquage descendant (étape-2-) si toutes les configurations sont marquées par "+" ou par "vue".
  Étape -3- : Marquage ascendant ou marquage des prédécesseurs avec "-"

1. Les ports manipulés dans cet algorithme sont de type Data Flow, en effet, nous rappelons qu'il n'y pas d'ordre de précédence entre les configurations PL, et par conséquent, pas de Ports de type Control Flow.

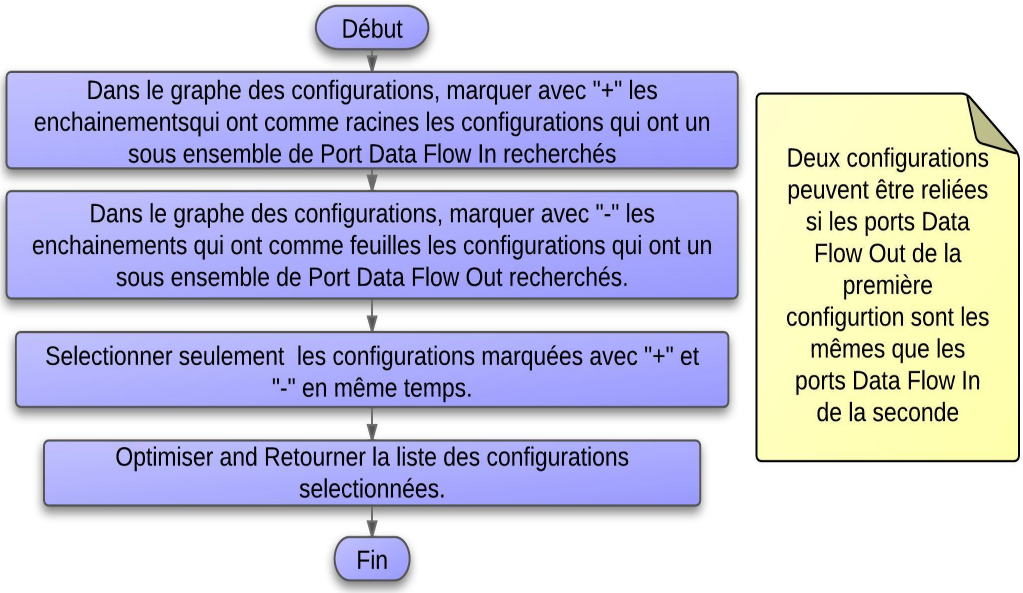


Figure 5.21 Les étapes de constitution de l'ensemble des configurations PL des solutions potentielles.

- Avant de commencer le marquage ascendant nous supprimons les marquages "vue".
- De la même manière que l'étape précédente, nous faisons le parcours inverse en partant des Work Product Out de la requête. Ainsi, pour toutes les configurations existantes dans notre ontologie faire :
Tant qu'il y a des configurations qui ne sont pas parquées par un "-" ou par "vue" faire :
- Si la configuration ne contient pas un sous ensemble des Ports Out recherchés alors nous marquons la configuration par "vue", ce qui signifie que cette configuration ne répond pas à notre recherche.
- Si la configuration contient un sous ensemble des Ports Out recherchés alors nous marquons la configuration avec un "-", par la suite nous déclenchons un marquage successif des configurations prédécesseurs.
- Nous arrêtons le marquage successif si la configuration est déjà marquée par "-" ou si la configuration ne possède pas de prédécesseurs (figure-5.22- (étape-3-)).
- Nous arrêtons le marquage ascendant (étape-3-) si toutes les configurations sont marquées soit par un "-" soit par un "vue".
- Les configurations marquées par "+" et "-" en même temps constituent les configurations de la solution potentielle.

– **Étape -4- : Constitution et optimisation de la solution finale**
La figure-5.23- résume l'étape -4- de l'algorithme suivie pour la recherche des connaissances configuration PL à partir d'un ensemble de configurations PL. Dans notre travail, la solution optimale est celle qui a les enchainements les plus courts et qui minimise le nombre de configurations. La constitution et l'optimisation de la solution finale se fait selon les étapes suivantes :
- Après avoir repéré les configurations qui peuvent contribuer à la solution finale, et après avoir vérifié l'existence de l'intégralité des Ports In et des Ports Out recherchés dans l'ensemble des

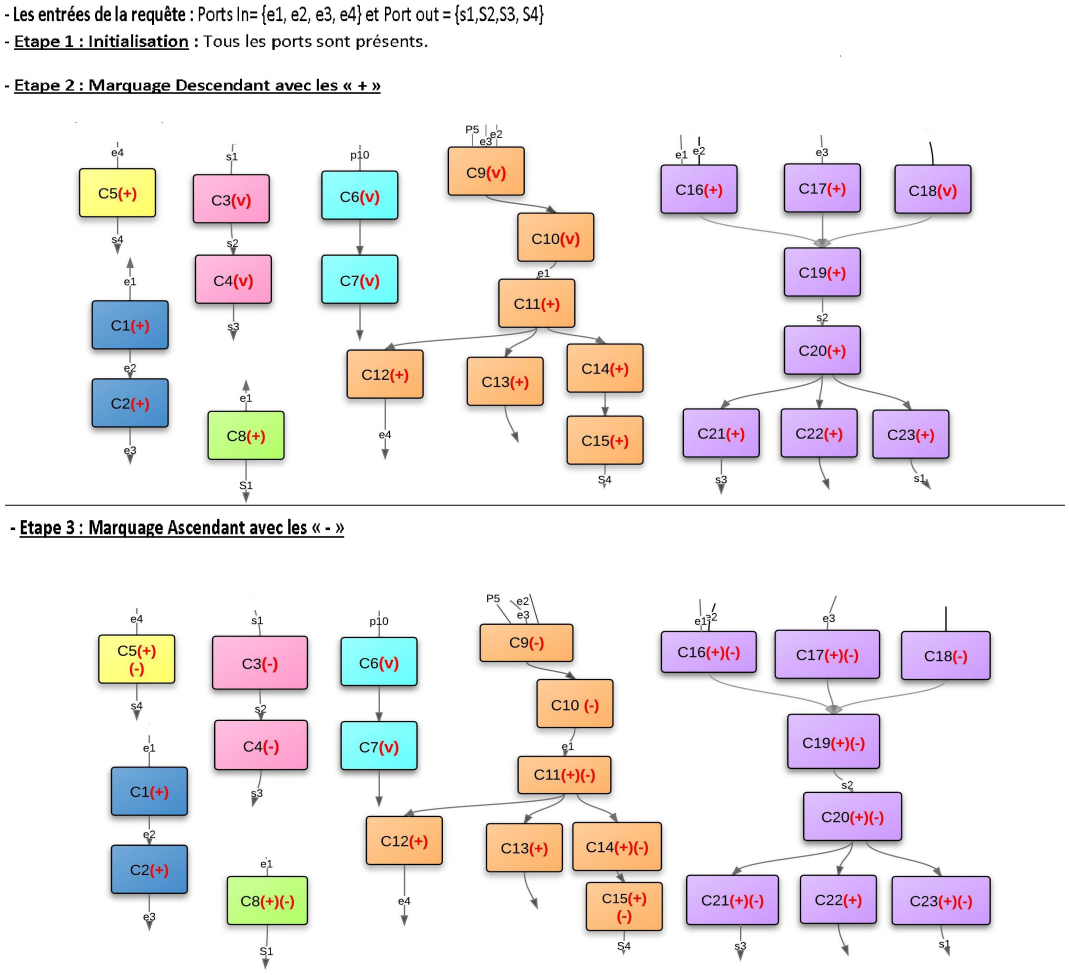


Figure 5.22 Exemple de recherche de connaissances configuration PL à partir d'une ensemble de configurations.

solutions potentielles, nous laçons la recherche de la solution finale.

- Nous regroupons les solutions potentielles en un graphe G= (S, A), [2] Tels que : "S "représente l'ensemble des sommets et "A" représente l'ensemble des arêtes. Un sommet représente une configuration PL. Une arrête entre deux sommets C1 et C2 représente le lien entre un ensemble de Ports Out de C1 et un ensemble de Ports In de C2 et qui ont les mêmes noms (figure-5.24).

Pour chaque composante connexe[3] "Graphe i" :

1- Nous calculons le nombre *"SNS(Graphe i)"* qui représente le nombre des "Sommets Non Souhaitables", en d'autres termes, le nombre des configurations qui n'ont aucun Ports In ou Port Out recherchés.

2. En théorie des graphes, un graphe non orienté V=(S,A) est dit connexe si quels que soient les sommets "i" et "j" de S, il existe une chaîne de "i" vers "j". C'est-à-dire, s'il existe une suite d'arêtes de A permettant d'atteindre "j" à partir de "i".
3. Un sous-graphe connexe maximal d'un graphe non orienté quelconque est une composante connexe de ce graphe.

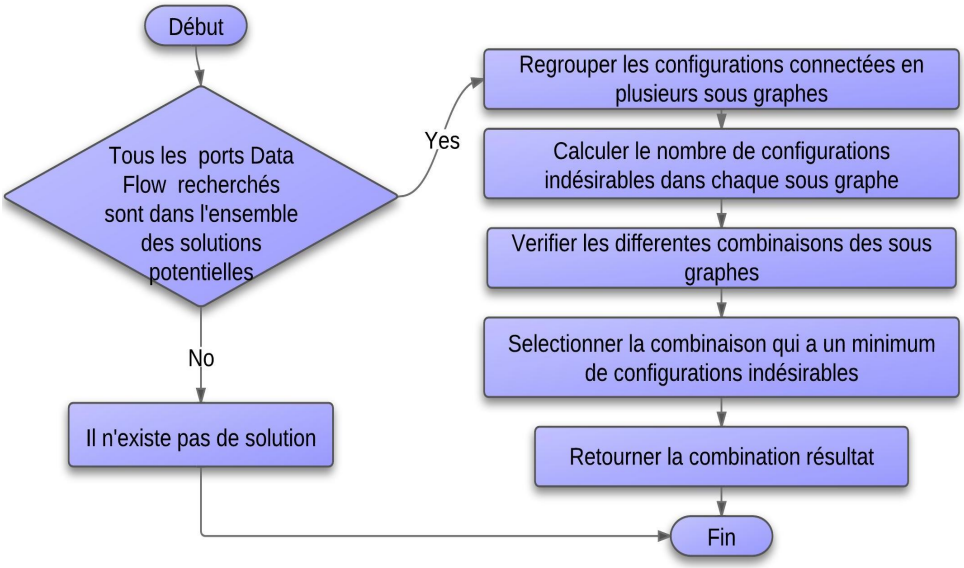


Figure 5.23 Les étapes de constitution et d'optimisation de la solution (Configuration PL) finale.

2- Nous identifions les Ports Out recherchés que le graphe "i" couvre, nous constituons deux ensembles : *"EnsPortIn(Graphe i)"* : ensemble de ports In du graphe i. et *"EnsPortOut(Graphe i)"* : ensemble de ports In du graphe i.

3- Notre solution optimale est l'ensemble des graphes connexes qui a :
- Un "SomSNS" (la Somme des Sommets Non Souhaitables) est minimal tel que :
"SomSNS" = $\sum SNS(graphe\ i)(i = 1...n)$: tel que "n" est le nombre des sous-graphes connexe de la solution.
- L'intersection des ensembles des EnsPortOut des graphes connexes est un ensemble vide, ce qui assure qu'un Work Product out n'est pas produit plusieurs fois :
$EnsPortOut(Graphe\ 1) \cap EnsPortOut(Graphe\ 2)... \cap EnsPortOut(Graphe\ \ n)$: tel que "n" est le nombre des graphes connexes de la solution optimale.

A cette étape toute les combinaisons sont testées. Dans le pire des cas, c'est tous les graphes connexes sont sélectionnés et constituent la solution optimale (figure-5.24).

L'avantage de cet algorithme est qu'il est décidable, en d'autres termes, si la solution existe elle est trouvée, optimisée et affichée et si la solution n'existe pas l'algorithme nous le signale.

5.4.1.2 Le principe de recherche d'une fusion de configurations PLs

L'algorithme de recherche d'un assemblage de configurations ne trouve pas de solution dans les cas suivants :

- Un Port In ou un Port out de la configuration recherchée n'existe pas (soit à l'étape -1- ou bien à l'étape -4-).
- On ne trouve pas d'enchainement entre un Port In et un Port Out de la configuration recherchée.

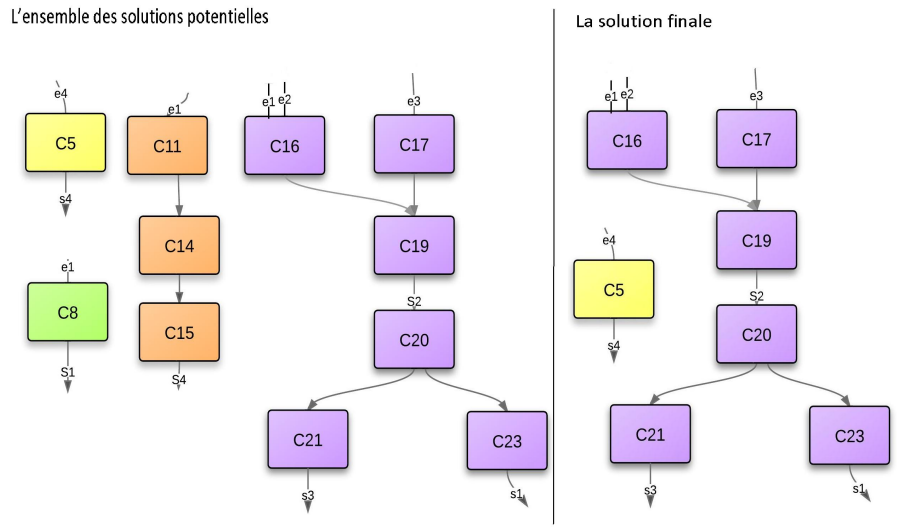


Figure 5.24 Exemple de recherche de connaissances configuration PL à partir d'une ensemble de configurations (étape -4-).

Si nous nous retrouvons dans cette situation, l'algorithme de recherche d'une fusion de configurations est lancé. Dans cette partie de la recherche nous accédons à l'état interne des configurations et nous recherchons un assemblage de composants. L'algorithme précédent est exécuté mais sur un ensemble de composants PLs. Nous recherchons toujours des ports Data Flow, cependant, lors de la recherche le passage d'un composant à un autre se fait selon les Ports Control Flow.

La seule différence réside dans les marquages successifs, en effet, Les composants peuvent être reliés par des connecteurs Control Flow qui explicite l'ordre d'exécution des composants PLs, par conséquent, le parcours de recherche des Work Products se fait en suivant les connecteurs Control Flow.

Les figures -5.25- et -5.26- résume les étapes de recherche de configuration PL dans un ensemble de composants PLs :

Si cette étape ne retourne pas de solution, nous créons une nouvelle configuration que nous intégrons dans SPEMOntology.

5.4.1.3 Le principe de recherche de configurations PL respectant des styles PLs

La recherche de configuration qui respecte un style suit exactement les mêmes étapes que la recherche d'une configuration ad-hoc. La priorité est toujours donnée à la recherche d'une seule configuration PL. Si la solution n'existe pas, nous recherchons un assemblage de configurations puis une fusion de configurations. La différence réside dans l'ajout d'étapes de vérification de la conformité des éléments architecturaux aux styles recherchés. Ainsi :

- Dans le cas de la recherche d'un assemblage de configurations PLs, l'ensemble initial de recherche est l'ensemble des configuration qui respectent le ou les styles demandés (figure-5.27-).
- Dans le cas de recherche d'une fusion de configurations PLs, l'ensemble initial de recherche est l'ensemble des composants qui respectent les composants types du ou des styles demandés (figure-5.28-).

Dans tous les cas de recherche, les connaissances configuration PL trouvées sont stockées dans un ficher ACME. Les connecteurs Control Flow sont rajoutés selon les arêtes du graphe résultat. Ce résultat constitue une des entrées du programme de déploiement de la configuration PL.

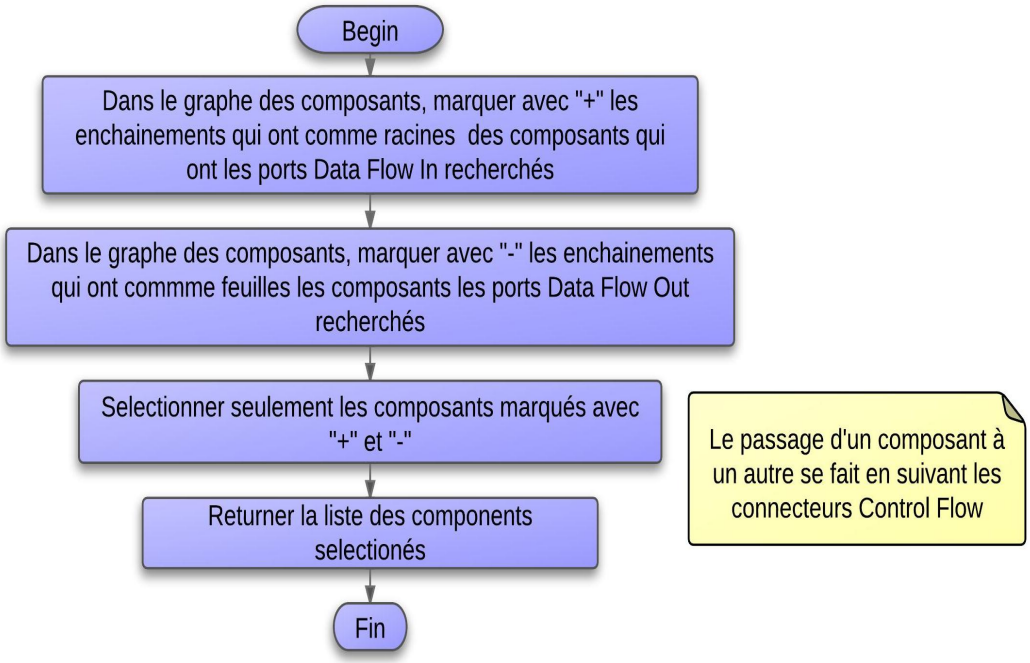


Figure 5.25 Les étapes de marquage ascendant et descendant des composants.

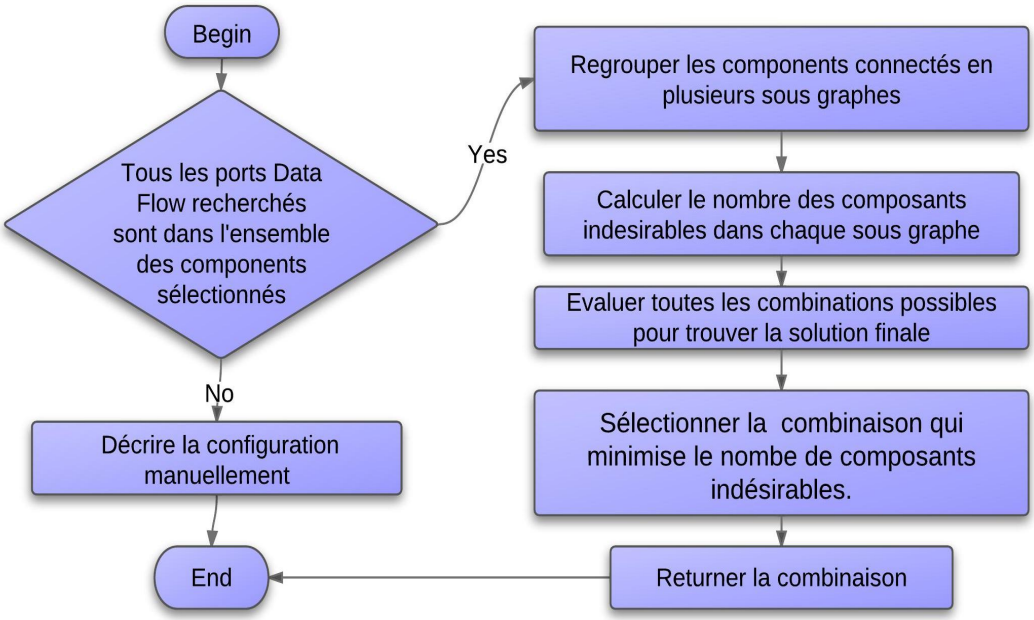


Figure 5.26 Les étapes de recherche de la solution optimale a partir d'un ensemble de composants.

5.4.2 La recherche et l'acquisition des connaissances des composants PLs

Les connaissances du composant PL ne sont rien d'autres qu'un enchainement d'activités. Rechercher les connaissances d'un composant PL revient à rechercher (dans un ensemble d'activités PL) un enchainement d'activités PL qui a comme produits en entrée les Ports Data Flow In du composant PL, et

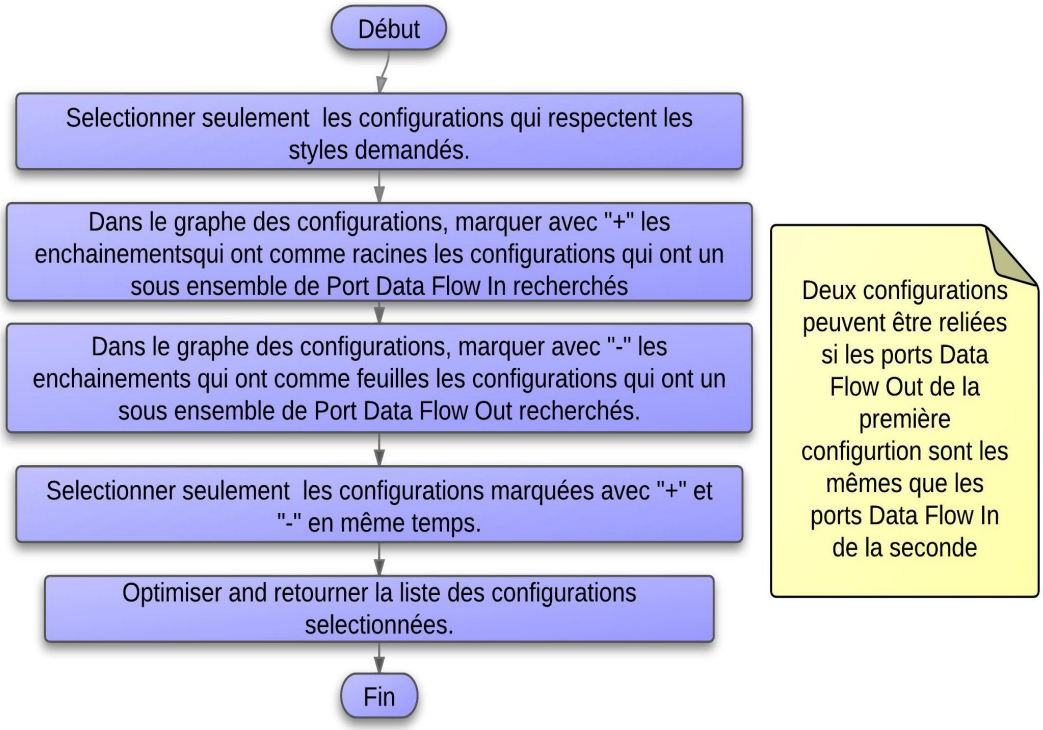


Figure 5.27 Les étapes de recherche de connaissances configuration PL qui respecte un ou plusieurs styles à partir d'un ensemble de configurations PLs.

a comme produits en sortie les Ports Data Flow Out du composant PL recherché.

Contrairement aux algorithmes de recherche de configurations PL, le problème de l'hétérogénéité du vocabulaire utilisé se pose et doit être traité. Le problème d'hétérogénéité du vocabulaire se pose dans deux cas :

- Les requêtes de recherches : les entrées de la requête sont les Ports Data Flow In et les Ports Data Flow Out (qui sont décris avec le vocabulaire de la compagnie) alors que la recherche se fait sur des connaissances extraites à partir de différents modèles de PLs des différentes sources.
- L'affichage du résultat final : de la même manière, les connaissances extraites sont décrites à l'aide du vocabulaire des modèles de PLs sources, le problème de l'hétérogénéité du vocabulaire se pose lors de l'affichage de la solution finale.

Ce problème a été anticipé et a été traité lors de la capitalisation des connaissances, nous avons mis à jours le data objet "Content Description" des Work Product Use, des Task Use et des Activity avec le nom de référence de ces derniers (compréhensible par l'utilisateur). Plusieurs règles d'inférence ont été dédiées à cet effet. Ainsi, le parcours se fait en consultant les "Content Description" au lieu des noms des instances Work Product Use.

5.4.2.1 Le principe de recherche d'un enchainement d'activités

Les étapes de l'algorithme de recherche de connaissances composant PL sont comme suit (figure-5.29-) :

- **Étape -1- : Initialisation** Afin de réduire le temps de recherche, nous utilisons une ontologie temporaire qui contiendra les concepts et les Object Type et Data Type properties minimaux pour

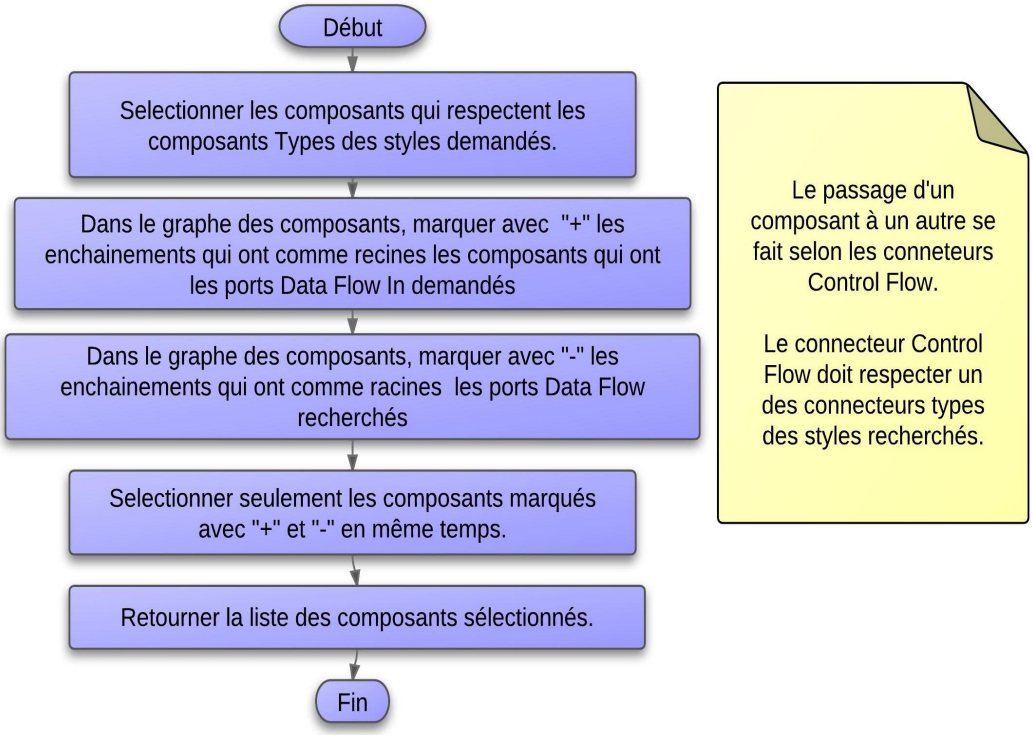


Figure 5.28 Les étapes de recherche de connaissances configuration PL qui respecte un ou plusieurs styles à partir d'un ensemble de composants PLs.

rechercher un enchainement d'activités. Les concepts et les Object Types de base de cette ontologie temporaire sont décris dans le tableau -5.2-.

Concept	**Object Type Property**
Pro_Activity : son extension décrit les instances activités sans Data Type properties.	- **haveINProducts** : représente les produits In utilisés par une activité. - **haveOUTProducts** : représente les produit Out utilisés par une activité. - **linkToPredecessors** : représente l'activité qui succède l'activité en cour. - **linkToSuccessors** : représente l'activité qui précède l'activité en cour.
Pro_WorkProductUse : représente les produits manipulés par les activités.	- **usedAsInBy** : représente les différents activés qui utilise ce produit comme entrée. - **usedAsOutBy** : représente les différents activités qui utilise ce produit comme sortie.

Table 5.2Les concepts et les object Type properties de l'ontologie temporaire.

Nous copions les activités avec leurs produits et enchainements respectifs (les ordres d'exécution) à partir de l'ontologie SPEMOntology vers l'ontologie temporaire. L'ontologie temporaire contiendra alors seulement les squelettes des solutions potentielles. Ces connaissances nous permettent de décrire un graphe d'activités où :
- Chaque sommet représente une activité.

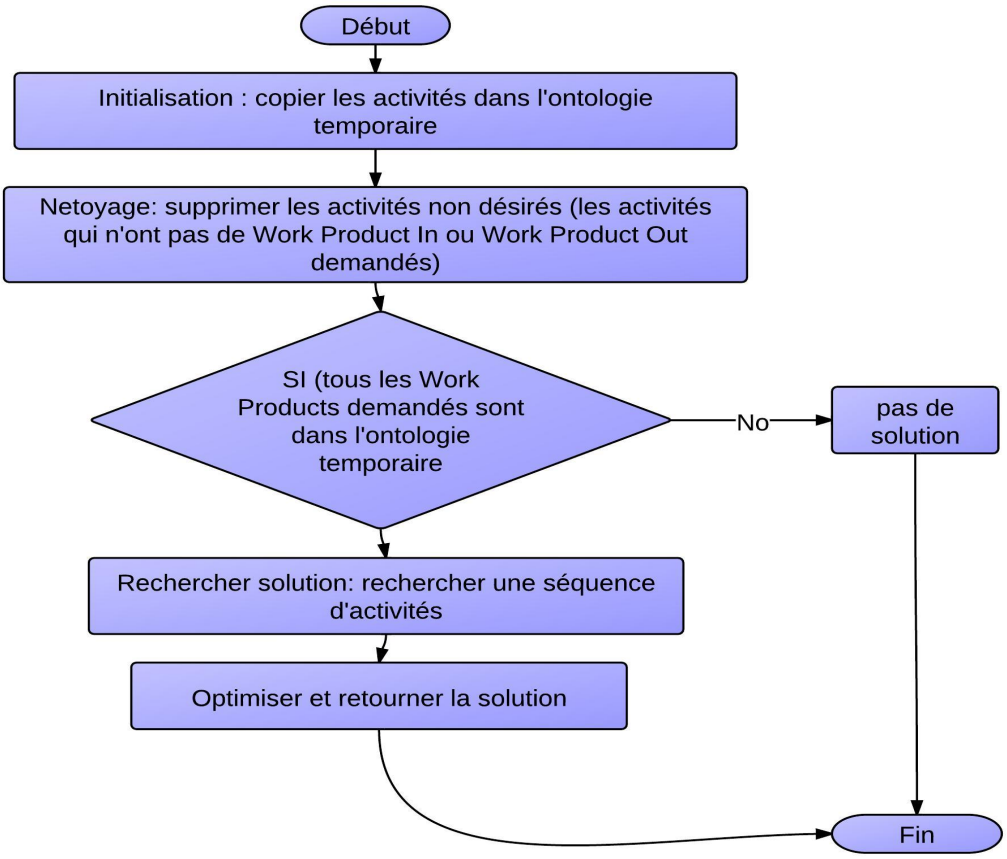


Figure 5.29 Les étapes de l'algorithme de recherche de connaissances composant PL.

- Un arc entre un sommet A1 vers un sommet A2 signifie que l'activité A1 précède l'activité A2.

– **Étape -2- : Nettoyage**
L'algorithme que nous avons établi recherche la solution exacte et non une solution équivalente qui peut remplacer le composant recherché. Une solution exacte est l'enchainement d'activités qui possède en entrée exactement les Work Products In et en sortie exactement les Work Product Out recherchés.
Une solution équivalente est l'enchainement d'activités qui a des Work Product Out additionnels ou qui a des Work Product In manquants, elle permet toujours de répondre à la demande, mais a besoin de moins de Work Products In ou fournit plus de Work Products Out.
L'étape de nettoyage revient à éliminer les activités qui ne répondent pas à la requête. Cela revient à supprimer toutes les instances qui sont dans le cas suivant :
- Les activités au sommet des arborescences qui utilisent au moins un Work Product In qui n'est pas recherché.
Le résultat de cette étape est un graphe qui contient seulement les éléments nécessaires à notre recherche (les solutions potentielles).

– **Étape -3- : Test d'existence de solutions potentielles** La recherche d'un enchainement d'activités peut commencer si cette condition est vérifiée :
- Tous les Work Product In ou un Work Product Out sont présents dans l'ontologie temporaire.

Dans le cas contraire : la solution n'existe pas et l'algorithme de recherche de l'enchainement d'activités n'est pas lancé.

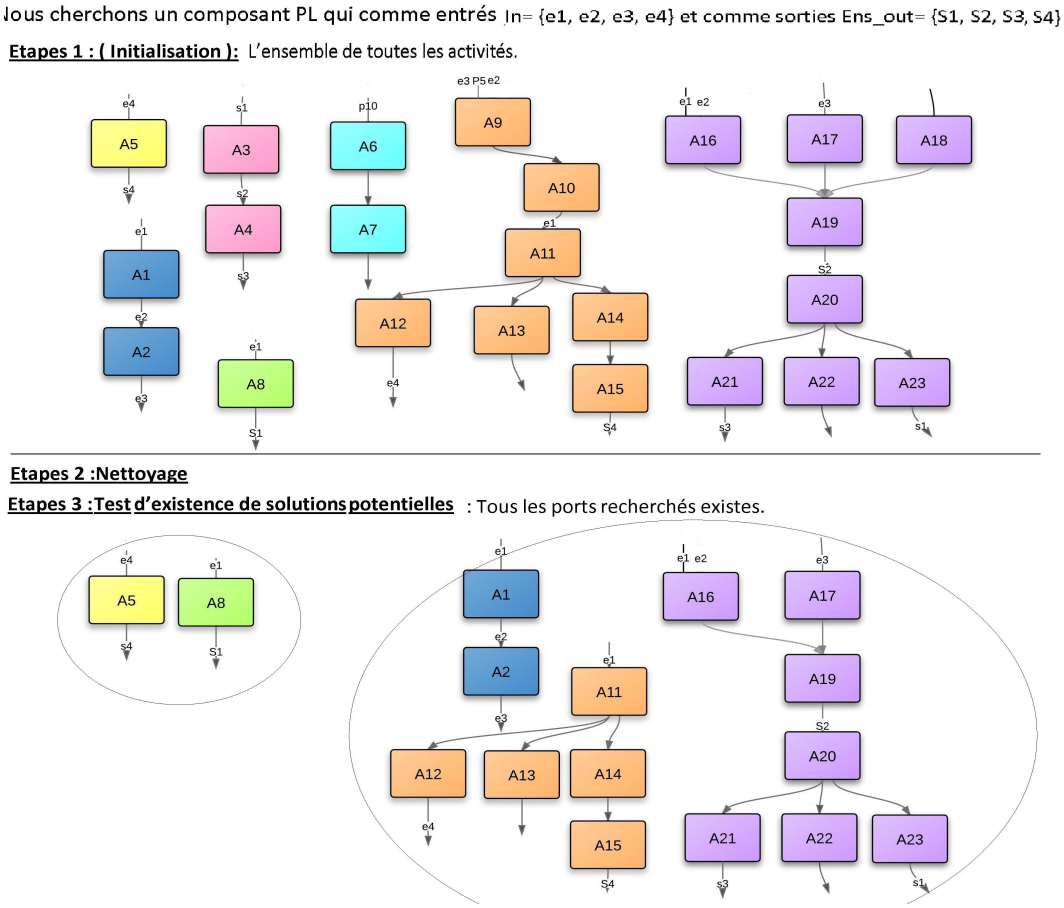


Figure 5.30 Exemple de recherche des connaissances composant PL.

La figure-5.30- illustre un exemple de recherche d'un enchainement d'activités. Nous remarquons que lors de l'étape de nettoyage, les activités A3, A4, A6, A7, A9, A18 ont été supprimées.

– **Étape -4- : Recherche d'un enchainement d'activités.**
Les étapes de recherche de l'enchainement d'activités sont comme suit (figure-
- Copier tout les Work Product In et les Work Product Out recherchés dans deux ensembles Ens_in et Ens_out.
- La condition d'arrêt de l'algorithme est que les ensembles Ens_in et Ens_out deviennent vides.
- Séparer les activités en deux groupes : groupe des sommets isolés "sommet" qui contient les activités avec aucun enchainement, et groupe "enchainement" qui contient le reste des activités.
- Initialiser Graphe_Resultat à l'ensemble vide, cet ensemble contiendra l'ensemble des activités qui répondent à notre demande.
- Exécuter la recherche sur l'ensemble des sommets isolés "sommet" et à chaque fois faire :
Tant que l'ensemble "sommets" n'est pas vide faire :
Copie_in = copie_in - Wok Product In [activité en cour]
Copie_out = copie_out - Wok Product out [activité en cour]
Graphe_Resultat = Graphe_Resultat + activité en cour [i]

Une fois le parcourt des sommets isolés terminé, nous vérifions s'il y a des Work Product In ou des Work Product Out restants dans les ensembles copie_in et copie_out. Si un des ces deux ensembles n'est pas vide nous lançons la recherche sur l'ensemble d'activités "enchainement".
Sachant que les sommets racines des graphes (les activités qui n'ont pas de prédécesseurs) ont tous au moins un Work Products In recherché, et que les Work Product Out des sommets feuilles (les activités qui n'ont pas de successeurs) sont quelconques, nous parcourons ces graphes de manières ascendante en commençant par les feuilles jusqu'à aboutir aux racines. Comme montré dans la figure- 5.29-(étape-4-), pour toutes les feuilles du graphe faire :
Si les Work Products Out de la feuille x sont inclus dans l'ensemble copie_out faire :
- Trouver un enchainement d'activités 'g' qui se termine par la feuille x. (parcourir de manière ascendante à partir de la feuille x jusqu'à arriver à la racine).

Copie_out = copie_out - Work Product Out [la feuille du graphe 'g'].
Graphe_Resultat = Graphe_Resultat + graphe 'g' *Copie_in = copie_in - Work Product In [racine du graphe 'g']*
Copie_out = copie_out - Work Product Out [la feuille du graphe 'g'].
Graphe_Resultat = Graphe_Resultat + graphe 'g'

– ***Étape -5- : Optimisation et transmission du résultat final***
L'étape-4- a comme résultat un ensemble de graphes d'enchainements qui ne sont pas forcément tous reliés et dont certaines activités peuvent être utilisées dans plusieurs enchainements. Dans ce cas, une étape d'optimisation est indispensable, ces enchainements doivent être fusionnés afin de supprimer les activités redondantes.
L'optimisation se fait comme suit :
- Supprimer les sommets isolés qui ont leur activité dans un enchainement d'activités.
- Fusionner les graphes d'enchainements par leurs activités en communs. La fusion revient à regrouper les Work Products In et les Work Products Out sur un même sommet.
Le squelette de la solution finale étant identifié, nous recouperons les connaissances compètes de SPEMOntologie et nous retournons le résultat final sous fichier XML.

La figure-5.31- illustre le résultat final de la recherche de connaissances composant PL.

La structure du fichier XML permet de décrire les connaissances du composant PL comme un fragment de PL qui représente un enchainement d'activités, elle permet de décrire toute ces connaissances à travers des balises dédiées à cet effet (figure-5.32-). Nous remarquons que les balises respectent la terminologie de l'ontologie de domaine, et par transition, respecte la terminologie du métamodèle SPEM.

Nous avons délibérément pris le même exemple (même enchainement) pour illustrer la recherche des connaissances composant PL et la recherche des connaissances configuration PL. Notre objectif est de justifier l'utilisation de deux algorithmes différents. En effet, nous aurions pu nous contenter d'appliquer l'algorithme de recherche de connaissances de configuration pour rechercher les connaissances composant.

Nous constatons que les résultats des deux algorithmes sont différents, les deux donnent des résultats dis "optimaux" (minimise le nombre de sommets) mais intègrent des sommets différents, ainsi :

– Le résultat de la recherche de connaissances composant est plus optimal car il favorise les sommets isolés et par conséquent les activités qui peuvent s'exécuter en parallèle, ce qui n'est pas le cas de l'algorithme de cherche de configuration.

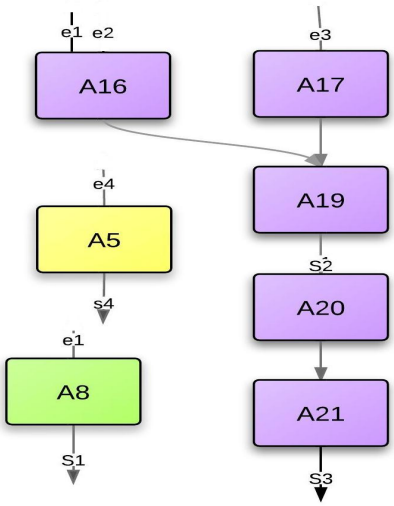
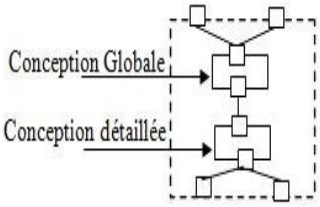


Figure 5.31 Le résultat final après optimisation.

```
?xml version="1.0" encoding="UTF-8"?>
<FragmentDeProcede>
<Activity>
<Task_Use>Conception_Global</Task_Use>
<Process_Performer>Role1</Process_Performer>
<Role_Use>Role1</Role_Use>
<pro_WorkProductUseIn>Doc_Specification</pro_WorkProductUseIn>
<pro_WorkProductUseIn>Documentation</pro_WorkProductUseIn>
<pro_WorkProductUseOut>Ebauche_Diag_de_classes</pro_WorkProductUseOut>
<successor>Conception_Detaillee</successor>
</Activity>
<Activity>
<Task_Use>Conception_Detaillee</Task_Use>
<Process_Performer>Role2</Process_Performer>
<Role_Use>Role2</Role_Use>
<pro_WorkProductUseIn>Ebauche_Diag_de_classes</pro_WorkProductUseIn>
<pro_WorkProductUseOut>Detail_Diag_de_classes</pro_WorkProductUseOut>
<pro_WorkProductUseOut>Documentation2</pro_WorkProductUseOut>
<predecessor>Conception_Global</predecessor>
</Activity>
</FragmentDeProcede>
```



Connaissances Composant PL

Figure 5.32 Le résultat de l'algorithme de recherche de connaissances de composants PL.

– Pour identifier l'ensemble des solutions potentielles, l'algorithme de recherche de configuration fait trois parcours (marquage"+", marquage"-" et recherche de la solution) ce qui n'est pas le cas de l'algorithme de recherche de connaissances composant qui fait deux parcours (suppression des sommets non utiles et recherche de solution).

Au fait, l'algorithme de recherche de connaissances composant est une amélioration de l'algorithme de recherche de connaissances configuration. Nous les avons présentés pour montrer que les algorithmes de recherches ont été améliorés.

5.4.3 Le déploiement d'une architecture de PL

Le déploiement de l'architecture de PL est l'étape finale qui permet de générer le modèle de PL final. Le déploiement de l'architectures PL a besoin de (figure-5.33-) :

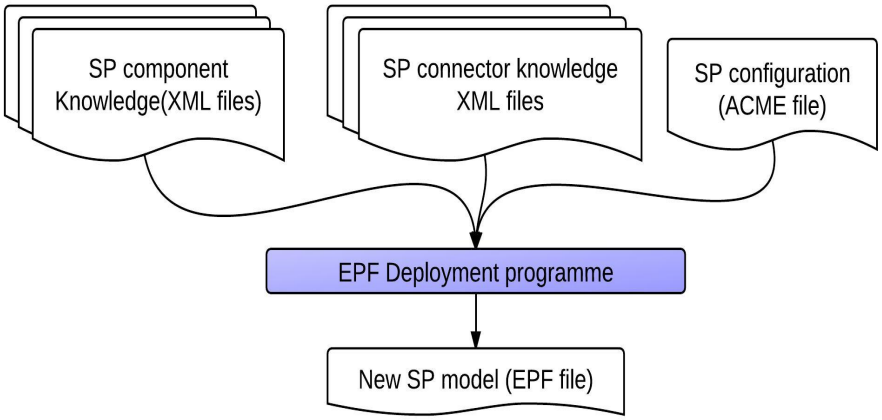


Figure 5.33 Déploiement d'architectures de PLs.

- La configuration PL décrite avec l'ADL ACME.
- Les connaissances de chaque composant PL, décrit par un fichier XML.
- Les connaissances de chaque connecteur PL décrit par un fichier XML.

La configuration PL et les composants PLs sont extrais à partir des connaissances de SPEMOntlogy, en revanche, les connecteurs PLs ne sont pas recherchés mais adaptés, en effet, un connecteur PL est une activité qui est connue (transmission, fragmentation, évaluation ...etc.) ; sa structure est connue, et les informations relatives à son exécution sont déduites à partir des composants PLs qu'ils relient.

5.4.3.1 L'adaptation des connaissances des connecteurs PL

Un connecteur PL est une activité d'adaptation particulière, par conséquent, la structure des fichiers XML des connecteurs PLs est prédéfinie et ressemble à celle du composant PL.

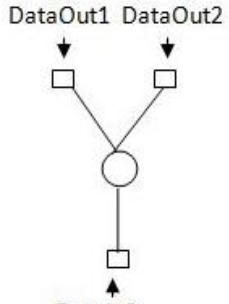
Comme cité dans le chapitre précédent, il existe deux types de connecteurs PLs : des connecteurs Data Flow et des connecteurs Control Flow.

Selon les catégories des connecteurs PLs, il existe trois structures de connecteurs PLs(figure-5.34-).

- **La structure des connecteurs Data Flow :** la caractéristique de cette structure est que nous ne trouvons pas de balises qui décrivent l'ordre d'exécution des connecteurs (Predecessor ou Sucssessor). Ce qui est naturel car ces connecteurs PLs gèrent les transmissions des produits seulement et ne gèrent pas l'ordre d'exécution.
- **Structure du connecteur Précédence :** la caractéristique de cette structure est que nous trouvons uniquement des balises Prececessor et Successor qui décrivent l'ordre d'exécution des composants PLs et pas de balises qui décrivent les transmissions des produits (Pro_WorkProductUseIn et Pro_WorkProductUseOut), car ces connecteurs illustrent les ordres d'exécution seulement.

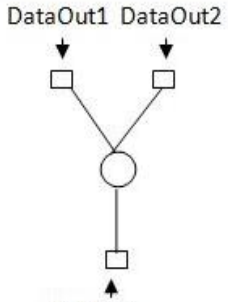
– Structure du reste des connecteurs Control Flow : la caractéristique de cette structure est qu'elle est hybride elle combine les deux structures précédentes. Elle est exactement similaire à celle du composant PL, en d'autres termes, elle procède toutes les balises sans exception. Ces connecteurs PLs assurent l'ordre d'exécution et évaluent la qualité de l'exécution en même temps, ce qui explique l'utilisation de toutes les balises.

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessSnippet>
<pro_Activity>
<pro_TaskUse>fusion</pro_TaskUse>
<Process_Performer>role</Process_Performer>
<pro_RoleUse>role</pro_RoleUse>
<pro_WorkProductUseIn>DateOut1</pro_WorkProductUseIn>
<pro_WorkProductUseIn>DateOut2</pro_WorkProductUseIn>
<pro_WorkProductUseOut>DateIn3</pro_WorkProductUseOut>
</pro_Activity>
</ProcessSnippet>
```



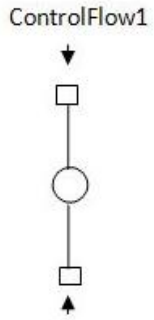
Connecteur fusion

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessSnippet>
<pro_Activity>
<pro_TaskUse>EoTemps</pro_TaskUse>
<Process_Performer>Role</Process_Performer>
<pro_RoleUse>Role</pro_RoleUse>
<pro_WorkProductUseIn>DataOut1</pro_WorkProductUseIn>
<pro_WorkProductUseOut>DataIn2</pro_WorkProductUseOut>
<Predecessor>ControlFlow1</Predecessor>
<Successor>ControlFlow2</Successor>
</pro_Activity>
</ProcessSnippet>
```



Connecteur fusion

```
<?xml version="1.0" encoding="UTF-8"?>
<ProcessSnippet>
<pro_Activity>
<pro_TaskUse>Precedence</pro_TaskUse>
<Process_Performer>Role</Process_Performer>
<pro_RoleUse>Role</pro_RoleUse>
<Predecessor>ControlFlow1</Predecessor>
<Successor>ControlFlow2</Successor>
</pro_Activity>
</ProcessSnippet>
```



Connecteur Precedence

Figure 5.34 Les structures des fichiers XML des connecteurs PLs.

Les connaissances d'un connecteur PL sont mises à jours de la manière suivante :

– Le nom du connecteur garde son nom identifié dans la configuration.
– Le Process Performer est le Process Performer du composant précédent. Dans tous les connecteurs que nous avons définis, le connecteur a un seul composant précédent, excepté le connecteur "Fusion" qui a plusieurs composants PLs précédents, dans ce cas, nous prenons un des Process

Performers de manière aléatoire.
- Le rôle est récupéré de la même manière à partir du composant précédent.
- Les Work Product Use In sont récupérés à partir des Work Product Use Out des composants PLs précédents.
- Les Work Product Use Out sont récupérés à partir des Work Product Use In des composants PLs suivants.
- Les Predecessors sont les noms des dernières activités des composants PLs précédents.
- Les Successors sont les noms des premières activités des composants PLs suivants.

Cette adaptation est faite pour tous les connecteurs de manière à avoir tous les fichiers XML des connecteurs nécessaire au déploiement final.

5.4.3.2 Les types de déploiements d'architectures de PLs implémentés

Comme cité dans le chapitre précédent, et afin de prendre en compte les spécificités des modèles de PLs, plusieurs types de déploiement ont été identifiés.

Dans notre travail, nous avons implémenté deux types de déploiement :

- Déploiement total où toute la configuration PL est déployée en une seule fois.
- Déploiement partiel où le déploiement se fait composant par composant et c'est à l'utilisateur d'identifier le composant à déployer, ce type de déploiement (après amélioration) nous permettra d' implémenter les déploiements itératifs, incrémentaux, distribués et hétérogènes.

5.4.3.3 Cas d'étude : Déploiement total d'architectures de PL dans le l'environnement EPF

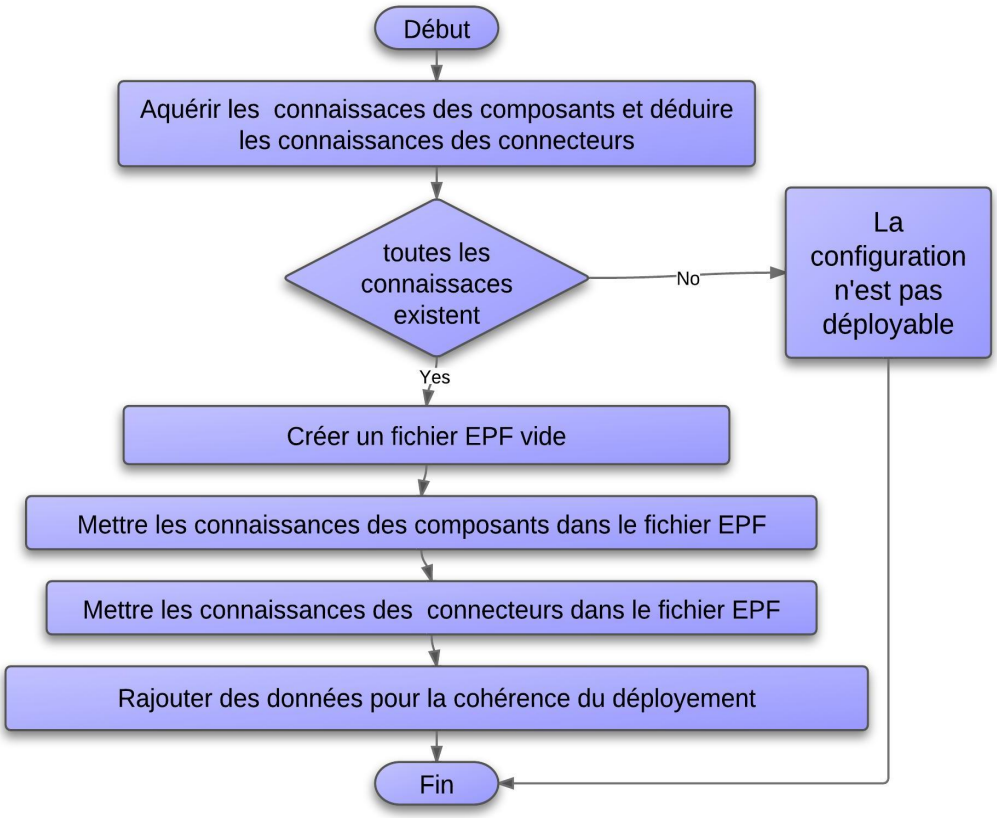


Figure 5.35 Déploiement Total d'architectures de PLs.

La figure-5.35- illustre les étapes de déploiement total qu'une architectures de PL. Comme exemple de déploiement nous générons des modèles EPF. Concrètement, nous exploitons la structure du fichier XML du modèle EPE pour générer notre modèle de PL. De plus, les balises du fichier XML de EPF respecte la terminologie et la logique du métamodèle SPEM ce qui nous facilité le déploiement. EPF exploite des identifiants numériques qui permettent d'identifier de manière unique chaque instance des éléments PLs. Afin de compléter la description du modèle EPF résultat, ces identifiants sont générés aléatoirement, mais sont utilisés de manière cohérente dans le déploiement.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ProcessSnippet>
<ActivitySnippet>
<ProcessPerformer>analyste</ProcessPerformer>
<TaskUse>Etablissement_cas_d'utlisation_global</TaskUse>
<RoleUse>analyste</RoleUse>
<In>besoins_utlisateur</In>
<Out>cas_d'utlisation_global</Out>

</ActivitySnippet><Successor>Etablissement_cas_d'utilisation_detaille</Successor>

<ActivitySnippet>
<ProcessPerformer>analyste</ProcessPerformer>
<TaskUse>Etablissement_cas_d'utilisation_detaille</TaskUse>
<RoleUse>analyste</RoleUse>
<In>cas_d'utlisation_global</In>
<Out>diagramme_de_cas_d'utlisation_detaille</Out>
<Predecessor>Etablissement_des_diagrammes_des_cas_d'utlisation_global</Predecessor>
</ActivitySnippet>

</ProcessSnippet>
```

Figure 5.36 Les connaissances du composant spécification.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ProcessSnippet>
<ActivitySnippet>
<ProcessPerformer>concepteur</ProcessPerformer>
<TaskUse>Etablissement_diag_de_classe_ebauche</TaskUse>
<RoleUse>concepteur</RoleUse>
<In>cas_d'utilisation_detaille</In>
<Out>ebauche_diagramme_de_classe</Out>
<Successor>Etablissement_diag_de_classe_detaille </Successor>
</ActivitySnippet>

<ActivitySnippet>
<ProcessPerformer>concepteur</ProcessPerformer>
<TaskUse>Etablissement_diag_de_classe_detaille</TaskUse>
<RoleUse>concepteur</RoleUse>
<In>ebauche_daigramme_de_classe</In>
<Out>diagramme_de_classes_detaille</Out>
<Predecessor> Etablissement_diag_de_classe_ebauche</Predecessor>
</ActivitySnippet>
</ProcessSnippet>
```

Figure 5.37 Les connaissances du composant conception.

Les figures -5.37- et-5.38- illustrent les fichiers connaissances en entrée du déploiement total. La configuration est décrite avec d'ADL ACME, notre configuration est constituée de deux composants (spécification et conception) reliés par un connecteur "Transmission" qui assure la transmission des don-

nées et un connecteur "Precedence" qui assure l'ordre d'exécution. Chaque fichier XML qui décrit les connaissances d'un composant PL, contient les connaissances de deux activités.

Les connaissances des connecteurs sont déduites à partir des connaissances des composants et de la configuration correspondante.

Le résultat du déploiement est un fichier EPF tels que (figure- 5.38-)) :

- Les composants vont constituer des activités. - Le connecteur "Transmission" constitue une activité transmission. - Le connecteur "Precedence" assure l'ordre d'exécution des activités.

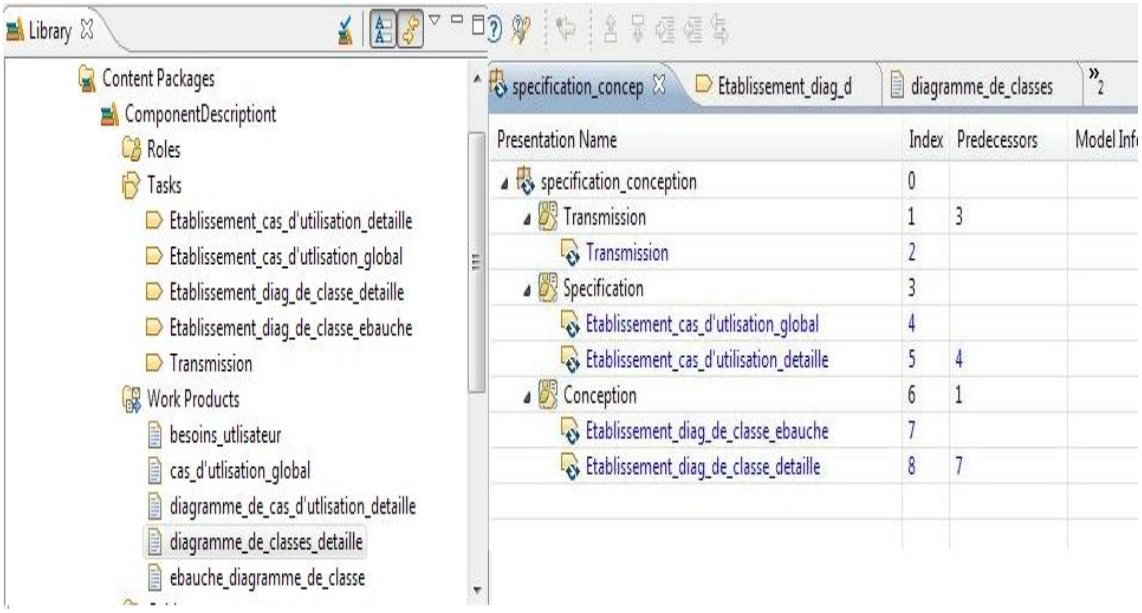


Figure 5.38 Résultat du déploiement Total d'architectures de PLs.

5.5 Conclusion

Dans ce chapitre nous avons présenté, les choix techniques que nous avons adoptés afin d'implémenter notre approche. Ainsi :

- Afin de générer notre ontologie, nous avons exploité, les mécanismes de transformation de modèles, ce choix nous a garantie la conformité de notre ontologie au métamodèle SPEM.
- L'inférence de connaissances PLs est effectuée à l'aide du langage SWRL.
- La recherche de connaissances PLs s'est faite en exploitant les principes de parcours de graphes, ce qui nous a permis d'exploiter les algorithmes déjà mis en place dans le domaine de la théories des graphes.
- L'implémentation s'est faite en utilisant le langage java (IDE Eclipse), l'intérêt est que nous avons pu bénéficier de la panoplie des plugins mises à disposition des développeurs et qui nous a faciliter le travail.

Un premier prototype a été réalisé ; le travail réalisé est très consistant, cependant, beaucoup d'améliorations restent à apporter. Pour le moment notre prototype est organisé en trois applications qui s'exécutent indépendamment :

- Une application qui capitalise les connaissances PLs, pour l'instant l'application utilise deux instanciateurs (PBOOL et EPF) et exécute les règles d'inférences identifiés.

- Une application qui fait la recherche de la configuration PL demandés (configurations Ad hoc).
- Une application qui fait le déploiement d'une architecture de PLs en ACME. La recherche des connaissances composants PL, adaptation des connaissances connecteurs PLs et les deux types de déploiements sont inclus dans cette application.

Les améliorations a apporter sont comme suit :

- De nouvelles règles d'inférences pour inférer plus de connaissances PLs.
- Décrire les contraintes et les propriétés des styles architecturaux de manière plus détaillée avec le langage Armani.
- Améliorer l'algorithme de recherche des connaissances composants en rajoutant une étape de recherche d'enchainement de Task Uses. En effet, selon SPEM une "Activité" peut être constituée de plusieurs Task Uses. L'algorithme que nous avons mis en place recherche un enchainement d'activité et si la solution n'existe pas, il ne recherche pas d'enchainement de Task Uses, ce qui peut retourner un résultat exploitable dans le déploiement.

L'étude et l'intégration de l'aspect dynamique du modèle de PL est aussi une des perspectives de recherche, et fait partie de nos objectifs à moyen terme.

Conclusion générale

La modélisation et l'amélioration des PLs est un sujet de recherche d'actualité, la concurrence technologique et commerciale entre développeurs de logiciels, les pressions du marché du logiciel (produire vite, bien, et pas cher), en plus des avancées technologiques qui doivent continuellement être pris en compte, sont autant de motivations qui incitent à l'amélioration des modèles de PLs.

"Software process are software too" : la mise en service (modélisation et exécution) d'un modèle de PL suscite autant de préoccupations que la mise en service d'un logiciel. Le sujet traité par cette thèse concerne la modélisation et l'exécution des PLs par approche de réutilisation. La maturité du domaine en termes d'expérience de développement et en termes de concepts et paradigmes nous oriente naturellement vers l'adoption d'approche de réutilisation pour la modélisation de PLs.

Le travail présenté par cette thèse constitue une contribution à la réutilisation des PLs à base d'ALs et pose les fondements de la réutilisation des PLs à base d'ontologie de domaine.

L'exploitation des ALs pour la prise en charge de l'aspect structure du modèle de PLs, combinée à l'exploitation d'une ontologie de domaine pour la prise en charge l'aspect expérience et savoir faire pour la modélisation et l'exécution des modèles de PLs est la solution adoptée. Ainsi, pour mener à bien notre travail nous avons étudié un nombre significatif de domaines :

- **Les procédés logiciels :** en plus de cerner et comprendre les concepts des PLs, nous nous somme focalisés sur les exigences et les insuffisances des modèles de PLs actuels. L'étude du domaine des PLs est présentée dans le chapitre -1-.
- **Les architectures logicielles :** dans le chapitre -2- nous nous somme intéressés aux avantages que pouvaient offrir les ALs pour promouvoir la réutilisation des PLs, par conséquent, la maîtrise du domaine des ALs était indispensable.
- **Les ontologies de domaines :** les concepts, les mécanismes d'inférences, les langages tels que OWL, SPARQL, SQWRL, SWRL...etc. ainsi que les outils existants tels que "Protégé" ont été étudiés pour la mise en place de notre ontologie de domaine. L'étude des ontologies de domaine n'étais pas aussi détaillée que celle des ALs. Au fait, l'ontologie de domaine a été traitée comme un outil pour capitaliser les connaissances PLs , ce qui explique que nous n'avons pas dédié un chapitre mais une annexe à la définition des concepts de base des ontologies de domaine. Néanmoins, il faut souligner que l'apport de l'exploitation d'une ontologie de domaine pour la réutilisation des PLs est considérable, et que l'exploration des solutions à base d'ontologies pour la modélisation des PLs fait partie de nos futurs objectifs.
- **Les approches de réutilisation de PLs à base d'ALs :** les solutions existantes qui combinent concepts PLs et concepts ALs ont été étudiées.
- **Les approches de modélisation à base d'ontologies :** les préoccupations et les solutions adoptées pour exploiter les ontologies dans le domaine des PLs ont aussi été étudiées.

Ces deux derniers points ont fait l'objet du chapitre-3-. D'autres points ont été étudiés pour mettre en place notre approche :

- **Le métamodèle SPEM :** la maîtrise du métamodèle SPEM de son organisation, l'identification des ses avantages et de ses inconvénients ont été les tâches principales de notre travail.
- **Les transformations des modèles :** Les possibilités de générer des ontologies de domaine (modèles OWL) à partir de modèles UML, ainsi, que la possibilité de générer des règles d'inférences à partir de contraintes OCL ont été étudiées.

– **La théorie des graphes :** nous nous somme intéressés plus particulièrement aux algorithmes de parcours et de recherche qui sont offerts dans ce domaine (algorithme de Djikstra...etc.).

Cette large étude bibliographique, nous a permis d'élargir nos connaissances et d'aiguiser notre sens de réflexion et d'analyse dans le domaine des PLs.

Contributions

L'étude de ces différents domaines, en plus de l'analyse des solutions existantes nous a permis de mettre de présenter deux contributions majeures :

Le cadre de comparaison pour les approches de réutilisation de PLs à base d'ALs

La mise en place d'un cadre de comparaison pour l'évaluation et l'analyse des approches de réutilisation de PLs à base d'ALs. Ce cadre de comparaison est structuré en trois axes :

– Un axe spécifique aux critères qui permettent d'évaluer les approches de réutilisation de PLs à base d'ALs selon la vue architecture logicielle.
– Un axe spécifique aux critères qui permettent d'évaluer qui permettent d'évaluer les approches de réutilisation de PLs à base d'ALs selon la vue procédé logiciel.
– Un axe qui permet d'évaluer l'aspect qualitatif des approches de réutilisation de PLs à base d'ALs.

Ce cadre de comparaison a les avantages suivants :

– Il est général est peut évaluer des approches qui modélisent différents catégories de PLs.
– Il peut évaluer toute approche de modélisation de PLs qui utilise les concepts des ALs même partiellement.
– Il se base sur les cadres de comparaison mis en place dans les domaines des PLs et des ALs. En effet, les critères d'évaluations (techniques ou qualitatifs) sont tirés des cadres existants, ce qui lui confère une facilité de compréhension et d'extension.

L'approche AoSP

La deuxième contribution est la mise en place une approche de réutilisation de PLs qui répond aux objectifs que nous nous somme fixés initialement. Le tableau- 5.3- résume les solutions mises en place pour atteindre nos objectifs. Ainsi, nous constatons que l'association ALs et ontologie de domaine est une combinaison gagnante, car elle permet d'améliorer de manière significative la réutilisation des PLs.

Toutefois, il faut souligner que AoSP à certains inconvénients qu'il est important de citer :

– Même si la capitalisation des connaissances est facilitée par l'utilisation d'instanciateurs et qu'elle est faite une seule fois, elle exige la présence d'un expert PLs. L'expert doit pouvoir non seulement connaitre la terminologie et le vocabulaire utilisés localement, mais surtout doit pouvoir faire les correspondances avec les connaissances capitalisées à partir des modèles de PLs externes.
– Le temps de recherche augmente avec la taille de la base de connaissances, des mécanismes ont été mis en place pour diminuer le temps de réponse, mais reste dépendant de la taille de la base de connaissances.
– La réutilisation d'un nouveau type de modèle de PL décrit dans un langage de modélisation de PLs non pris en charge par AoSP exige le développement d'un nouvel instanciateur.
– La génération d'un nouveau modèle de PL décrit dans un nouveau langage de modélisation de PLs non pris en charge par AoSP exige le développement d'un nouveau générateur de modèle de PL.

Objectif	Solution
Proposer une solution générale	Le travail réalisé est indépendant de tout type de PL.
Augmenter la réutilisation des anciens modèles de PLs.	-Utilisation d'une ontologie de domaine "SPEMOntology" qui regroupe les connaissances de la plupart des modèles de PLs existants indépendamment de leur Langage de modélisation ou métamodèle. - Inférence de nouvelles connaissances et émergence de nouvelles solutions (de nouveaux enchainements).
Augmenter la réutilisabilité des nouveaux modèles de PLs.	- Exploitation de la structure abstraite des PLs lors de la modélisation. - Définition de styles d'exécution pour pouvoir implémenter différents types d'exécution avec la même configuration. - Définition de connecteurs explicites et réutilisables. - Définition de styles de PLs pour les architectures de PLs. - Possibilité de différents types de déploiements. - Possibilité de déploiement avec différents langages de modélisation de PLs.
Augmenter la qualité du modèle de PL	- Exploitation des expériences précédentes de modélisation et d'exécution de PLs. - Meilleure exploitation de la structure abstraite du modèle de PL. - Traitement indépendant des interactions PLs. - Inférence et extraction de nouvelles solutions optimales qui répondent à la demande de l'utilisateur.

Table 5.3 Les solutions mises en place pour la réalisation de nos objectifs.

Perspectives

Les perspectives ouvertes par les travaux effectués dans le cadre de notre thèse peuvent être organisées en trois axes qui font l'objet des paragraphes suivants : perspectives à court, moyen et long terme.

Perspectives à court terme

Les perspectives à court terme relèvent de la continuité de l'approche AoSP :

- Définir de règles d'inférence à partir des contraintes du métamodèle SPEM que ce soit des contraintes décrites en OCL ou des contraintes décrites en langage naturel : ces règles d'inférence permettront d'enrichir les connaissances PLs capitalisées en se basant sur le raisonnement de SPEM qui n'oublions pas est un standard dans le domaine des PLs.
- Définir d'autres styles de PLs et expliciter en détail les propriétés des styles architecturaux à l'aide du langage de contraintes tel que Armani.
- Amélioration des algorithmes de recherche de manière à extraire des connaissances plus optimales et qui prennent en compte l'orientation du modèle de PL. Ainsi, par exemple la configuration PL recherchée peut être :
  - Orientée temps : la configuration PL résultat minimisera le temps d'exécution du modèle de PL modélisé.
  - Orientée coût : la configuration PL résultat minimisera le coût d'exécution du modèle de PL modélisé.

- Orientée qualité : la configuration PL résultat maximisera la qualité du produit résultat de l'exécution du modèle de PL modélisé.
Nous pensons qu'il est possible d'améliorer la recherche de connaissances configuration en rajoutant des paramètres qui expriment ces orientations. Les algorithmes de recherche ne changerons pas, le changement sera au niveau de la métrique de sélection "MS" qui prendra en compte ces paramètres.
- Implémenter la recherche de configuration PL qui respecte un ou plusieurs styles architecturaux.
- Implémenter les autres types de déploiements : les déploiements implémentés jusqu'à présent sont le déploiement global (toute la configuration PL est déployée en une seule fois) et le déploiement partiel (le déploiement de la configuration PL se fait composant par composant). Les déploiements incrémentaux et itératifs qui reflètent les spécificités des méthodes agiles font partie de nos futurs travaux.

Perspectives à moyen terme

L'étude d'un nombre important de solutions de réutilisation nous a contraint dés le début à donner la priorité à certains objectifs ; les autres font partie des perspectives à moyen terme, nous citons :

- L'aspect dynamique du modèle de PL n'a pas été abordé dans cette thèse, néanmoins, les fondements ont été posés pour leur prise en charge, en effet, de la même manière que pour l'aspect structurel du PL qui a été abordé dans cette thèse, il est possible de décrire l'aspect dynamique du PL à travers :
  - La description de règles SWRL qui permettent d'expliciter, d'inférer les comportements des PLs au niveau de l'ontologie de domaine.
  - L'intégration d'un ADL qui permet d'expliciter la dynamicité des ALs tels que wright [5] au niveau de l'architecture logicielle.
- La mise en place de modules de transformation qui permet de transformer la description du comportement dynamique (décrit à l'aide ADL wright par exemple) en une représentation exploitable dans le domaine des PLs (diagramme état transition ou réseau de pétri par exemple).
- La vérification et la validation de l'architectures de PLs : selon des métriques qui doivent être définies pour évaluer la qualité de l'architectures de PLs.

Perspectives à long terme

La définition d'une sémantique d'architecture de PLs qui adhère à l'esprit des ALs ouvre la porte à un large éventail de perspectives de recherche, ainsi, les axes de recherche ouverts dans le domaine des ALs peuvent être translatés directement au domaine des architectures de PLs, nous citons :

- L'analyse et la validation des architectures de PLs.
- L'évolution des architectures de PLs,

Aussi, cette nouvelle vision des PLs crée de nouveaux axes de recherche tels que la définition de métriques et de modèles de qualité spécifiques à l'architecture de PLs qui permettront d'évaluer la qualité d'une architecture de PL.

Finalement, l'intégration des ALs comme paradigme de modélisation des PLs à part entière, a permis de donner une vision inédite et innovante non seulement à la modélisation des PLs, mais aussi à leur exécution. Après l'étude et la reconnaissance du potentiel des ALs, il est certain que les architectures logicielles constituent des solutions prometteuses aux préoccupations actuelles du domaine des PLs.

Annexes

ANNEXE A

Le métamodèle SPEM

A.1 Les métamodèles de SPEM

A.1.1 Package Noyau (Core)

Le package "CORE " permet à l'utilisateur de SPEM d'abstraire les classes de base des PLs pour permettre la description de toutes les autres classes SPEM. Le nombre des classes de ce package est restreint, ils décrivent les concepts de base de tout PL, à savoir : l'unité de travail, l'enchainement des unités de travail, le réalisateur des unités de travail et tous élément qui peut aider à dérailler cette enchainement.

A.1.2 Package Structure de procédé (Process Structure)

Contient les éléments de base pour la structuration d ?un PL. Le noyau de ce package est la représentation d ?un fragment procédé "Breakdown Element" définit indépendamment de toute méthode ou cycle de vie de logiciel. Un élément fragment peut être l'abstraction d ?un enchainement d'activités récurrent et réutilisable et peut prendre en compte d'autres concepts PL.

Les fragments de PLs définis avec la structure "fragment" de ce package peuvent être utilisé pour décrire des patrons de procédés ou des PLs ah doc qui ne font référence à aucune méthode particulière. Les meilleurs exemples sont : les procédé SCRUM, les procédés itératifs et incrémentaux utilisés souvent dans les développements logiciels qui se basent sur l'auto-organisation des équipes.

A.1.3 Package Method Content (contenu méthode)

Ce package fourni les concepts de base pour définir des méthodes, techniques, meilleures pratiques de développement réutilisables indépendamment d'un projet de développement ou d'une structure du PL particulière. Il définit et décrit les éléments de base de chaque méthode tels que rôle, tâche et Work Product...etc. Ainsi, l ?utilisateur peut réutiliser ces éléments contenu méthode en les organisant en séquences semi ordonnées puis les personnaliser selon son projet.

A.1.4 Package Process With Method (Procédé avec Méthode)

Ce package définit des PLs en intégrant des instances du package Process Structure avec des instances du package Method Content. Ce package permet de décrire un PL structuré qui respecte les concepts du package "Process Structure", et en même temps, suit une méthode de développement définit dans le package "Method Content", plaçant le PL dans un contexte particulier et en respectant une méthode ou cycle de vie particulier.

A.1.5 Package Managed Content (Contenu géré)

Les processus de développement sont souvent représentés sous forme de modèles mais également sous forme de documents et de descriptions informelles. Pour la plupart des méthodes de développement, la documentation textuelle est aussi importante que la modélisation de la méthode elle-même. Aussi, certaines méthodes considèrent que les meilleures pratiques de développement sont souvent communiquées à travers des descriptions en langage naturelle sur le format papier, ce qui est le cas des méthodes agiles tel que la méthode XP (eXtreme Programming).

Ce package fourni les concepts de base pour gérer les descriptions textuelles des meilleures pratiques, la documentation et les guidances nécessaires aux PLs. L'avantage de ce package est qu'il est possible d'utiliser ses concepts de manière autonome ou de les associer aux concepts du package "Process Structure" et du package "Method Content"".

A.1.6 Package process behavior (Comportement de processus)

L'objectif de ce package n'est pas de fournir les concepts de base pour décrire ses propre modèle comportementaux mais de capturer des modèles comportementaux externes déjà prédéfinit dans des modèles de PL qui ne sont pas forcément conformes au metamodel SPEM. Il fournit les outils de mapping entre les concepts des PL existants et les éléments des packages "Process Structure" et "Method Content".

Ce package permet, d'une part, la réutilisation de PLs existants qui ne sont pas conforment à SPEM, et d'autres part, une grande flexibilité en permettent à l'utilisateur SPEM de choisir une approche de modélisation de comportement qui lui convient.

A.1.7 Package Method Plugin (Plugin de méthode)

Ce package définit des mécanismes d'extension et de variabilité des packages Method Content, Process Structure et Process with Method.

L'objectif de ce package est de fournir les outils pour étendre et réutiliser les connaissances des packages Method Content et Process Structure et Process with Method. Le principe de base de la réutilisation et d'extension se base sur les notions de "Method Configuration" et "Method Plugin".

Une Method Configuration est un ensemble de Method Plugin. Une Method Plugin décrit une association entre un élément Method Content et un élément Process Structure. Elle décrit aussi les changements à opérer sur ces éléments pour pouvoir les réutiliser et sans les modifier directement.

Aussi, Method Plugin intègre la réutilisation à base de composants en définissants des classes décrivant certains concepts architecturaux tels que : Process Component, Work Product Port et Work Product Port Connector. Cependant, les travaux sur la réalisation à base de composant sont à leur début des problèmes d'assemblage et de connexion entre composants procédés sont soulevés et doivent être traités [16].

ANNEXE B

Les ontologies de domaine : Notions et concepts

B.1 Notion d'ontologie

B.1.1 Origine des ontologies en informatique

De [1] :

En intelligence artificielle dans le cadre de l'ingénierie des connaissances, les systèmes experts n'avaient pour objectif que la résolution automatique de problèmes par exemple le diagnostic. Par la suite sont apparus les systèmes à base de connaissances (SBC) qui permettent le stockage, la consultation, la modification de connaissances, le raisonnement automatique sur les connaissances stockées et avec le développement des réseaux, le partage des connaissances entre les systèmes informatiques. L'expérience de leur développement a toutefois montré que la construction d'une base de connaissances était un processus complexe et nécessitait un temps considérable.

De ce fait, le souhait des développeurs est de pouvoir réutiliser et partager des bases de connaissances ou des parties de ces dernières. Plusieurs scénarios ont été envisagés : réutiliser une base de connaissances pour étendre la classe de problèmes pouvant être résolus au sein d'un SBC ou pour concevoir un nouveau système réalisant des tâches différentes mais dans le même domaine, ou bien au cours d'un raisonnement faire appel à des compétences d'autres systèmes.

Cependant, ces scénarios se heurtent à plusieurs barrières, les SBC sont représentés dans des langages très différents, à la fois par leurs structures de données (par exemple langages de règles, langage à objet) et leurs mécanismes d'inférences. Aussi, indépendamment du langage de représentation utilisé, les représentations manipulent souvent des termes différents, même dans le cas de SBC réalisant des tâches similaires. De plus il n'est pas certain que ces mêmes termes soient employés dans le même sens. D'ailleurs ce sens n'est habituellement pas représenté explicitement.

La réutilisation et le partage des bases de connaissances est donc une question difficile. Au début des années 90, des chercheurs réunis au sein du projet américain Knowledge Sharing Effort, décident de s'attaquer au problème en privilégiant la représentation explicite du sens. Ils nomment ontologie une telle représentation.

Pour cela, la représentation des connaissances sous forme de règles logiques, utilisée dans les systèmes experts ne suffit plus. Pour modéliser la richesse sémantique des connaissances, de nombreux formalismes ont été introduits, qui représentent les connaissances au niveau conceptuel, tels que les langages à base de frames, les logiques de description...etc. Ces langages permettent de représenter les

concepts sous-jacents à un domaine de connaissances, les relations qui les lient et la sémantique de ces relations indépendamment de l'usage que l'on souhaite établir sur ces connaissances.

B.1.2 Définitions d'une ontologie

Le terme ontologie est d'origine grecque, il a vu le jour dans le domaine de la philosophie, où il signifie : explication systématique de l'existence, l'étude de ce qui existe dans le monde [2]. Par la suite le terme a été utilisé en informatique et en science de l'information où une ontologie est une représentation des connaissances du monde au niveau conceptuel qui par la suite sera employée pour permettre le raisonnement automatique sur les objets du domaine concerné.

Les ontologies ont pour but de comprendre la connaissance dans un domaine, d'une façon générale et de fournir une représentation communément acceptée qui pourra être réutilisée et partagée par diverses applications. [2]

- Réutilisable : car cette représentation est faite de façon déclarative c'est-à-dire sans lien avec la manière dont ces connaissances vont être utilisées.
- Partageable : car cette représentation est établie sur la base des concepts qui caractérise un domaine ainsi que sur des concepts fondamentaux communs qui sont utilisable à travers divers domaines, ce qui permet la communication entre les systèmes d'informations qui doivent partager des informations basées sur des concepts communs. [3]

B.1.2.1 Étude descriptive

Il n'existe pas une définition unifiée pour les ontologies, beaucoup ont été proposées, nous en citons quelques unes :

- Définition 1 : En 1993, GRUBER formule la définition suivante qui deviendra par la suite la définition la plus célèbre et la plus utilisée : "une ontologie est une spécification explicite d'une conceptualisation" [4]. Dans cette définition :

- Conceptualisation correspond à un modèle abstrait à base de concepts et relation entre concepts des phénomènes du monde.
- Explicite signifie que le modèle en question doit être décrit de façon non ambiguë dans un langage formel pour pouvoir être manipulé par un agent logiciel ou par un agent humain.

- Définition 2 : "Une ontologie est un ensemble de termes structurés de façon hiérarchique, conçu afin de décrire un domaine et qui peut servir de charpente à une base de connaissance". [5]

- Définition 3 : Une ontologie peut être construite en partant d'une base de connaissances qui sera raffinée et enrichie de nouvelles définitions si de nouvelles applications sont créées. Une définition a été proposée dans ce sens : "Une ontologie fournit le moyen de décrire de façon explicite la conceptualisation des connaissances représentées dans une base de connaissances" [5].

- Définition 4 : "Un ontologie est un modèle conceptuel spécifique élaboré dans le domaine de la gestion du savoir. Une ontologie peut représenter des relations complexes entre des objets et inclure les règles et axiomes manquants dans un réseau sémantique. Une ontologie qui décrit le savoir dans un domaine précis est souvent reliée à des systèmes de prospection de données et de gestion des connaissances." [2].

Définition 5 : Bien après, ROCHE a donné une définition générique et simple : "Une ontologie est une conceptualisation d'un domaine à laquelle sont associés un ou plusieurs vocabulaires de termes. Les concepts se structurent en un système et participent à la signification des termes. Elle est définie pour un objectif donné et exprime un point de vue partagé par une communauté. Une ontologie s'exprime

dans un langage (représentation) qui repose sur une théorie (sémantique) qui garantit des propriétés de l'ontologie en termes de consensus, cohérence, partage et réutilisation" [6].

En résumé :

Une ontologie est une représentation des connaissances d'un domaine au niveau conceptuel, elle regroupe les concepts représentatifs d'un domaine donné, leurs relations et la sémantique des concepts et des relations. C'est une représentation non ambiguë et formelle manipulable par des systèmes informatiques et indépendante d'une application particulière afin de permettre sa portabilité d'une application à une autre dans le contexte du domaine modélisé : elle est réutilisable et partageable.

B.2 Constituants d'une ontologie

Une ontologie constitue un modèle de données représentatif d'un ensemble de concepts dans un domaine donnée, réel ou imaginaire ainsi que les relations entre ces derniers. Les concepts sont organisés dans un graphe dont les relations peuvent être sémantiques ou de subsomption.

B.2.1 Concepts

B.2.1.1 Description d'un concept [1]

Un concept peut représenter un objet, une notion ou une idée. Il peut se définir comme une entité composée de trois éléments :

- terme (ou plusieurs) exprimant le concept en langue.
- Une notion également appelée intension (avec 's'), contenant la sémantique du concept, exprimée en terme de propriétés et d'attributs, de règles et de contraintes. Les règles qui décrivent des inférences possibles sont des affirmations sous la forme : antécédent -> conséquent.
- Un ensemble d'objets appelé aussi extension du concept, regroupant les objets manipulés à travers le concept : ces objet sont appelés instances du concept.

Exemple : Soit le concept Voiture :

Terme : Voiture.

Intension : Véhicule de transport automobile conçu et aménagé pour le transport d'un petit nombre de personnes.

o Règle : Une voiture rare est chère.

Extension : Liste de toutes les voitures du monde.

Deux extensions peuvent ne pas être disjointes alors que deux intensions s'excluent mutuellement par au moins une propriété.

Deux concepts partageant la même extension sans avoir la même intension, peuvent être désignés par un même terme ; ceci correspond à des points de vue différents sur un même objet. Exemple : Les chiens peuvent être considérés comme des animaux de compagnie ou comme des sources culinaires.

Les concepts sont désignés par les nombreux termes que contient le langage naturel. Il se pourrait qu'un terme désigne plusieurs concepts différents d'où la présence d'une ambiguïté.

Exemple : Le terme table pour un meuble et table pour tableau de valeurs numériques. La machine, où on identifie généralement un concept à partir de ses termes, ne gère pas ces ambiguïtés. Mais la restriction à un domaine de connaissances permet d'éviter la homonymie de concepts c'est-à-dire des concepts différents désignés par un même terme, par contre afin d'assurer une grande souplesse d'utilisation de l'ontologie, il serait souhaitable de permettre la désignation d'un concept par plusieurs termes.

Un concept peut être défini à partir d'autres concepts. Exemple : Le concept de table peut se définir en utilisant les concepts meuble, plateau et pieds.

B.2.1.2 Propriétés d'un concept

Propriété Description Généricité Un concept est générique s'il n'admet pas d'extension.

Exemple : La vérité prise dans le sens de ce qui est vrai et non pas du degré de vérité. Identité Un concept porte une propriété d'identité si cette propriété permet de conclure que deux instances de ce concept sont identiques. Exemple : Le concept étudiant porte une propriété d'identité liée au numéro de l'étudiant. Rigidité Un concept est rigide si toute instance de ce concept en reste instance dans tous les mondes possibles. Exemple : Le concept humain est un concept rigide, étudiant est un concept non rigide. L'anti-rigidité Un concept est anti-rigide si toute instance de ce concept est essentiellement définie par son appartenance à l'extension d'un autre concept. Exemple : Le concept étudiant est un concept anti-rigide car un étudiant est avant tout un humain. Unité Un concept est un concept unité si pour chacune de ses instances, les différentes parties de l'instance sont liées par une relation qui ne lie pas d'autres instances de concepts. Exemple : Les deux parties d'un couteau, manche et lame sont liées par la relation emmanché qui ne lie que cette lame et ce manche. Tableau 3.1 : Propriétés d'un concept.

B.2.2 Relations

Certain liens conceptuels existants entre les concepts peuvent être exprimés par les propriétés portant sur les concepts, d'autres doivent être représentés à l'aide de relations autonomes.

Comme pour les concepts, les relations peuvent être spécifiées par des propriétés, elles sont organisées de manière hiérarchisée à l'aide de la propriété de subsomption.

B.2.2.1 Relation de subsomption

Les concepts dans un domaine de connaissances sont manipulés au sein d'un réseau de concepts. Ces derniers sont structurés hiérarchiquement et sont liés par des propriétés conceptuelles. La propriété utilisée pour structurer la hiérarchie des concepts est la subsomption qui lie deux concepts.

Un concept C1 subsume un concept C2 si toute propriété sémantique de C1 est aussi une propriété sémantique de C2, c'est à dire C2 est plus spécifique que C1 et l'extension de C2 est forcément plus réduite que celle de C1, par contre son intension est plus riche.

B.2.2.2 Liens conceptuels

Une relation permet de lier des instances de concepts ou des concepts génériques. Elle est caractérisée par : Un terme (ou plusieurs). Une signature qui précise le nombre d'instances de concepts que la relation lie, leurs types et l'ordre des concepts, c'est à dire la façon dont la relation doit être lue.

3.2.2.2. Propriétés d'une relation Propriété Description Cardinalité nombre possible de relations de ce type entre les même concepts ou instances de concepts. Exemple : Une pièce a au moins une porte. Propriétés algébriques symétrie, réflexivité, transitivité. Tableau 3.2 : Propriétés d'une relation.

3.2.2.3. Propriétés liant deux relations Propriétés Description Incompatibilité Deux relations sont incompatibles si elles ne peuvent lier les mêmes instances de concepts. Exemple : Les relations être violet et être blanc sont incompatibles. Inverse Deux relations binaires sont inverses l'une de l'autre, si quand l'une lie deux instances I1 et I2 et l'autre lie I2 et I1. Exemple : les relations a pour père et a pour enfant sont inverses l'une de l'autre. Tableau 3.3 : Propriétés liant deux relations.

3.2.2.4 Propriétés liant deux concepts Propriété Description Equivalence Deux concepts sont équivalents s'ils partagent la même extension. Exemple : étoile du matin et étoile du soir qui font référence à Venus. Disjonction Deux concepts sont disjoints si leurs extensions sont disjointes. Exemple : les concepts homme et femme. Dépendance Un concept C1 est dépendant d'un concept C2 si pour toute instance de C1 il existe une instance de C2. Exemple : le concept parent est dépendant de enfant (et viceversa) Tableau 3.4 : Propriétés liant deux concepts.

B.2.3 Fonctions

Les fonctions sont des cas particuliers de relations dans lesquelles le nième élément de la relation est défini à partir des n-1 premiers. [2] Exemple : La fonction mère-de ou carré-de.

B.2.4 Axiomes

De [2] et [9] : Les axiomes sont utiles à la formulation de phrases qui sont toujours vraies. Ils permettent de : Contraindre les valeurs de classes ou d'instances. Représenter les intensions des concepts et des relations dans un domaine. Spécifier la façon dont les concepts d'un domaine et les relations qui les lient peuvent être utilisés afin d'exprimer des connaissances dans le domaine. Certains axiomes sont particuliers, ils se retrouvent dans plusieurs ontologies. Exemple : Soit l'axiome : La relation parent de est l'inverse de la relation enfant de. Certains sont propres à un domaine, ils participent à la définition de la sémantique du domaine. Exemple : Dans le domaine de la géométrie, soit l'axiome : Il n'existe pas plus d'une droite à laquelle appartiennent deux points A et B.

B.2.5 Instances

Les instances sont utilisées pour représenter des éléments dans un domaine. [2]

B.3 Étude comparative de l'ontologie

Après l'étude descriptive nous présentons une étude comparative entre les ontologies, les thésaurus, les bases de connaissances, les bases de données et enfin avec une hiérarchie de classes dans le paradigme orienté objet.

B.3.1 Ontologies et thésaurus

Un terme et un concept peuvent être vus comme similaires en ce que chacun d'eux est composé d'un nom et d'une signification. La différence est que lorsqu'on parle d'un terme son nom est une question importante, alors que lorsqu'on parle d'un concept son nom n'a aucune importance, un concept est indépendant de la façon dont on le nomme.

Un vocabulaire est une liste de termes qui ont été définis pour des besoins particuliers. Il peut avoir une forme plus complexe, cette complexité ne réside pas dans la difficulté de son utilisation mais dans sa spécialisation pour les besoins d'un domaine particulier.

Un vocabulaire contrôlé est une liste de termes définie par un groupe (communauté de pratiques) afin de pouvoir classifier et partager l'information entre les membres de ce groupe.

La signification des termes n'est pas forcément définie et dépend du groupe qui l'utilise. Pas d'organisation logique entre les termes. Le vocabulaire contrôlé peut être structuré en une hiérarchie simple pour obtenir ce qu'on appelle une taxonomie.

Cette hiérarchisation correspond à une spécialisation on passe donc d'un vocabulaire contrôlé à un vocabulaire organisé car le lien qui existe entre un terme et ses enfants ajoute une signification.

Un thésaurus est une taxonomie. Il permet d'obtenir une spécialisation des termes grâce à la taxonomie et il permet de donner des informations qui sont en rapport avec les termes employés ce qui élargit le champ de connaissance.

D'autre part il peut le restreindre en utilisant des liens de spécialisation /généralisation. De ce fait, un thésaurus ou une taxonomie sont des formes d'ontologies dont la grammaire n'a pas était formalisée, la grammaire ici comporte un ensemble de règles qui montrent comment utiliser les termes du vocabulaire pour exprimer la sémantique de quelque chose.

Une ontologie correspond donc à un vocabulaire contrôlé et organisé et à la formalisation explicite des relations existantes entre les différents termes du vocabulaire.

B.3.2 Ontologies et bases de connaissances

Une ontologie définit un ensemble de concepts d'un domaine appelés classes. Pour certains chercheurs les instances des classes ne font pas partie de l'ontologie, pour d'autres, l'ontologie plus l'ensemble des instances des classes qu'elles décrivent forme une base de connaissances. Il devrait y avoir une différence claire entre ontologie et base de connaissances à partir de son rôle : [8]

- Une ontologie fournit un système de concepts qui sont utilisés pour construire une base de connaissances et elle spécifie aussi les contraintes entre ces concepts pour garantir deux propriétés qui sont nécessaires pour la réutilisation et le partage des connaissances : la transparence et la consistance de la base, c'est donc un méta-système d'une base de connaissances.
- Une ontologie n'est pas utilisée directement pour la résolution de problèmes ; elle fournit une spécification des connaissances et des modèles dans le système, tandis qu'une base de connaissances est utilisée pour la résolution des problèmes.

B.3.3 Ontologies et hiérarchie de classes dans le paradigme orienté objet

Une ontologie et une hiérarchie de classes dans le paradigme orienté objet sont similaires dans la méthodologie de développement au niveau supérieur et diffèrent au niveau inférieur car l'ontologie se concentre sur les aspects déclaratifs alors que la hiérarchie se concentre sur les aspects liés à la performance : [8]

- Dans le paradigme orienté objet, la signification des classes et les relations entre classes sont intégrées de façon implicite et procédurale c'est-à-dire dire leurs spécifications contiennent la façon dont ces classes doivent être utilisées.
- Dans le paradigme ontologique, dans la plupart des cas les descriptions sont faites de façon explicite et déclarative, les descriptions des classes et des relations entre classes sont faites indépendamment de leurs exploitations ultérieures, afin de permettre leurs réutilisation et d'assurer les caractères : formel et explicite.

B.3.4 Ontologies et schémas de base de données

Une ontologie et une base de données sont similaires en ce que chacune d'elles fournit une conceptualisation explicite qui permet de décrire la sémantique des données ou connaissances d'un domaine. Elles diffèrent dans les points suivants : [5]

- Le langage utilisé pour définir les ontologies est syntaxiquement et sémantiquement plus riche qu'un langage de bases de données car :

- Avec une base de données, on a une sémantique opérationnelle de traitement des données.
- Avec une ontologie, on a la possibilité de tenir des raisonnements indépendamment de l'utilisation effective des données de l'ontologie (inférence).

– L'information décrite par une ontologie relative à des concepts et relations sémantiques organisés en un réseau est plus riche que celle des tables de bases de données. Car les concepts représentatifs des connaissances d'un domaine décrivent non seulement la structure de ces dernières mais aussi leur sens.
– Une ontologie fournit une théorie de domaine et non la structure de stockage de données.

La liste des publications

Revue Internationale avec comité de lecture

– Fadila Aoussat, Mourad Oussalah, Mohamed Ahmed-Nacer, SPEMOntology for software process reusing, Computing and Informatics Journal, 2012 (à paraitre). *http ://www.cai.sk/ojs/index.php/cai*

Conférences Internationales avec comité de lecture

– Fadila Aoussat, Mohamed Ahmed-Nacer : Reusing heterogeneous software process models. IEEE Symposium on Computers and Communications (ISCC 2009), 291-294, 2009.
http ://www.ieee-iscc.org/2009/

– Fadila Aoussat, Mohamed Ahmed-Nacer, Mourad Oussalah : Reusing Approach for Software Processes based on Software Architectures. 12th International Conference on Enterprise Information Systems (ICEIS ?2010), 8 - 12 June, 2010, Madeira Portugal. 366-369. 2010.
http ://www.iceis.org/CEIS2010/

– Fadila Aoussat, Mohamed Ahmed Nacer, Mourad Oussalah, Approach for software processes models reusing based software architecture. The 15th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI'10,), Orlando, USA, 2010.

– Fadila Aoussat, Mourad Oussalah, Mohamed Ahmed-Nacer, SPEM Extension with software process architectural concepts, The 35th Annual International Computer Software and Applications Conference (Compsac 2011), Munich, Germany 215-223, 2011.
http ://compsac.cs.iastate.edu/ ?y=3

– Fadila Aoussat, Mourad Oussalah, Mohamed Ahmed-Nacer, A Domain Ontology for Software Process Architecture Description, 7th International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE 2012), 2012. *http ://www.enase.org/*

Conférence nationale avec comité de lecture

– Fadila Aoussat, Mourad Oussalah, Mohamed Ahmed-Nacer, Méta modélisation architecturale des procédés logiciels, 5ème Conférence francophone sur les architectures logicielles (CAL2011). 2011. *http ://cal2011.redcad.org/*

Bibliographie

[1] Projet ACCORD.
Etat de l'art sur les langages de description d'architecture (adls), 2002.
Disponible à l'adresse
http://www.infres.enst.fr/projets/accord/lot1/lot_1.1-2.pdf.
2, 27, 43

[2] Silvia T. ACUNA, Santiago ESTERO et Xavier FERRE.
Software process modelling.
2, 8, 8, 9, 9, 9, 15, 35, 41

[3] Silvia Teresita ACUÑA et Graciela Elisa BARCHINI.
The socio-cultural environment in the software process modeling.
Dans *SCCC*, pages 216–, 1999.
15

[4] Jonathan ALDRICH, Craig CHAMBERS et David NOTKIN.
Archjava: connecting software architecture to implementation.
Dans *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/581339.581365.
23, 25

[5] Robert John ALLEN.
A formal approach to software architecture.
Thèse de Doctorat, Pittsburgh, PA, USA, 1997.
AAI9813815.
25, 27, 132

[6] Ilham ALLOUI et Flávio OQUENDO.
Supporting decentralised software-intensive processes using zeta component-based architecture description language.
Dans *ICEIS (1)*, pages 207–215, 2001.
16, 38, 49, 51, 51, 63

[7] Adel ALTI, Tahar KHAMMACI et Adel SMEDA.
Integrating software architecture concepts into the mda platform with uml profile.
J. of Computer Science, 3 (10):793–802, 2010.
63

[8] Vincenzo AMBRIOLA, Reidar CONRADI et Alfonso FUGGETTA.
Assessing process-centered software engineering environments.
ACM Trans. Softw. Eng. Methodol., 6(3):283–328, juillet 1997.
Disponible à l'adresse
http://doi.acm.org/10.1145/258077.258080.
14, 15, 35, 35

[9] Fadila AOUSSAT, Mohamed AHMED-NACER et Mourad OUSSALAH.
Reusing approach for software processes based on software architectures.
Dans *ICEIS*, pages 366–369, 2010.
62

[10] Fadila AOUSSAT, Mourad OUSSALAH et Mohamed AHMED-NACER.
Spem extension with software process architectural concepts.
Dans *COMPSAC*, pages 215–223, 2011.
78, 78, 165

[11] Fadila AOUSSAT, Mourad OUSSALAH et Mohamed AHMED-NACER.
Domain ontology for software process description.
Dans *ENASE 2012 - 7th International Conference on Evaluation of Novel Software Approaches to Software Engineering*, 2012.
3, 61

[12] Atlas LINA GROUP.
ATL: Atlas transformation language, atl user manual, version 0.7, nantes,, 2006.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?Formal/2008-04-01.
89, 91, 92

[13] Atlas LINA GROUP.
Atl transformations list, 2006.
Disponible à l'adresse
http://www.eclipse.org/m2m/atl/atlTransformations/.
91, 91, 91, 92

[14] Standards AUSTRALIA.
Australian standard 4651-2004: Standard metamodel for software development methodologies, 2004.
Disponible à l'adresse
http://130.203.133.150/showciting;jsessionid=668063C51A9AC35FDAD887974E6CDA3B?cid=10336799.
10

[15] Denis AVRILIONIS, Noureddine BELKHATIR et Pierre-Yves CUNIN.
A unified framework for software process enactment and improvement.
Dans *4th International Conference on the Software Process*, page 102, Washington, DC, USA, 1996. IEEE Computer Society.
36

[16] Muhammad Ali BABAR, Liming ZHU et Ross JEFFERY.
A framework for classifying and comparing software architecture evaluation methods.
Dans *Proceedings of the 2004 Australian Software Engineering Conference*, ASWEC '04, pages 309–, Washington, DC, USA, 2004. IEEE Computer Society.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=987680.987740.
2, 43

[17] Len BASS, Paul CLEMENTS et Rick KAZMAN.
Software architecture in practice.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

22, 27

[18] Kent BECK, Mike BEEDLE, Arie van BENNEKUM, Alistair COCKBURN, Ward CUNNINGHAM, Martin FOWLER, James GRENNING, Jim HIGHSMITH, Andrew HUNT, Ron JEFFRIES, Jon KERN, Brian MARICK, Robert C. MARTIN, Steve MELLOR, Ken SCHWABER, Jeff SUTHERLAND et Dave THOMAS.
Manifesto for agile software development, 2001.
Disponible à l'adresse
http://agilemanifesto.org/.
15, 69

[19] Noureddine BELKHATIR et Jacky ESTUBLIER.
Supporting reuse and configuration for large scale software process models.
Dans *10th International Software Process Workshop*, page 35. IEEE Computer Society, 1996.
37, 49, 53

[20] Reda BENDRAOU et Marie-Pierre GERVAIS.
A framework for classifying and comparing process technology domains.
Dans *Proceedings of the International Conference on Software Engineering Advances*, ICSEA '07, pages 5–, Washington, DC, USA, 2007. IEEE Computer Society.
Disponible à l'adresse
http://dx.doi.org/10.1109/ICSEA.2007.3.
15, 35

[21] Reda BENDRAOU, Marie-Pierre GERVAIS, Xavier BLANC et Jean-Marc JÉZÉQUEL.
Vers l'Exécutabilité des Modèles de Procédés Logiciels.
Disponible à l'adresse
http://hal.inria.fr/inria-00371227.
1, 15, 16

[22] Reda BENDRAOU, Andrey SADOVYKH, Marie-Pierre GERVAIS et Xavier BLANC.
Software process modeling and execution: The uml4spm to ws-bpel approach.
Dans *EUROMICRO-SEAA*, pages 314–321, 2007.
9, 9

[23] Jesal BHUTA, Barry BOEHM et Steven MEYERS.
Process Elements: Components of Software Process Architectures, volume 3840, pages 332–346. Springer Berlin/Heidelberg, 2005.
16

[24] Pam BINNS, Matt ENGLEHART, Mike JACKSON et Steve VESTAL.
Domain-specific software architectures for guidance, navigation and control.
International Journal of Software Engineering and Knowledge Engineering, 6(2):201–227, 1996.
Disponible à l'adresse
http://dblp.uni-trier.de/db/journals/ijseke/ijseke6.html#BinnsEJV96.
27

[25] Barry BOEHM.
Software process architectures.
Dans *Software Architecture Workshop, 17th ICSE*, ICSE'95, 1995.
v, v, 21, 69, IV, IV

[26] Barry BOEHM, Alexander EGYED, Julie KWAN, Dan PORT, Archita SHAH et Ray MADACHY.

Using the winwin spiral model: A case study.
Computer, 31:33–44, July 1998.
2, 62

[27] Barry BOEHM et Steven WOLF.
An open architecture for software process asset reuse.
Dans *Proceedings of the 10th International Software Process Workshop*, 1996.
13, 32

[28] Barry W. BOEHM.
A spiral model of software development and enhancement.
Computer, 21:61–72, May 1988.
16, 32, 69

[29] Barry W. BOEHM, Chris ABTS, A. Winsor BROWN, Sunita CHULANI, Bradford K. CLARK, Ellis HOROWITZ, Ray MADACHY, Donald J. REIFER et Bert STEECE.
Software Cost Estimation with COCOMO II.
Prentice Hall, 2000.
68, 72, 72

[30] Beatriz Terezinha BORSOI et Jorge Luis Risco BECERRA.
A method to define an object oriented software process architecture.
Software Engineering Conference, Australian, 0:650–655, 2008.
53

[31] Eric BRUNETON, Thierry COUPAYE, Matthieu LECLERCQ, Vivien QUÉMA et Jean-Bernard STEFANI.
The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems.
Softw. Pract. Exper., 36(11-12):1257–1284, septembre 2006.
Disponible à l'adresse
http://dx.doi.org/10.1002/spe.v36:11/12.
25

[32] S. James CHOI et Walt SCACCHI.
Modeling and simulating software acquisition process architectures.
Journal of Systems and Software, 59(3):343–354, 2001.
35, 39, 49, 51

[33] Benoît COMBEMALE, Xavier CRÉGUT, Alain CAPLAIN et Bernard COULETTE.
Towards a rigorous process modeling with spem.
Dans *ICEIS (3)*, pages 530–533, 2006.
11

[34] Benoît COMBEMALE.
Approche de métamodélisation pour la simulation et la vérification de modèle : Application à l?ingénierie des procédés.
Thèse de Doctorat, 2008.
8, 8

[35] Benoît COMBEMALE.
Ingénierie dirigée par les modèles (idm) État de l?art.
Rapport technique, Toulouse, France, 2008.
7, 165

[36] M. COSSENTINO, S. GAGLIO, B. HENDERSON-SELLERS et Seidita V..
A metamodelling-based approach for method fragment comparaison.
Dans *11 The International Workshop on Exploring Modeling Methods In Systems Analysis and Design EMMSAD 06*, 2006.
5, 76

[37] Bernard COULETTE, Xavier CRÉGUT, Tran Dan THU et Dong Thi Bich THUY.
To component engineering of reuse process.
Génie logciel, pages 8–18, 1995.
38

[38] Bernard COULETTE, Tran Dan THU, Xavier CRÉGUT et Dong Thi Bich THUY.
Rhodes, a process component centered software engineering environment.
Dans *ICEIS*, pages 253–260, 2000.
38, 49, 76

[39] Xavier CRÉGUT et Bernard COULETTE.
Pbool: an object-oriented language for definition and reuse of enactable processes.
Software - Concepts and Tools, 18(2):47–62, 1997.
49, 89, 99, 100

[40] Bill CURTIS, Marc I. KELLNER et Jim OVER.
Process modeling.
Commun. ACM, 35(9).
8, 14

[41] Ligia da MOTTA SILVEIRA BORGES et Ricardo de ALMEIDA FALBO.
Managing software process knowledge.
Dans *Proc. of the International Conference on Computer Science, Software Engineering, Information Technology, e- Business, and Applications,*, 2002.
57

[42] Fei DAI, Tong LI, Na ZHAO, Yong YU et Bi HUANG.
Evolution process component composition based on process architecture.
Dans *International Symposium on Intelligent Information Technology Application Workshops*, pages 1097–1100, 2008.
39, 67

[43] Samir DAMI, Jacky ESTUBLIER et Mahfoud AMIOUR.
APEL: a graphical yet executable formalism for process modeling.
Automated Software Engineering, 5:61–96, 1997.
37, 53, 53, 63, 67, 67, 67

[44] EPF.
Eclipse process framwork, 2007.
Disponible à l'adresse
http://epf.eclipse.org/uploads/14.pdf.
89, 99, 102

[45] Roy Thomas FIELDING.
Architectural styles and the design of network-based software architectures.
Thèse de Doctorat, University of California, Irvine, USA, 2000.
21, 22

[46] Brian FITZGERALD.

Software crisis 2.0.
IEEE Computer, 45(4):89–91, 2012.
21

[47] Medina-Domínguez FUENSANTA, Saldaña-Ramos JAVIER, Mora-Soto ARTURO, Sanz-Esteban ANA et Segura Maria Isabel SÁNCHEZ.
Dans José CORDEIRO, Boris SHISHKOV, Alpesh RANCHORDAS et Markus HELFERT, réds., *ICSOFT (PL/DPS/KE)*.
16

[48] Alfonso FUGGETTA.
Software process: a roadmap.
Dans *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, 2000.
2, 5, 8, 41, 42, 43

[49] Lizbeth GALLARDO.
Une approche à base de composants pour la modélisation des procédés logiciels.
Thèse de Doctorat, 2000.
8, 16, 41

[50] David GARLAN.
Software architecture: a roadmap.
Dans *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/336512.336537.
22, 30

[51] David GARLAN, Robert ALLEN et John OCKERBLOOM.
Exploiting style in architectural design environments.
SIGSOFT Softw. Eng. Notes, 19(5):175–188, décembre 1994.
Disponible à l'adresse
http://doi.acm.org/10.1145/195274.195404.
30

[52] David GARLAN, Robert ALLEN et John OCKERBLOOM.
Architectural mismatch or why it's hard to build systems out of existing parts.
Dans *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 179–185, New York, NY, USA, 1995. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/225014.225031.
27

[53] David GARLAN, Robert T. MONROE et David WILE.
Foundations of component-based systems.
chapitre Acme: architectural description of component-based systems, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=336431.336437.
25

[54] David GARLAN et Dewayne E. PERRY.
Special Issue on Software Architecture.

IEEE, 1995.
22, 22

[55] David GARLAN et Mary SHAW.
An introduction to software architecture.
Rapport technique, Pittsburgh, PA, USA, 1994.
23, 23, 27, 70

[56] David GARLAN et Zhenyu WANG.
Acme-based software architecture interchange.
Dans *Proceedings of the Third International Conference on Coordination Languages and Models*, COORDINATION '99, pages 340–354, London, UK, 1999. Springer-Verlag.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=647015.713291.
25, 25, 26, 27, 28, 165

[57] Kevin GARY, Timothy E. LINDQUIST, Harry KOEHNEMANN et Jean-Claude DERNIAME.
Component-based software process support.
Dans *ASE*, pages 196–199, 1998.
37

[58] Kevin A. GARY et Timothy E. LINDQUIST.
Cooperating process components.
Dans *23rd International Computer Software and Applications Conference*, COMPSAC '99, pages 218–, Washington, DC, USA, 1999. IEEE Computer Society.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=645981.674454.
37

[59] Kevin A. GARY et Timothy E. LINDQUIST.
Distributed architectures for process component support.
Dans *Proceedings of the 5th Int. Conf. on Information Systems Analysis and Synthesis*, ISAS'99, 1999.
37

[60] L. GROENEWEGEN et G. ENGELS.
Reuse of software process fragments is reuse of software too.
Dans *10th International Software Process Workshop*, page 68, Washington, DC, USA, 1996. IEEE Computer Society.
13

[61] Jianying HE, Haihua YAN, Chao LIU et Maozhong JIN.
A framework of ontology supported knowledge representation in software process., 2007.
57, 58

[62] Arthur S. HITOMI, Gregory Alan BOLCER et Richard N. TAYLOR.
Endeavors: a process system infrastructure.
Dans *19th international conference on Software engineering*, 1997.
36

[63] Charlotte HUG.
Méthode, modèles et outil pour la méta-modélisation des processus d'ingénierie de systèmes d'information.
Thèse de Doctorat, Université Joseph Fourier- Grenoble I , France, 2009.

Disponible à l'adresse
http://tel.archives-ouvertes.fr/docs/00/43/76/92/PDF/These_Charlotte_Hug.pdf.
9, 9, 9, 10

[64] Watts S. HUMPHREY et Marc I. KELLNER.
Software process modeling: principles of entity process models.
Dans *Proceedings of the 11th international conference on Software engineering*, ICSE '89, pages 331–342, New York, NY, USA, 1989. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/74587.74631.
15

[65] Adrian IACOVELLI et Carine SOUVEYET.
Framework for agile methods classification.
Dans *MoDISE-EUS*, pages 91–102, 2008.
5, 15, 69

[66] International Organization for STANDARDIZATION.
Quality management systems fundamentals and vocabulary.
Geneva, Switzerland, 2000.
Disponible à l'adresse
http://www.iso.org/iso/qmp.
14

[67] JENA.
Jena api tutorial, 2007.
Disponible à l'adresse
http://jena.sourceforge.net/tutorial/index.html.
97

[68] Derniame J.C. KABA A.B., TANKOANO J..
Une approche incrémentale d'évolution des modèles de procédés de développement de logiciels.
Dans *Proceedings of the second African Conference on research in computer science*, CARI'94, pages 351–365, 1994.
9, 14, 16

[69] M. I. KELLNER.
Connecting reusable software process elements and components.
Dans *ISPW '96: Proceedings of the 10th International Software Process Workshop*, page 12, Washington, DC, USA, 1996. IEEE Computer Society.
16

[70] Marc I. KELLNER, Peter H. FEILER, Anthony FINKELSTEIN, Takuya KATAYAMA, Leon J. OSTERWEIL et Maria H. PENEDO.
Ispw-6 software process example.
Dans *1st International Conference on the Software Process*, pages 176–186. IEEE Computer Society, 1991.
66

[71] Fabien LEYMONERIE.
ASL: un langage et des outils pour les styles architecturaux ? contribution à la description d'architectures dynamiques.

Thèse de Doctorat, 2004.
22, 30, 30, 31

[72] Li LIAO, Yuzhong QU et Hareton K. N. LEUNG.
A software process ontology and its application.
Dans *Workshop on Semantic Web Enabled Software Engineering(SWESE)*, 2005.
57, 58

[73] Jacques LONCHAMP.
A structured conceptual and terminological framework for software process engineering.
Dans *ICSP*, pages 41–53, 1993.
8, 41

[74] David C. LUCKHAM et James VERA.
An event-based architecture definition language.
IEEE Transactions on Software Engineering, 21(9):717–734, 1995.
27

[75] Jeff MAGEE et Jeff KRAMER.
Dynamic structure in software architectures.
SIGSOFT Softw. Eng. Notes, 21(6):3–14, 1996.
25, 27

[76] Nenad MEDVIDOVIC, Paul GRÜNBACHER, Alexander EGYED et Barry W. BOEHM.
Bridging models across the software lifecycle.
J. Syst. Softw., 68:199–215, December 2003.
38, 49, 51, 53

[77] Nenad MEDVIDOVIC, David S. ROSENBLUM et Richard N. TAYLOR.
A language and environment for architecture-based software development and evolution.
Dans *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
Disponible à l'adresse
http://portal.acm.org/citation.cfm?id=302405.302410.
25, 27

[78] Nenad MEDVIDOVIC et Richard N. TAYLOR.
A framework for classifying and comparing architecture description languages.
Dans *ESEC / SIGSOFT FSE*, pages 60–76, 1997.
2, 24, 27, 27, 29, 30, 43

[79] Nenad MEDVIDOVIC et Richard N. TAYLOR.
A classification and comparison framework for software architecture description languages.
IEEE Transactions on Software Engineering, 26:70–93, 2000.
24

[80] Nikunj R. MEHTA, Nenad MEDVIDOVIC et Sandeep PHADKE.
Towards a taxonomy of software connectors.
Dans *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, 2000.
27, 43, 77

[81] Oren MISHALI et Shmuel KATZ.
Using aspects to support the software process: Xp over eclipse.

Dans *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 169–179, New York, NY, USA, 2006. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/1119655.1119678.
35

[82] R.T. MONROE.
Capturing software architecture design expertise with armani.
Rapport technique, School of Computer Science, Carnegie Mellon University, 2001.
Disponible à l'adresse
http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-163R.pdf.
30

[83] Verónica ÑAUPAC, Robert ARISACA et Abraham DÁVILA.
Software process improvement and certification of a small company using the ntp 291 100 (moprosoft).
Dans *PROFES*, pages 32–43, 2012.
18

[84] R. Mehta NIKUNJ et N MEDVIDOVIC.
Understanding software connector compatibilities using a connector taxonomy.
Dans *First Workshop on Software Design and Architecture,*, 2002.
27

[85] Elisabetta Di NITTO, Luigi LAVAZZA, Marco SCHIAVONI, Emma TRACANELLA et Michele TROMBETTA.
Deriving executable process descriptions from uml.
Dans *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 155–165, New York, NY, USA, 2002. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/581339.581361.
15

[86] OMG Object Management GROUP.
Meta Object Facility mof 2.0 core specification, 2006.
Disponible à l'adresse
http://uml3.ru/content/omg-spec/mof/06-01-01.pdf.
10

[87] OMG Object Management GROUP.
Object Constaints Language, 2006.
Disponible à l'adresse
http://www.omg.org/spec/OCL/2.0.
91

[88] OMG Object Management GROUP.
Software Systems Process Engineering Metamodel, v2.0, 2008.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?Formal/2008-04-01.
5, 10, 10, 10, 12, 53, 67, 77, 77, 77, 78, 96, 99, 165, 166

[89] Martin. J. O'CONNOR et A. K. DAS..

Sqwrl: a query language for owl.
Dans *OWL: Experiences and Directions (OWLED), 6th International Workshop,*, 2009.
104

[90] The Institute of ELECTRICAL et Eletronics ENGINEERS.
Ieee standard glossary of software engineering terminology, 1990.
14, 17

[91] olivier LE GOAER.
Styles d'évolution dans les architectures logicielles.
Thèse de Doctorat, 2009.
21, 23, 29

[92] OMG.
Object management group.
6

[93] Object Management Group OMG.
Ontology definition metamodel, 2008.
Disponible à l'adresse
http://www.omg.org/cgi-bin/doc?Formal/2008-04-01.
76, 91

[94] Object Management Group OMG.
Uml: Unified modeling language, 2008.
Disponible à l'adresse
http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.
25, 76

[95] OPF.
Open process framework repository organization., 2005.
Disponible à l'adresse
http://www.opfro.org/.
10

[96] Leon OSTERWEIL.
Software processes are software too.
Dans *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, 1987.
v, v, 13, IV, IV

[97] Mourad OUSSALAH.
Ingénierie des composants logiciels , Principes et fondements.
Vuibert, 2005.
27, 29, 45, 45

[98] Mourad Chabane OUSSALAH, Adel SMEDA et Tahar KHAMMACI.
Software connectors reuse in component-based systems.
Dans *IRI*, pages 543–550, 2003.
27

[99] Jorge Enrique PÉREZ-MARTÍNEZ.
Heavyweight extensions to the uml metamodel to describe the c3 architectural style.
SIGSOFT Softw. Eng. Notes, 28:5–5, May 2003.
63

[100] Dewayne E. PERRY et Alexander L. WOLF.

Foundations for the study of software architecture.
SIGSOFT Softw. Eng. Notes, 17(4):40–52, octobre 1992.
22

[101] F. PLÁSIL, D. BÁLEK et R. JANECEK.
Sofa/dcup: Architecture for component trading and dynamic updating.
Dans *Proceedings of the International Conference on Configurable Distributed Systems*, CDS '98, pages 43–, Washington, DC, USA, 1998. IEEE Computer Society.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=582972.792780.
25

[102] Juergen RILLING, Yonggang ZHANG, Wen Jun MENG, René WITTE, Volker HAARSLEV et Philippe CHARLAND.
A unified ontology-based process model for software maintenance and comprehension.
Dans *Proceedings of the 2006 international conference on Models in software engineering*, MoDELS'06, pages 56–65, Berlin, Heidelberg, 2006. Springer-Verlag.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=1762828.1762839.
57

[103] Nassima SADOU.
Evolution Structurelle dans les Architectures Logicielles à base de Composants.
Thèse de Doctorat, Université de Nantes , France, 2004.
Disponible à l'adresse
http://tel.archives-ouvertes.fr/docs/00/48/80/05/PDF/these-nassima-sadou-2007.pdf.
23

[104] Sonia Djamal SANLAVILLE.
Environnement de procédé extensible pour l?orchestration Application aux services web.
Thèse de Doctorat, 2005.
9

[105] Walt SCACCHI.
Process Models in Software Engineering, J.J. Marciniak (ed.), Encyclopedia of Software Engineering,.
John Wiley and Sons, Inc, New York,, 2001.
5, 5, 5, 6, 6, 14, 165

[106] Bradley SCHMERL et David GARLAN.
Acmestudio: Supporting style-centered architecture development.
Dans *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 704–705, Washington, DC, USA, 2004. IEEE Computer Society.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=998675.999479.
26

[107] Mary SHAW, Robert DELINE, Daniel V. KLEIN, Theodore L. ROSS, David M. YOUNG et Gregory ZELESNIK.
Abstractions for software architecture and tools to support them.
IEEE Transactions on Software Engineering, 21:314–335, 1995.

23, 25, 27

[108] Adel SMEDA.
méta-modélisation dans le domaine de l?architecture logicielle.
Thèse de Doctorat, 2006.
21, 25, 26, 27, 28, 28, 29, 29, 30, 163

[109] ISO/IEC Software ENGINEERING.
Metamodel for development methodologies. iso/iec 24744, 2007.
Disponible à l'adresse
http://www.iso.org/iso/catalogue_detail.htm?csnumber=38854.
10

[110] Ian SOMMERVILLE et Tom RODDEN.
Human, social and organisational influences on the software process, 1996.
Disponible à l'adresse
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.8957.
2, 14, 15, 42

[111] Gokhan Halit SOYDAN et Mieczyslaw M. KOKAR.
An owl ontology for representing the cmmi-sw model.
2006.
Disponible à l'adresse
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.153.9122.
58

[112] Fritz STALLINGER, Alec DORLING, Terence P. ROUT, Brian HENDERSON-SELLERS et Bruno LEFEVER.
Software process improvement for component-based software engineering: An introduction to the oospice project.
Dans *EUROMICRO*, pages 318–323, 2002.
10

[113] Gernot STARKE.
Why is process modelling so difficult?
Dans *EWSPT '94: Proceedings of the Third European Workshop on Software Process Technology*, pages 163–166, London, UK, 1994. Springer-Verlag.
1

[114] Muhammad SULAYMAN, Cathy URQUHART, Emilia MENDES et Stefan SEIDEL.
Software process improvement success factors for small and medium web companies: A qualitative study.
Inf. Softw. Technol., 54(5):479–500, mai 2012.
Disponible à l'adresse
http://dx.doi.org/10.1016/j.infsof.2011.12.007.
18

[115] Roh SUNGHWAN, Kim KYUNGRAE et Jeon TAEWOONG.
Architecture modeling language based on (uml2.0).
Dans *11th Asia-Pacific Software Engineering Conference*, pages 663–669. IEEE Computer Society, 2004.

63

[116] Clemens SZYPERSKI.
Component Software: Beyond Object-Oriented Programming.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002.
22, 26, 26, 26

[117] Hanh Nhi TRAN.
Modélisation de Procédés Logiciels à base de patrons réutilisables.
Thèse de Doctorat, université de Toulouse, France, 2007.
16, 32, 69

[118] Dan TURK, Robert FRANCE et Bernhard RUMPE.
Limitations of agile software processes.
Dans *Third International Conference On Extreme Programming And Flexible Processes In Software Engineering (XP2002)*, pages 43–46. Springer-Verlag, 2000.
1, 15

[119] Alf Inge WANG.
A process centred environment for cooperative software engineering.
Dans *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 469–472, New York, NY, USA, 2002. ACM.
Disponible à l'adresse
http://doi.acm.org/10.1145/568760.568841.
35

[120] Takahira YAMAGUCHI, Satoshi KOMORI, Kaori MORI et Tomohiko SHIOZAWA.
A process-centered software engineering environment using ontologies.
Dans *IEICE TRANSACTIONS on Information and Systems*, MoDELS'06, pages 1387–1393, Berlin, Heidelberg, 1998. Springer-Verlag.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=1762828.1762839.
56, 58

[121] Takami YAMAGUCHI..
Modeling software processes by using process and object ontologies.
Dans *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE '97, pages 319–, Washington, DC, USA, 1997. IEEE Computer Society.
Disponible à l'adresse
http://dl.acm.org/citation.cfm?id=786767.786791.
56

[122] Kamal Zuhairi ZAMLI.
Process modeling languages: A literature review.
Malaysian Journal of Computer Science, 14(2):26–37, 2001.
8

[123] Kamal Zuhairi ZAMLI.
A survey and analysis of process modeling languages.
Malaysian Journal of Computer Science, 17(2):68–89, 2004.
2, 9, 41

[124] Kamal Zuhairi ZAMLI et Peter LEE.
Taxonomy of process modeling languages.

Computer Systems and Applications, ACS/IEEE International Conference, 0:0435, 2001.
9, 15, 35, 41, 42, 42

[125] Apostolos ZARRAS, Valérie ISSARNY, Christos KLOUKINAS et Viet Khoi NGUYEN.
Towards a base uml profile for architecture description.
Dans *Proceedings of the ICSE Workshop on Architecture and UML*, pages 22–26. IEEE Computer Society, 2001.
63

Liste des tableaux

— *Corps du document* —

2.1 Correspondances adoptées entre les caractéristiques exigées par les PLs et les avantages offerts par des ALs. 24
2.2 Domaine d'application et élément architecturaux de base des principaux ADLs [108]. 25
2.3 Les correspondances adoptées entre les concepts de base des PLs et les concepts de base des ALs. 33

3.1 Les critères d'évaluation selon l'axe PL. 41
3.2 Les critères d'évaluation selon l'axe AL. 43
3.3 Les critères d'évaluation selon l'axe qualité. 46
3.4 Résumé des approches de réutilisation de PLs à base d'ALs étudiées. 48
3.5 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe PL. 50
3.6 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe AL. 52
3.7 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe qualité. 55
3.8 Les ontologies de domaine utilisées dans les approches de réutilisation de PLs à base d'ontologies. 58

4.1 Les correspondances adoptées entre les concepts de PLs et les concepts ALs. 64
4.2 Les caractéristiques des styles architecturaux de PLs. 70
4.3 Description du style structurel en V. 73
4.4 La description du style structurel UP. 75
4.5 Les types de déploiement offerts par l'approche AoSP. 84
4.6 Évaluation de l'approche AoSP selon l'axe PL. 85
4.7 Évaluation de l'approche AoSP selon l'axe AL. 86
4.8 Évaluation de l'approche AoSP selon l'axe qualité. 86

5.1 Les correspondances adoptées entre les concepts PBOOL et les concepts de SPEMOntology. 101
5.2 Les concepts et les object Type properties de l'ontologie temporaire. 116
5.3 Les solutions mises en place pour la réalisation de nos objectifs. 131

— *Annexes* —

Liste des figures

— *Corps du document* —

1.1 Cycle de vie du logiciel Waterfall (Royce 1970) [105]. 6
1.2 L'architecture à 4 niveaux de l'OMG [35]. 7
1.3 La méta modélisation des procédés logiciels. 7
1.4 Le procédé logiciel (le monde réel du développement). 9
1.5 Le Métamodèle noyau du procédé logiciel. 10
1.6 Les métamodèles de SPEM. 12
1.7 Les stéréotypes représentant les concepts abstraits de chaque métamodèle de SPEM. 13

2.1 Les éléments architecturaux de base d'une architecture logicielle [56]. 28
2.2 Les éléments architecturaux de base d'une architecture logicielle décrit à l'aide de L'ADL ACME. 28
2.3 Les propriétés et les contraintes d'une AL décrites à l'aide de L'ADL ACME. 29
2.4 Description du style client-serveur à l'aide de L'ADL ACME. 31

3.1 Les axes d'évaluation des approches de réutilisation de PLs à base d'ALs. 40
3.2 Approches de réutilisation de PLs à base d'ALs étudiées. 47

4.1 Les étapes de l'ingénierie "pour" la réutilisation de PLs. 63
4.2 Le métamodèle d'architecture de procédé logiciel. 65
4.3 L'architecture de PL selon la sémantique adoptée. 66
4.4 Architecture de PL décrite avec l'ADL ACME (Vue partielle exemple ISPW). 67
4.5 La taxonomie des connecteurs de procédés logiciels. 69
4.6 Le style structurel en V. 73
4.7 Le style structurel UP. 74
4.8 Le style structurel UP decrit avec l'ADL ACME. 75
4.9 Les concepts architecturaux du métamodèle SPEM [88]. 77
4.10 Extension du profil Method Plugin avec les concepts architecturaux manquants [10]. 78
4.11 Extension du modèle SPEM avec les concepts architecturaux manquants. 79
4.12 Organisation de la capitalisation des connaissances PLs. 81
4.13 Les étapes ingénierie "pour" la réutilisation des PLs. 82
4.14 L'organigramme global suivi pour l'extraction des connaissances architecture de PL. 83

5.1 Les étapes de réalisation de l'approche AoSP. 90
5.2 Les transformations ATLs appliquées pour la génération de SPEMOntology. 92
5.3 Le module de transformation ATL ApplySPEMProfil2SPEMModel. 93
5.4 Entête et helpers du module ApplySPEMProfil2SPEMModel. 93
5.5 Les modèles sources du module ApplySPEMProfil2SPEMModel décrit en UML. 94
5.6 La méthode *applyProfil()* appliquée dans le module ApplySPEMProfil2SPEMModel. 94
5.7 La méthode *applyStereotype()* appliquée dans le module ApplySPEMProfil2SPEMModel. . 95

5.8 Exemples d'instructions ATLs qui permettent d'intégrer les Tagged Values 96
5.9 La structure de SPEMOntology (vue classe UML vue concept OWL). 97
5.10 Les concepts utilisés pour capitaliser les connaissances des architectures de PLs. 98
5.11 Concepts utilisés pour capitaliser le vocabulaire référence. 98
5.12 Quelques associations SPEM exploitées pour gérer les correspondances entre les connaissances PLs [88]. 99
5.13 Librairie qui permet de sauvegarder les informations des instanciateurs. 100
5.14 Exemple de code PBOOL. 100
5.15 Organigramme décrivant les étapes d'extraction des connaissances PL à partir de modèles PBOOL. 101
5.16 Le résultat de l'instanciation de SEMOntologie avec un modèle PBOOL. 102
5.17 Modèle de PL décrit à l'aide EPF Composer (version XML). 103
5.18 Modèle de PL décrit à l'aide EPF Composer (version graphique). 103
5.19 Les étapes d'instanciation de SPEMOntology à partir de modèles EPF. 104
5.20 Les étapes principales de recherche de connaissances configuration PL. 108
5.21 Les étapes de constitution de l'ensemble des configurations PL des solutions potentielles. . . 110
5.22 Exemple de recherche de connaissances configuration PL à partir d'une ensemble de configurations. 111
5.23 Les étapes de constitution et d'optimisation de la solution (Configuration PL) finale. 112
5.24 Exemple de recherche de connaissances configuration PL à partir d'une ensemble de configurations (étape -4-). 113
5.25 Les étapes de marquage ascendant et descendant des composants. 114
5.26 Les étapes de recherche de la solution optimale a partir d'un ensemble de composants. . . . 114
5.27 Les étapes de recherche de connaissances configuration PL qui respecte un ou plusieurs styles à partir d'un ensemble de configurations PLs. 115
5.28 Les étapes de recherche de connaissances configuration PL qui respecte un ou plusieurs styles à partir d'un ensemble de composants PLs. 116
5.29 Les étapes de l'algorithme de recherche de connaissances composant PL. 117
5.30 Exemple de recherche des connaissances composant PL. 118
5.31 Le résultat final après optimisation. 120
5.32 Le résultat de l'algorithme de recherche de connaissances de composants PL. 120
5.33 Déploiement d'architectures de PLs. 121
5.34 Les structures des fichiers XML des connecteurs PLs. 122
5.35 Déploiement Total d'architectures de PLs. 124
5.36 Les connaissances du composant spécification. 125
5.37 Les connaissances du composant conception. 125
5.38 Résultat du déploiement Total d'architectures de PLs. 126

— *Annexes* —

Table des matières

— Corps du document —

Introduction générale **1**

1 Les procédés logiciels **5**
1.1 Introduction 5
1.2 L'origine des Procédés Logiciels 5
1.2.1 Le modèle de cycle de vie de logiciel 6
1.2.2 Le modèle de procédé logiciel 6
1.2.3 Le modèle de processus logiciel 6
1.3 Modélisation et Méta Modélisation des procédés logiciels 6
1.3.1 Le niveau M0 : Les procédés logiciels 8
1.3.2 Le niveau M1 : Les modèles de procédés logiciels 8
1.3.3 Le niveau M2 : Les métamodèles de procédés logiciels 9
1.3.4 Le niveau M3 : Les méta-métamodèles de procédés logiciels 10
1.4 Le métamodèle SPEM (Systems and Software Process Engineering Metamodel) 10
1.4.1 Noyau conceptuel de SPEM 11
1.4.2 Les métamodèles de SPEM 11
1.4.3 Organisation des classes de SPEM à travers les métamodèles 12
1.5 Les singularités du modèle de procédé logiciel 13
1.5.1 Les différences entre produit logiciel et procédé logiciel. 13
1.5.2 Les caractéristiques difficilement accordables du modèle de procédé logiciel 14
1.5.3 Les autres caractéristiques des procédés logiciels 16
1.6 La modélisation des procédés logiciels par approche de réutilisation 16
1.6.1 Les difficultés de réutilisation spécifiques aux procédés logiciels 16
1.6.2 Ingénierie pour la réutilisation des Procédés Logiciels 17
1.6.3 Ingénierie par la réutilisation des Procédés Logiciels 18
1.7 Conclusion : synthèse et analyse 18

2 Les architectures logicielles au service de la réutilisation des procédés logiciels **21**
2.1 Introduction 21
2.2 Les architectures logicielles 22
2.2.1 Définitions 22
2.2.2 Les avantages des architectures logicielles 22
2.2.3 L'apport des architectures logicielles aux procédés logiciels 23
2.3 Les Langages de Description d'Architectures (ADLs) 23
2.3.1 Définition 23
2.3.2 Les caractéristiques des ADLs 23
2.3.3 Exemple d'ADL : ACME 25
2.3.4 ACME au service des procédés logiciels 26

2.4 Définition des éléments architecturaux de base 26
2.4.1 Le composant 26
2.4.2 Le connecteur 27
2.4.3 La configuration 27
2.4.4 L'interface 27
2.5 Les caractéristiques communes aux éléments architecturaux de base 29
2.5.1 Les propriétés 29
2.5.2 Les contraintes 29
2.5.3 Les sémantiques 30
2.5.4 Les types 30
2.6 Les styles architecturaux 30
2.6.1 Les objectifs des styles architecturaux 31
2.6.2 Les avantages des styles architecturaux 32
2.6.3 Les styles architecturaux et les procédés logiciels 32
2.7 Conclusion : synthèse et analyse 32

3 Évaluation des approches de réutilisation de procédés logiciels à base d'architectures logicielles 35
3.1 Introduction 35
3.2 Les approches de réutilisation de Procédés Logiciels à base d'architectures logicielles 36
3.2.1 Les approches de réutilisation de procédés logiciels centrées composants 36
3.2.2 Les approches de réutilisation de procédés logiciels centrées connecteurs 38
3.2.3 Les approches de réutilisation de procédés logiciels centrées configurations 39
3.3 Cadre de comparaison pour les solutions de réutilisation de modèles de procédés logiciels à base d'architectures logicielles 39
3.3.1 Les critères d'évaluation selon l'axe procédé logiciel 40
3.3.2 Les critères d'évaluation selon l'axe architecture logicielle 43
3.3.3 Les critères d'évaluation selon l'axe qualité 45
3.4 Évaluation des approches de modélisation et d'exécution de PLs à base d'ALs 47
3.4.1 Résumé des approches de réutilisation de procédés logiciels à base d'architectures logicielles 47
3.4.2 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe PL 49
3.4.3 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe AL 51
3.4.4 Évaluation des approches de réutilisation de PLs à base d'ALs selon l'axe qualité 53
3.4.5 Bilan et discussions 56
3.5 La réutilisation de PLs à base d'ontologies de domaine 56
3.5.1 Les approches de réutilisation à base d'ontologies de domaine 56
3.5.2 Analyse des approches de réutilisation à base d'ontologies de domaine 58
3.6 Conclusion 59

4 L'approche AoSP (Architecture oriented Software Process) 61
4.1 Introduction 61
4.2 L'approche Architecture oriented Software Process (AoSP) 61
4.3 Ingénierie pour la réutilisation de procédés logiciels 62
4.3.1 La sémantique adoptée pour la notion d'architectures de procédés logiciels 63
4.3.2 Les connecteurs de Procédés Logiciels 67
4.3.3 Les styles architecturaux de PLs 69

4.3.4 La conception de l'ontologie des procédés logiciels 75
4.4 Ingénierie par la réutilisation des procédés logiciels 81
4.4.1 La recherche et acquisition des connaissances de l'architecture de PL 82
4.4.2 Le déploiement de l'architecture de procédé logiciel 84
4.5 Évaluation de l'approche AoSP 85
4.5.1 Évaluation de l'approche AoSP selon l'axe procédé logiciel 85
4.5.2 Évaluation de l'approche AoSP selon l'axe architecture logicielle 85
4.5.3 Évaluation de l'approche AoSP selon l'axe qualité 86
4.6 Conclusion 87

5 Implémentations et expérimentations 89
5.1 Introduction 89
5.2 Description des étapes de l'implémentation de l'approche AoSP 89
5.3 Ingénierie pour la réutilisation des PLs : Préparer l'ontologie de domaine. 91
5.3.1 La génération de SPEMOntology 91
5.3.2 Le module ATL ApplySPEMProfil2SPEMModel 92
5.3.3 La structure de l'ontologie SPEMOntology 96
5.3.4 La capitalisation des connaissances procédés logiciels 97
5.3.5 L'inférence et l'enrichissement des connaissances de SPEMOntology 104
5.4 Ingénierie par la réutilisation des PLs : réutilisation effective des connaissances capitalisées . 108
5.4.1 La recherche et l'acquisition des connaissances d'une configuration PL 108
5.4.2 La recherche et l'acquisition des connaissances des composants PLs 114
5.4.3 Le déploiement d'une architecture de PL 121
5.5 Conclusion 126

Conclusion générale 129

— *Annexes* —

A Le métamodèle SPEM 135
A.1 Les métamodèles de SPEM 135
A.1.1 Package Noyau (Core) 135
A.1.2 Package Structure de procédé (Process Structure) 135
A.1.3 Package Method Content (contenu méthode) 135
A.1.4 Package Process With Method (Procédé avec Méthode) 135
A.1.5 Package Managed Content (Contenu géré) 136
A.1.6 Package process behavior (Comportement de processus) 136
A.1.7 Package Method Plugin (Plugin de méthode) 136

B Les ontologies de domaine : Notions et concepts 137
B.1 Notion d'ontologie 137
B.1.1 Origine des ontologies en informatique 137
B.1.2 Définitions d'une ontologie 138
B.2 Constituants d'une ontologie 139
B.2.1 Concepts 139

B.2.2 Relations 140
B.2.3 Fonctions 141
B.2.4 Axiomes 141
B.2.5 Instances 141
B.3 Étude comparative de l'ontologie 141
B.3.1 Ontologies et thésaurus 141
B.3.2 Ontologies et bases de connaissances 142
B.3.3 Ontologies et hiérarchie de classes dans le paradigme orienté objet 142
B.3.4 Ontologies et schémas de base de données 142

— *Pages annexées* —

Bibliographie **147**

Liste des tableaux **163**

Liste des figures **165**

Table des matières **167**

Réutilisation des procédés logiciels :

Une approche à base d'architectures logicielles

Fadila AOUSSAT

Résumé

Boehm [25] met en évidence la dualité produit logiciel/procédé logiciel concernant les architectures logicielles. En se basant sur l'article *"Software Processes Are Software Too"* d'Osterweil [96], il confirme que si les architectures logicielles sont efficaces pour la réutilisation des produits logiciels, elles seront d'une réelle contribution pour la réutilisation des procédés logiciels. *"If open architectures are good for software product reuse, then their process counterparts will be good for software process reuse"*. Nos travaux se référent donc à la réutilisation des Procédés Logiciels (PLs) en se basant sur le paradigme d'architecture Logicielle (AL).
Cette thèse constitue une première contribution à la réutilisation des procédés logiciels à base d'architectures logicielles, notre contribution se décline en deux points essentiels :
- La définition d'un cadre de comparaison où nous identifions les caractéristiques essentielles des procédés logiciels et les spécificités des approches de réutilisation de procédés logiciels à base d'architectures logicielles. Ce cadre de comparaison permettra de cerner, classifier et évaluer les approches de réutilisation de PLs à base d'architectures logicielles.
- L'élaboration d'une approche de réutilisation de procédés logiciels baptisée AoSP (Architecture oriented Software Process). Cette approche a pour objectif d'exploiter la réutilisation "pour" et "par" des PLs tout en les combinant aux autres opportunités de réutilisation offertes par l'exploitation d'une ontologie de domaine.

Mots-clés : Procédés logicieL (PL), réutilisation, architectures logicielles, ontologie de domaine, SPEM (System and Software Engineering Metamodel), extension de profil UML, éléments architecturaux, connecteurs PL explicites, styles de PLs, transformation de modèles, capitalisation de connaissances PLs, inférence de connaissances PLs, recherche d'architectures de PLs, déploiement d'architecture de PLs.

Software processes reusing :

An approach based on software architectures

Abstract

Boehm [25] highlights the duality between software product / software process concerning software architectures. Based on the paper *"Software Processes Are Software Too"* of Osterweil [96], he confirms that *"If open software architectures are good for product reuse, then processes Their Counterparts Will Be good for software process reuse"*. Our work therefore refer to the reuse of software processes based on the software architecture paradigm.
This thesis is a first contribution on the reuse of software processes based on software architectures ; our contribution is constituted in two main points :

- The definition of framework for comparison in which we identify the essential characteristics of software processes and the specificities of the approaches for reusing software processes based on software architectures. This framework will be used to classify and evaluate the proposed approaches for reusing software processes based on software architectures.

- The elaboration of an approach to software process reusing called AOSP (Software Process Oriented Architecture). This approach attempts to exploit "for" and "by" reusing software processes combined with other reuse opportunities offered by exploiting a domain ontology.

Keywords: Software Process (SP), reuse, Software architectures, domain ontology, SPEM (System and Software Engineering Metamodel), UML profil extention, architectural elements, Explicit SP connectors, SP styles, models transformation, SP knowledge capitalizing, SP knowledge infering, software architectures retreiving, SP architecture deployement.

Discipline : INFORMATIQUE
Spécialité : INFORMATIQUE

Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
CNRS FRE 2729. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.