THÈSE DE DOCTORAT
DES UNIVERSITÉS PIERRE & MARIE CURIE ET GASTON BERGER

SPÉCIALITÉ :
SYSTÈMES INFORMATIQUES

PRÉSENTÉE PAR
M. FALL IBRAHIMA

POUR OBTENIR LE GRADE DE
DOCTEUR DES UNIVERSITÉS
PIERRE & MARIE CURIE (PARIS VI) ET GASTON BERGER (SAINT-LOUIS)

SUJET DE LA THÈSE :

GESTION OPTIMISEE DE PRODUITS-MODÈLES DE PROCÉDÉS LOGICIELS

SOUTENUE LE 08-02-2012 DEVANT LE JURY COMPOSÉ DE :

NOM	QUALITÉ	TITRE
Marie Pierre GERVAIS	Directrice	Professeure à l'université Paris X Nanterre La Défense
Moussa LO	Co-directeur	Professeur à l'université Gaston Berger de Saint-Louis
Mireille BLAY-FORNARINO	Rapporteur	Professeure à l'université de Nice - Sophia Antipolis
Bernard COULETTE	Rapporteur	Professeur à l'université de Toulouse II
Xavier BLANC	Examinateur	Professeur à l'université Bordeaux I
Fabrice KORDON	Examinateur	Professeur à l'université P. & M. Curie
Reda BENDRAOU	Co-encadreur	McF à l'université P. & M. Curie

ECOLE DOCTORALE :
EDITE (ÉCOLE DOCTORALE INFORMATIQUE, TELECOMUNICATION, ET ELECTRONIQUE) DE PARIS

A ma mère et mon père.
A ma fille Sokhna Mame Diarra FALL.

Remerciements

L'aventure fascinante et palpitante que constitue une thèse est le fruit d'un travail mené dans un cadre stimulant, comportant d'intéressantes et déterminantes collaborations qui participent à ses fondations. Cette thèse a ainsi été rendue possible par la contribution inestimable de tous ceux qui m'ont soutenu depuis mes premiers souffles jusqu'à ce jour, et surtout d'une façon qui va au delà de ce que je peux exprimer sur cette page.

Je remercie le Seigneur Tout Puissant pour m'avoir donné la force de réaliser ce travail. Je prie sur le Sceau de Ses Prophètes (PSL).

Ma recherche ne serait certainement pas ce qu'elle est sans le soutien ô combien important des personnes suivantes. M. Xavier BLANC, Professeur à l'Université Bordeaux I, membre de mon jury de thèse, et avec qui le travail a été initié et son fondement construit. Mme Marie-Pierre GERVAIS, Professeure à l'Université Paris X et infatigable directrice de cette thèse, pour tous ses encouragements, son soutien, et sa disponibilité durant ces années. M. Reda BENDRAOU Maître de Conférence à l'Université Paris VI et co-encadrant de la thèse, pour tout le temps consacré à des discussions autour des idées qui fondent cette thèse. J'estime beaucoup vous devoir et suis très content de le dire publiquement à travers cette page. Veuillez agréez ma reconnaissance.

Je ne peux rien dire de M. Moussa LO que tout le monde ne sache déjà. Professeur à l'Université de Saint-Louis, il a été mon co-directeur de thèse. Son humanité, son humilité et la direction implicite dont j'ai bénéficié de sa part depuis même avant ce projet me resteront d'un grand apport. Merci également au Professeur Mary Teuw NIANE, recteur de l'université de Saint-Louis pour ses conseils et encouragements. Merci infiniment.

Je voudrais remercier M. Bernard COULETTE Professeur à l'Université de Toulouse II, et Mme Mireille BLAY-FORNARINO, Professeure à l'Université de Nice Sophia Antipolis, pour avoir tous les deux accepté d'être les rapporteurs de ma thèse. Je tiens aussi à remercier Monsieur Fabrice KORDON Professeur à l'Université Paris VI, et responsable de l'équipe MoVe du LIP6, pour m'avoir fait l'honneur de participer au jury de ma soutenance de thèse.

Je remercie la coopération française pour avoir financé ce travail. Merci à Nathalie Bernard et à Audrey De Forville du SCAC. Merci à Egide pour l'organisation de mes séjours parisiens.

Merci à tous les membres du groupe META de l'équipe MoVe du LIP6 de Paris. Merci à tous les autres membres de l'équipe. Merci pour leur soutien et pour tous les bons moments que nous avons passés ensemble. Merci à Lom et Nicolas GIBELIN pour la gentillesse de depuis le début. Merci à Marcos pour son aide et sa gentillesse. Merci à Laurent et Selma avec qui j'ai toujours été dans la bonne entente et la bonne humeur. Merci à tous.

Mes pensées vont à ceux qui ont fait que mes nombreux séjours à Paris se soient tous très bien passés. Merci à Mass pour tout. Merci à Aboubacry et Fatima pour l'ambiance familiale dont ils m'ont souvent fait bénéficier avec un humeur restée irréprochable. Merci à Mère B. SARR pour toute l'attention. Merci à Abass pour le soutien. Je retiendrai de vous la générosité, la gentillesse et la disponibilité qui du reste m'ont marqué à jamais. Je vous resterai infiniment reconnaissant.

Je pense particulièrement à mes amis à Dakar. Farra, Diallo, Dethié, Bachir. Vous avez toujours cru en moi et m'avez toujours encouragé. Cela m'a beaucoup aidé dans ce travail. Merci les potes.

Je tiens à vous remercier encore, Matar FALL (Limamoulaye) et Ousmane Sylla NIANG (Ndindy). Vous m'avez tellement soutenu à tous les points de vue depuis mon enfance. Vos conseils, encouragements, et toutes les discussions qu'on a à chaque occasion m'ont été très précieux dans ce travail. Encore merci mes grands frères.

Je voudrais saluer mes collègues de l'Université Cheikh Anta Diop et ceux et celles de l'ESP en particulier pour les encouragements de toutes sortes. Merci du fond de mon coeur.

Que dire encore de Cheikh Ahmadou Bamba (RA), ainsi que de toute sa famille. Je suis certain que les références que vous constituez pour moi, me guideront toujours sur la voie du travail et de la foie en Dieu et Son Prophète (PSL). Ce courage et cette foie m'ont été fondamentaux au cours des jours et nuits passés sur ce travail. Veuillez agréer la gratitude du tout petit que je suis.

MERCI à ma famille. En l'occurrence, MERCI à ma mère, MERCI à ma mère, MERCI à mon père. Je ne sais que dire d'autre; tellement votre amour, vos prières et vos conseils ont été vitaux pour ce travail, comme ils l'ont été et le resteront d'ailleurs dans toutes mes entreprises. Puisse Dieu vous donner longue vie et me permette de vous REMERCIER encore et encore et au meilleur de moi même.

Résumé

Actuellement, la communauté des procédés logiciels ne porte que peu d'attention aux artefacts de ces procédés. Leur gestion est minimale. Or l'introduction des principes de l'ingénierie des modèles dans les procédés logiciels modifie en profondeur la nature de ces artefacts et augmente leur complexité: ils deviennent des modèles (que nous appelons produits-modèles). La prise en compte de cette complexité est alors impérative en vue d'optimiser la gestion de ces produits-modèles.

Dans cette thèse, nous posons le problème de la modélisation des relations entre produits-modèles en analysant l'importance de ces relations dans la gestion de l'évolution des produits-modèles lors de l'exécution du procédé qui les utilise et/ou les produit. Ainsi nous illustrons qu'exploiter les modèles de relations à l'exécution du procédé optimise la gestion des produits-modèles en matière de cohérence, de synchronisation ou encore de flexibilité dans la granularité des produits-modèles.

Notre contribution est composée de deux éléments majeurs qui sont relatifs à la modélisation puis à l'exécution de procédés.

Du point de vue de la modélisation des procédés, nous avons proposé un méta-modèle permettant de structurer les éléments relatifs à la spécification des produits-modèles. Ce méta-modèle contient les concepts nécessaires à la définition des relations entre ces éléments et d'associer à ces relations les caractéristiques nécessaires à une meilleure gestion des produits-modèles à l'exécution des procédés modélisés. Notre approche supporte les relations d'inclusion (Nest) et de partage d'éléments (Overlap) entre deux ou plusieurs produits-modèles d'un procédé en exécution.

Du point de vue de l'exécution des procédés, dans le but de structurer les entités logiques à travers lesquelles sont gérés les produits-modèles d'un procédé modélisés avec les concepts du précédent méta-modèle, nous avons proposé un autre méta-modèle. Ce dernier contient les concepts nécessaires à la représentation des objets de procédé correspondants aux produits-modèles ainsi qu'aux relations, en conformité avec le premier méta-modèle.

Dans le but d'assurer une correspondance entre les concepts des deux méta-modèles ainsi que de permettre une exploitation automatique des concepts de modélisation à travers ceux d'exécution, la proposition comprend également des règles de transformation entre ces différents concepts.

Les apports de notre approche sont relatifs à la gestion des produits-modèles de procédés en exécution. Il s'agit d'une cohérence relationnelle systématique, d'une synchronisation également systématique, d'une construction assistée, d'une flexibilité du point de vue de la granularité, et d'une intégrité ou cohérence sémantique de ces produits-modèles.

Nous avons également conçu et réalisé un prototype qui simule un environnement de modélisation et d'exécution de procédés selon notre approche.

Mots clés

Procédés Logiciels, Produits de Procédés, Modélisation de Procédés, Exécution de Procédés, Nest, Overlap.

Abstract

Currently, software process artefacts are under-considered by the software process modelling and execution community. They are minimally managed. Moreover, the application of the principles of the model driven engineering on software processes have deeply changed the nature and increased the complexity of software artefacts : they become models (model-products hereafter). Taking into account such a complexity is a necessity with a view to optimize model-products management policies.
This thesis has focused on the issue of the specification of the relationships between model-products by analysing the importance of the use of those relationships in model-products evolution management during process execution. We therefore have illustrated that using the respective specifications of relationships during process execution optimizes the management of model-products evolution in term of their consistency, their synchronization, and a flexibility in their granularity. Our solution fits in two major points that respectively relate to process modelling and enactment.
According to process modelling, we have proposed a meta-model that captures the concepts to use to specify the model-products and the relationships between them. The meta-model takes also into account the details on these relationships as they are useful for an enhancement of used model-products management policies during the execution of the modelled processes. Our approach currently supports the *nest* and the *overlap* relationships.
According to the process execution point of view, in order to structure the logical entities through which are managed the model-products of a process, we have proposed another meta-model. Such a meta-model therefore specifies process objects that represent model-products and their relationships during process execution. This meta-model also supports the nest and the overlap relationships, in compliance with the first one.
The proposition also comprises transformation rules used to map concepts of the two meta-models and therefore to give the possibility of an automatic use of the modelling concepts through the enactment ones.
The contributions of the approach are related to model-products management during process enactment. They essentially consist of a systematic synchronization and relational consistency between model-products, their aided creation, a flexibility in the granularity of their use, and their semantic integrity.
We finally have prototyped a process modelling and enactment environment that implements our approach.

Keywords

Software Process, Process Products, Process Modeling, Process Execution, Nest, Overlap.

Table des Matières

Liste des Figures

Liste des Tableaux

Chapitre 1

Introduction Générale

1.1 Contexte et Motivations

Les applications logicielles ont conquis une place et un rôle à la fois essentielles et critiques dans notre société [Fuggetta 2000]. Elles vont des petits systèmes embarqués au sein de produits d'utilisation courante à ceux de plus grande envergure chargés par exemple, d'assurer le contrôle de centrales thermiques, de vols d'avions, etc. Ces logiciels sont donc variés, très souvent complexes et difficiles à construire, et les enjeux inhérents à la réussite de leur développement toujours énormes.

Face à cette situation, la communauté « Génie Logiciel » a adopté l'utilisation de procédés de développement logiciels comme moyen d'assurer la maîtrise des activités relatives au cycle de vie de ces systèmes et s'est inscrite dans une logique d'amélioration continue de la gestion de ces procédés [Curtis et al. 1992]. C'est ainsi que les procédés ont différemment été représentés dans le temps au sein de cette communauté. Les représentations sont parties des modèles initiaux, génériques et très abstraits de cycles de développement [Hosier 1987; Royce 1970; Boehm et al. 1998], aux modèles de procédés beaucoup plus concrets et détaillés [Scacchi 2001; Sommerville 2006]. Par définition, les modèles traditionnels de cycle de développement étaient purement conceptuels. Ils pouvaient essentiellement aider à définir l'organisation, le planning, le budget, etc. du projet à développer [Scacchi 2001]. Il a ensuite été montré qu'une représentation plus détaillée qu'un modèle de cycle de vie était nécessaire pour décrire un procédé [Curtis et al. 1992]. Dans [Sommerville 2006], Sommerville soutient qu'un modèle de procédé doit représenter de façon explicite tous les composants d'un processus de développement logiciel. La même idée a été soutenue par d'autres auteurs du domaine [Méndez Fernández et al. 2010; Silva & Oliveira 2011; Curtis et al. 1992] qui soulignent qu'un tel modèle doit couvrir plusieurs perspectives en même temps. Celles-ci peuvent être de natures fonctionnelle, organisationnelle, comportementale, informationnelle,

etc. et sont aussi importantes les unes que les autres [Méndez Fernández et al. 2010]. Dans [Silva & Oliveira 2011], les auteurs notent l'importance de la perspective informationnelle. Celle-ci est relative aux données produites, modifiées ou consommées par les activités d'un procédé et que nous appellerons « produits » dans la suite.

Les produits capturent toute l'information dont les acteurs ont besoin, ou qu'ils produisent, lors de l'accomplissement de leurs rôles respectifs au cours de l'exécution du procédé. Par exemple, une activité d'analyse d'un procédé peut produire un modèle d'analyse. Les produits peuvent être de différentes natures (code, documents, modèles, etc.) et peuvent, entre eux, avoir des relations également de natures diverses et parfois très complexes. Une mauvaise gestion des produits d'un procédé ainsi que des informations qui leur sont relatives influe largement sur la qualité des applications finales [Fuggetta 2000; Silva & Oliveira 2011]. Un modèle de procédé doit donc fournir le maximum d'information sur la structure des produits du procédé représenté de même que sur les relations entre eux [Fuggetta 2000; Curtis et al. 1992]. En effet, l'exploitation de ces informations à l'exécution du procédé contribue à assurer une gestion adéquate des produits.

Malgré leur importance, la modélisation des produits de procédés reçoit une très faible attention de la part de la communauté scientifique « procédés logiciels »; ils sont représentés et utilisés comme des documents monolithiques et très faiblement structurés [Silva & Oliveira 2011]. Dans le contexte traditionnel de la gestion de procédés, les produits sont représentés comme des données de type « fichiers » dont la structure interne est très souvent négligée. Les liens établis entre eux sont des relations classiques de composition et de dépendance sans aucune caractéristique particulière associée. Les approches traditionnelles de gestion de procédés [Sutton et al. 1990; Kaiser et al. 1990; Bandinelli et al. 1994; Belkhatir et al. 1992; Bendraou et al. 2010; Di Nitto et al. 2002; Chou 2002] se basent sur des langages qui supportent une modélisation « simpliste » de ces relations. Certaines d'entre elles sont également associées à des environnements d'exécution qui assurent une relative prise en compte de ces relations basée sur des mécanismes spécifiques aux produits d'alors, de type fichiers (textes, binaires, etc.), programmes, etc.

Les conséquences de ce manque d'attention que la communauté porte aux produits de procédés logiciels sont encore plus pénalisantes avec l'avènement et la consolidation de

l'ingénierie dirigée par les modèles (IDM). Le changement intervenu est lié aux trois principaux facteurs suivants.

- Les modèles sont devenus les citoyens de première classe dans le nouveau contexte IDM [Steel & Jézéquel 2005].
- En tant que données essentiellement manipulées dans ce contexte, les modèles ont une nature différente et plus complexe que celle des simples fichiers (de type texte pour la plupart) que l'on trouve dans l'ancien contexte.
- Les relations entre eux sont plus nombreuses, plus complexes, d'un nouveau type, et sont souvent associées à des informations caractéristiques.

Dans un tel contexte, en l'absence de toute information sur la structure des produits (qui sont des modèles) et les relations entre eux, il est difficile d'assurer leur gestion de façon optimale. Cela se justifie par le fait que, contrairement aux considérations actuelles, l'évolution des produits est fortement influencée par les relations entre eux ainsi que par leur structure interne. Par exemple, si deux produits ont en commun des éléments de modèle, l'évolution de l'un n'est que partiellement liée à celle de l'autre. De même, il arrive que l'évolution d'un produit ne soit spécifiquement relative qu'à un sous-ensemble précis de ses éléments de modèles. Par ailleurs, avec la complexité croissante des systèmes et applications logiciels, les procédés utilisés pour les développer ont considérablement changé de nature. Ils sont composés d'activités qui peuvent se dérouler de façon séquentielle, itérative, voire parfois concurrente. Ils impliquent des produits complexes, nombreux, de taille importante et fortement liés les uns aux autres, mais aussi de nombreux développeurs qui ne peuvent avoir une vue globale de tous ces produits qui représentent différents aspects du système global. Les développeurs qui jouent des rôles dans ces procédés se chargent eux-mêmes de gérer l'évolution des produits par le biais d'outils pour lesquels il n'existe aucun moyen automatique de fournir de l'information sur les relations intra et inter-produits. En conséquence, en travaillant sur un produit d'un procédé, un développeur pense toujours créer, modifier, et supprimer des éléments de modèles de ce seul produit. Or en réalité, cela n'est pas le cas si le produit en question partage des éléments de modèles avec d'autres produits du même procédé. Plus généralement, selon la nature des relations de dépendance qui s'établissent entre un produit sur lequel travaille un développeur et les autres, la propagation des actions de ce

dernier nécessite une connaissance des relations existantes entre les produits du procédé. Cette situation fait que la synchronisation des produits afin de les maintenir cohérents les uns par rapport aux autres durant l'exécution du procédé est quasiment impossible dans le contexte actuel. Les incohérences relationnelles entre les produits qui peuvent en résulter constituent une menace sur les délais de livraison du système, voire la réussite d'un projet. Elles constituent ainsi un élément auquel les entreprises doivent faire face pour sauvegarder leur compétitivité. Aussi, la taille importante des produits ajoutée à l'absence d'information sur leur structure interne pose un autre problème. En effet, lorsqu'un développeur désire travailler sur un produit d'un procédé, il est obligé de charger ce dernier en entier à l'aide de l'outil qu'il utilise pour accomplir sa tâche, même si celle-ci ne s 'intéresse qu'à un fragment précis du produit (la partie du modèle relative aux classes d'analyse, par exemple). Il est vrai que certains environnements comme ModelBus [Sriplakich et al. 2008; Sriplakich et al. 2006; SRIPLAKICH 2007] permettent un accès à des parties d'un produit de taille importante à travers des *fichiers de modèles*. Malheureusement, aucun mécanisme permettant de guider la création de *fichiers de modèles* selon les besoins d'un procédé n'est disponible dans ces environnements.

Ce sont toutes les difficultés précédentes qui traduisent l'impossibilité d'une gestion optimale des produits dans un contexte de projets d'envergure au vu de l'existant actuel. Une telle gestion des produits prendrait, en effet, en compte tous les aspects précédemment soulignés sous forme d'exemples. Cela n'est pas le cas au sein des approches actuelles de gestion de procédés dont l'inadéquation est accentuée du point de vue des produits. En effet, aucune d'entre elles n'offre à la fois les deux possibilités suivantes.

- Spécifier complètement les relations entre les modèles consommés, produits, ou modifiés par les activités de procédés dans ce nouveau contexte IDM, ainsi que les caractéristiques qui leur sont associées.
- Prendre en compte ces relations durant l'exécution des procédés. Cela est une conséquence directe du premier problème.

La situation que nous venons de décrire engendre donc un impact négatif sur le support des activités de développement. Elle ne va donc pas dans la direction tracée par la communauté scientifique qui s'intéresse aux procédés et selon laquelle il faut continuer à trouver des

méthodes et technologies contribuant davantage à l'amélioration de ce support [Fuggetta 2000; Méndez Fernández et al. 2010].

1.2 Objectifs

Face à la problématique décrite dans la sous-section précédente et que nous allons illustrer de façon plus approfondie au chapitre 2, la thèse que nous défendons s'articule autour de deux objectifs principaux qui sont associés aux dimensions principales de la gestion de procédés logiciels : la modélisation et l'exécution de ces procédés.

- **La définition des liens entre les produits d'un procédé.** Les modèles constituent l'essentiel des produits d'un procédé de développement logiciel dans le contexte IDM. Nous proposons le moyen de spécifier les relations entre eux avec tous les détails nécessaires dès la modélisation du procédé.
- **La prise en compte de ces relations au cours de l'exécution du procédé.** Une fois les relations modélisées entre les produits d'un procédé, leur exploitation par l'environnement chargé de l'exécution du procédé va permettre une amélioration de la gestion des produits. Nous visons, de ce point de vue, les améliorations qui suivent.
  - **Une cohérence systématique des modèles ayant des éléments communs.** Nous proposons de stocker en un seul exemplaire tout élément commun entre deux ou plusieurs modèles. Un modèle pourra ainsi être construit non pas toujours ex nihilo mais souvent en partant d'éléments déjà créés lors de la construction d'un ou de plusieurs autres modèle(s). Cette approche garantit la meilleure cohérence possible entre des modèles ainsi liés.
  - **Une évolution automatique et dynamique des modèles.** En effet, si deux ou plusieurs modèles se partagent des éléments, toute évolution de l'un d'eux due à un élément de la partie commune doit être automatiquement propagée sur les autres modèles. Une synchronisation systématique des modèles ainsi liés sera alors garantie tout le long de l'exécution du procédé. Nous qualifions de dynamique toute évolution

d'un produit qui est liée à celle d'un autre sans nécessiter une manipulation explicite du produit considéré. Quant à elle, l'évolution automatique d'un produit dénote son changement d'état qui n'intervient qu'après celui d'un autre, et qui nécessite une manipulation explicite du produit considéré par un moyen automatique qui se charge de propager le changement intervenu.

- **Une modularité des modèles.** Nous proposons une adaptation de la relation de composition entre les produits classiques des approches existantes au contexte IDM. Le bénéfice est une flexibilité accrue du point de vue de la granularité avec laquelle les modèles sont gérés par les environnements d'exécution des procédés.

Notre proposition devra être générique et s'adapter à tout contexte de modélisation et d'exécution de *procédé* dans le cadre IDM.

1.3 Structure du document

Le reste du présent document est organisé en cinq chapitres.

Le deuxième chapitre fournit les définitions de concepts de base. En effet, l'ingénierie des procédés logiciels est une discipline qui comporte plusieurs aspects communs avec plusieurs autres domaines. Par conséquent, il nous a paru nécessaire de fixer sans ambiguïté le sens que nous donnons à certains termes et expressions dont la bonne compréhension est essentielle au lecteur. Après ces définitions, le chapitre revient sur le contexte et la problématique qui y sont beaucoup plus clairement précisés et illustrés.

Le troisième chapitre présente l'état de l'art. Il présente le support des relations entre produits de procédés dans deux domaines connexes au thème de la thèse. Nous nous sommes intéressés aux approches de modélisation et d'exécution de procédés ainsi qu'aux systèmes de gestion d'évolution de produits.

Le quatrième chapitre présente notre proposition de solution à la problématique soulevée. Cette solution consiste à permettre, dès la modélisation d'un procédé, une spécification détaillée des relations existantes entre les produits qui sont des modèles pour l'essentiel dans ce contexte. Elle consiste ensuite à exploiter ces spécifications dans le but d'assurer une

gestion optimisée des produits à l'exécution des procédés.

Le cinquième chapitre présente un outillage de l'approche que nous avons proposée et montre sa mise en oeuvre à travers un exemple. Nous y présentons, en effet, le prototype que nous avons développé et qui représente un environnement de modélisation et d'exécution de procédés qui s'appuie sur notre proposition.

Le sixième chapitre conclut le document par une synthèse de la contribution de cette thèse. Il dresse ensuite une liste des perspectives ouvertes par le travail qui a fait l'objet de la thèse.

Chapitre 2

Prérequis, Contexte, Problématique et Illustrations

2.1 Introduction

L'ingénierie des procédés est une discipline qui se situe au carrefour de plusieurs domaines incluant le Business Process Management, le Workflow Management, les Systèmes d'Information et l'Ingénierie Logicielle. Par conséquent, il nous a paru nécessaire de fixer sans ambiguïté le sens que nous donnons à certains termes et expressions dont la bonne compréhension est essentielle à la lecture de cette thèse. C'est ainsi que dans ce chapitre l'objectif est d'abord de fournir les définitions de concepts de base, ensuite de clairement définir la problématique de la thèse ainsi que son contexte, et enfin de présenter ses principaux objectifs.

Le présent chapitre est structuré de la façon suivante. Après la définition d'un ensemble de concepts généraux relatifs au thème ainsi que du contexte du travail, nous reviendrons avec plus de précisions d'illustrations sur le contexte et la problématique de la thèse avant de conclure le chapitre.

2.2 Définitions

Dans cette section, nous donnons les définitions des concepts qui nous seront utiles pour la suite du document. Cela permet de lever toute ambiguïté pour le lecteur. Certains de ces concepts sont d'ordre général, d'autres sont relatifs à la modélisation de procédés de développement logiciels et enfin certains sont relatifs à l'exécution de procédés.

2.2.1 Concepts introductifs

2.2.1.1 Modèle

Le terme « modèle » possède plus d'une vingtaine de synonymes selon le dictionnaire

encyclopédique en ligne LINTERN@UTE [Linternaute-Website 2011]. Selon Wikipédia [Wikipedia-Website 2011], une autre encyclopédie en ligne, le même terme est utilisé dans divers domaines tels que les Sciences et Techniques, les Sciences Humaines, les Sciences de l'Information , et les Sciences de l'Art. Plusieurs définitions différentes de la notion de « modèle » sont alors disponibles; il suffit simplement de passer d'un domaine à un autre. Dans le contexte précis du génie logiciel, il n'existe pas de définition acceptée de façon standard du concept de « modèle ». Muller et al. [Muller et al. 2010] soutiennent cette idée à travers leur formule *«we are using models, as Molière's Monsieur Jourdain was using prose, without knowing it »* après avoir identifier neuf (9) définitions du même terme.
Dans cette thèse, nous retenons la définition suivante du concept de «modèle », paraphrase de [Curtis et al. 1992].

> Définition 1 : **Modèle** : une représentation abstraite d'une réalité excluant certains de ses détails jugés non pertinents dans le contexte de la modélisation.

Un *modèle* permet ainsi, comme le soutient Jeff Rotemberg dans [Rothenberg 1989], de considérer une chose du monde réel de manière simplifiée afin d'éviter la complexité, le danger et l'irréversibilité de sa réalité. Il est donc le résultat d'une abstraction et dépend ainsi du point de vue de celui qui le crée.
Dans le contexte de l'IDM, les *modèles* sont considérés comme citoyens de première classe; le développement d'une application se ramène à la construction de *modèles* successivement dépendants entre eux. Dans un tel contexte, les *modèles* auxquels on s'intéresse sont des *modèles* susceptibles d'être traités et exploités à travers des services construits pour l'effet. Ces services ont besoin des formalismes des *modèles* qu'ils traitent pour fonder leur raisonnement qui guide ces traitements et qui peut être automatisé. La nécessité d'une formalisation des *modèles* est alors de mise et est à l'origine de la notion de méta-modélisation qui correspond à l'activité de création de méta-modèles.

2.2.1.2 Méta-modèle

Pour être manipulé par une machine, un *modèle* doit être écrit dans un langage précis qui doit

être clairement défini. Afin de définir un langage de modélisation, il est nécessaire de spécifier tous ses aspects. Le langage de modélisation devient donc un sujet de modélisation. Comme tout système complexe, un langage peut être représenté par plusieurs *modèles*, descriptifs ou normatifs, de divers niveaux d'abstraction. Cette idée introduit la notion de méta-modèle [Favre 2004].

Définition 2 : **Méta-Modèle** : le *modèle* d'un langage de modélisation.

A nouveau, cette définition reste très ouverte. Mais dans le contexte de l'IDM, un *méta-modèle* est lui-même un *modèle*. Par exemple, l'OMG définit le MOF comme étant le formalisme permettant de spécifier tout *méta-modèle*. Le MOF est ainsi défini comme un méta-méta-modèle et est le seul défini par l'OMG, sa version actuelle est 2.0 [OMG-MOF2 2006]. Le MOF est ainsi essentiel pour pouvoir définir tous les langages de modélisation dans un cadre unifié. Une relation de conformité donne lieu à une architecture qui permet de faire face à la diversité des *méta-modèles*, et de les structurer dans un cadre qui permet d'envisager la définition de mécanismes de comparaison, de transformation et d'intégration de modèles, appartenant à différents langages. Le schéma de la **Figure 2.1** montre l'architecture de méta-modélisation [Favre 2004]; les *modèles* sont conformes aux *méta-modèles* qui à leur tour sont conformes aux méta-méta-modèles. Ainsi, dans ce cadre, tous les *méta-modèles* sont conformes au MOF qui est conforme à lui-même.

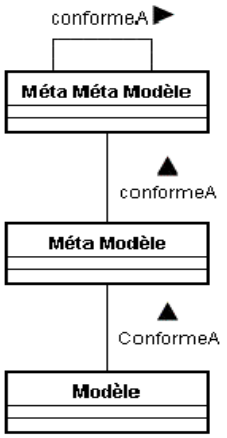


Figure 2.1: L'Architecture de Méta-Modélisation

2.2.2 Concepts relatifs à la modélisation de procédés logiciels

2.2.2.1 Procédé et modèle de procédé logiciel

Plusieurs définitions du concept de procédé sont reprises dans [Bendraou 2007] selon des communautés diverses telles que celles de la gestion de Workflow (en anglais Workflow Management (WfM) community), de Processus Métiers (Business Process Management (BPM) community), et des Systèmes d'Information (Information System (IS) community). Dans le domaine de l'Ingénierie Logicielle (Software Engineering (SE) community) qui nous intéresse directement, plusieurs définitions de la notion de procédé sont proposées par des auteurs tels que Humphrey [Humphrey & M I Kellner 1989], Lonchamp [Lonchamp 1993], Fuggetta [Fuggetta 2000], Sommerville [Sommerville 2006], etc.

Nous adoptons la définition de [Fuggetta 2000] qui met l'accent non seulement sur les spécialités en Génie Logiciel (conception, programmation, déploiement, etc.), mais également sur les éléments constitutifs d'un procédé de développement qui en l'occurrence peuvent avoir un impact sur la qualité des applications développées.

> Définition 3 : **Procédé logiciel (ou procédé)**: ensemble cohérent de politiques, de structures organisationnelles, de technologies, de procédures et d'artefacts dont on a besoin pour concevoir, développer, déployer, et maintenir un logiciel

Lorsque plusieurs agents (humains ou machines) coopèrent dans le cadre d'un projet commun, ils ont besoin d'un moyen de coordination de leur travail. Pour des tâches relativement simples ou petites, cette coordination peut se faire de façon informelle. Quand il s'agit par contre d'activités de grande envergure avec un nombre important d'agents impliqués, des arrangements plus ou moins formels sont nécessaires. De plus, au sein d'une organisation ou d'un domaine d'application, il n'est pas rare de voir les *procédés* de différents projets renfermer des éléments de similitude. Le besoin d'identifier ces points communs dans la représentation des *procédés* et d'en faire un moyen d'homogénéisation des pratiques au sein de la dite communauté s'est ainsi fait sentir.

Ainsi [Curtis et al. 1992] a défini le concept de modèle de procédé logiciel de la façon

suivante.

> Définition 4 : **Modèle de procédé (logiciel)** : description abstraite d'un procédé concret qui représente des éléments de procédé jugés importants et qui peut être exécuté par un humain ou une machine.

Un *modèle de procédé* peut donc être vu comme une description représentant un type de *procédé* donné . Un *procédé* lui-même est alors une instance d'un *modèle de procédé* qui donc peut être instancié plusieurs fois pour réaliser plusieurs métiers, applications, etc. Le *modèle de procédé* est alors une prescription de comment les choses doivent, devraient, ou pourraient se faire, là où le *procédé* lui-même correspond à ce qui se passe effectivement*.
Un *modèle de procédé* logiciel est exprimé par un langage de modélisation de procédés (LMP). Nous empruntons à [Fuggetta 2000] la définition suivante.

> Définition 5 : **Langage de modélisation de procédés (LMP)** : formalisme qui permet de représenter, d'une façon précise et compréhensible, les éléments d'un *procédé* comme les activités, les rôles, la structure et la nature des artefacts à créer et/ou modifier, et les outils à utiliser (exemple: outils CASE, compilateurs, etc.).

2.2.2.2 Produit de procédé et modèle de produit de procédé

Plusieurs éléments composent un *procédé*, et un *modèle de procédé* doit faire ressortir chacun d'eux. Ces éléments sont des activités, des humains, des rôles, des outils, des équipes, des produits, etc.

> Définition 6 : **Produit (de procédé logiciel)** : une donnée créée, consommée ou modifiée durant l'exécution d'un *procédé*.

Les *produits* sont les entrées et sorties des activités. Un *produit* correspond à un résultat attendu ou à utiliser pour faciliter le déroulement du *procédé*. Plusieurs termes sont employés

* A noter qu'un *modèle de procédé* ne doit pas être confondu avec un *modèle de cycle de vie.* Ce dernier est plus abstrait et se limite à indiquer la philosophie qui sous-tend le cycle de développement [6].

dans la littérature pour les désigner: *Produit* [Conradi R., Fernström C., Fuggeta A. 1992], *WorkProduct* [OMG-SPEM2 2008] et [ISO-SEMDM 2007], *Ressource* [Cass et al. 2000], *Artefact* [Fuggetta 2000]. Plusieurs définitions sont également proposées dans la littérature pour cette notion. Dans SPEM2 [OMG-SPEM2 2008] par exemple, un WorkProduct est défini comme *un produit consommé, produit ou modifié par une tâche*. Il peut servir de base pour la définition d'une brique réutilisable. Les rôles utilisent des WorkProducts pour réaliser leurs tâches et produisent d'autres WorkProducts durant cette réalisation. Ils sont sous la responsabilité d'agents qui incarnent ces rôles.

Ces concepts rencontrés dans la littérature sont utilisés par leurs auteurs respectifs pour désigner un *produit*. Dans nos travaux, nous avions besoin, en plus, de désigner la spécification (ou description) d'un *produit*. Nous emploierons le concept de WorkProduct à cet effet vu l'importance que nous avons donnée à la différence entre un *produit* et sa spécification. En effet, la spécification d'un *produit* est abstraite et est fournie à la modélisation du *procédé*, alors que le *produit* lui-même représente la donnée concrète créée et/ou modifiée au cours de l'exécution. La définition que nous donnons à ce concept de WorkProduct diffère donc de celle de l'OMG, comme le précise la définition 7 ci-après.

Définition 7 : **WorkProduct**: Partie d'un *modèle de procédé* restreinte à la spécification d'un *produit* du *procédé*. Un *WorkProduct* est donc le *modèle* d'un *produit de procédé*.

L'avènement de la vision IDM fait que les *produits* d'un *procédé* sont des *modèles* pour l'essentiel. Ce sont de tels *produits* auxquels nous nous intéresserons dans cette thèse comme il sera montré plus loin à la section 2.3 de ce chapitre qui traite de notre problématique

Définition 8 : **Produit-modèle** : Tout *produit* consommé ou créé par une activité du *procédé* qui vient sous la forme d'un *modèle*.

2.2.3 Concepts relatifs à l'exécution de procédés logiciels

2.2.3.1 Moteur d'exécution de procédé

Dans sa classification des *modèles de procédés* [Scacchi 2001], Scacchi distingue deux types principaux: les *modèles de procédés* dits opérationnels et ceux dits non-opérationnels. Les *modèles* non opérationnels doivent leur qualification au fait qu'ils correspondent à des approches conceptuelles qui ne produisent pas une sémantique d'exécution suffisante. Le modèle WinWin défini dans [Boehm et al. 1998] est cité en exemple par Scacchi.

C'est alors que l'auteur définit un *modèle de procédés* opérationnel comme une définition de *procédé* exprimée sous forme de programmes ou de *modèles* exécutables.

Un *modèle de procédé* exécutable est par ailleurs défini par Zamly et al. comme étant interprétable par un agent qui alors devient capable de guider de façon active ou même de contrôler le déroulement du *procédé* représenté [Zamli & Lee 2001].

> Définition 9 : **Moteur d'exécution (de modèles de procédés)**: interpréteur d'un *LMP* permettant l'exécution de *modèles de procédé* écrits dans ce langage.

Les *moteurs d'exécution* permettent ainsi de guider les activités du développement, de produire des éléments (guides, templates, etc.) capables d'augmenter la performance des humains qui travaillent sur un projet. Ils permettent aussi d'offrir un support collaboratif entre individus et/ou équipes intervenant dans le même projet, et d'aider à assurer l'intégrité du *procédé*. Par ailleurs, le *moteur d'exécution* doit aussi s'occuper de la gestion des *produits* d'un *procédé* en exécution. Nous reviendrons plus amplement sur cet aspect en définissant la problématique de la thèse dans la section 2.3 de ce chapitre, notamment pour préciser les faiblesses significatives des moteurs d'exécution actuellement disponibles du point de vue de cette gestion.

2.2.3.2 Process-centered Software Engineering Environment

Dans les années 90, la tendance dominante associée à l'automatisation de *procédés logiciels* favorise le développement d'environnements d'exécution et de coordination appelés PSEEs

(*Process-centered Software Engineering Environment*).

Avec un *modèle de procédé* en entrée, un PSEE peut produire un nombre varié de services [Ambriola et al. 1997] tels que:

1. l'automatisation des tâches routinières et une assistance aux développeurs dont la productivité est alors accentuée,
2. l'invocation automatique et le contrôle des outils de développement,
3. l'application de règles définies au préalable,
4. etc.

Un PSEE est construit sur la base d'une philosophie dépendante des *modèles de procédés* qu'il supporte. Certains opèrent sur un *modèle de procédé* en réponse à des ordres de l'utilisateur (un développeur) qui alors guide l'exécution du *procédé*. D'autres fournissent un guidage actif de l'exécution du modèle en interagissant avec l'utilisateur pour par exemple lui rappeler certaines de ses tâches, les deadlines, etc. Un PSEE peut donner la possibilité à un utilisateur de choisir d'exécuter ou non une tâche qui lui est soumise, il peut également exiger l'action de celui-ci. Enfin un PSEE peut exécuter un *modèle de procédé* sans l'intervention de l'utilisateur. Enfin, un seul PSEE peut adopter plus d'une forme de support parmi les précédentes [Cugola & Ghezzi 1998].

Dans [Gruhn 2002], Gruhn définit un PSEE comme un environnement offrant le support de l'exécution de *procédés*. Ce support est assuré à travers une coordination des outils et agents impliqués dans cette exécution. Ces environnements peuvent supporter une variété de *procédés* moyennant un paramétrage, et l'auteur préconise l'utilisation d'un *méta-modèle* pour structurer les *procédés* supportés par un PSEE.

Dans les chapitres à venir, nous utilisons le terme PSEE avec une définition légèrement différente de celles qui précédent.

Définition 10 : **PSEEs (*Process-centered Software Engineering Environment*)** : environnement intégrant un espace de modélisation de *procédés* et un *moteur d'exécution*. Il offre un support pour tout ou partie des activités de modélisation et d'exécution de *procédé*. En particulier, il permet la coordination des outils et agents impliqués dans l'exécution d'un *procédé*.

C'est donc, pour simplifier notre vocabulaire dans la suite, que nous avons choisi d'étendre la définition du terme *PSEE* à l'environnement de modélisation de *procédé*. En effet, nous considérons que la procédure de création et de chargement d'un *modèle de procédé* définit, si on se réfère à l'idée de Gruhn [Gruhn 2002] évoquée ci-avant, tout ou partie du paramétrage d'un *PSEE* afin de lui permettre l'exécution du *procédé* en question. En effet, par définition, un *modèle de procédé* contient les réponses à la question du comment l'exécution du *procédé* en question va se faire. En particulier, un PSEE intègre un *moteur d'exécution*.

2.2.3.3 La gestion des *produits* d'un *procédé*

Lors de l'exécution d'un *procédé*, les différents *produits* qui sont les entrées et sorties des activités des développeurs subissent des modifications successives. L'origine et la nature de tels changements des *produits* varient suivant le contexte d'exécution et la nature du *procédé*. En effet un changement peut provenir d'un ou de plusieurs développeurs répartis sur des sites différents. Il peut également s'agir d'un changement sur un *produit* qui implique ou non d'autres changements sur d'autres *produits* du même *procédé*. Il est ainsi nécessaire de disposer d'un système de gestion de l'évolution des *produits*. Par ailleurs, cette gestion de l'évolution des *produits* d'un *procédé* en exécution inclut une gestion de la cohérence de tels *produits*. En effet, des modifications simultanées de plusieurs développeurs sur les *produits* peuvent faire naître des conflits. Dans cette section nous définissons des concepts relatifs à la gestion des *produits* et de leur évolution.

Dans le cadre de la construction de larges systèmes, l'utilisation de techniques propres aux systèmes de gestion de configuration logicielle (CMS, *(Software)* Configuration Management System en anglais) est devenue une nécessité [Estublier et al. 2005]. La gestion de configuration logicielle peut être définie comme une gestion, un contrôle des changements d'un logiciel. Plus d'information sur les CMS peut être trouvée dans [Dart 1991] et [Reidar Conradi & Westfechtel 1998]. Par ailleurs, en offrant la possibilité de gérer les versions successives des *produits* impliqués dans l'exécution d'un *procédé*, un système de gestion de version, encore appelé système de contrôle de version (VCS, Version Control System) réalise un aspect de la gestion de configuration logicielle. En effet, les changements qui interviennent sur un système logiciel sont les conséquences de changements appliqués aux *produits* impliqués dans son *procédé* de développement.

Nous emploierons, dans la suite de ce document, indifféremment les termes CMS et VCS auxquels nous donnons la définition suivante.

> Définition 11 : **Système de gestion de configuration (CMS)** ou **Système de Contrôle de version (VCS)**: un système qui permet de gérer l'évolution (les versions successives) des *produits* impliqués dans l'exécution d'un *procédé*. La structure des *produits* dont l'évolution est gérée par un CMS/VCS est guidée par un **modèle de donnée**. C'est celui-ci qui détermine la façon dont les *produits* sont stockés ainsi que les relations entre eux.

Nous donnons ci-après les définitions d'autres concepts souvent rencontrés dans ce domaine [Dart 1991; Reidar Conradi & Westfechtel 1998] et que nous utiliserons dans la suite de ce document .

> Définition 12 : **Unité de configuration** (CI, Configuration Item) ou **unité de version** (UV) : élément défini au sein d'un modèle de donnée d'un CMS/VCS et pour lequel le système de gestion peut stocker des informations de version.

Il convient à ce niveau de noter que dans la plupart des *VCS* existants, l'*UV* est le fichier [H. Oliveira et al. 2005].

> Définition 13 : **Unité de comparaison** (UC, Unit of Comparison) : élément sur lequel le système interdit un accès simultané.

L'*UC* est donc l'unité de verrouillage. Dans le cas d'un fichier texte, elle peut correspondre au paragraphe, à la ligne, etc.
Les définitions précédentes sont relatives à la gestion de l'évolution des *produits de procédés*. La cohérence de ces *produits* peut quant à elle se définir à deux niveaux différents, que nous nommons respectivement cohérence structurelle et cohérence relationnelle.

> Définition 14 : **Cohérence structurelle** : préservation de contraintes d'intégrité structurelle par un *produit* au cours de son évolution.

Un exemple de cette cohérence, dans le cas de modèles UML, peut s'énoncer comme suit: *une classe ne peut pas posséder deux attributs de même nom*. Cette définition de la cohérence considère les *produits* pris les uns indépendamment des autres. Plusieurs travaux sont réalisés dans la littérature pour la gestion d'une telle cohérence de *produits* [Alexander Egyed 2007; Mens et al. 2006; Blanc et al. 2008].

> Définition 15 : **Cohérence relationnelle** : préservation de relations entre *produits* au cours de leurs évolutions respectives.

Dans cette deuxième définition de la cohérence qui nous intéresse le plus, les *produits* sont considérés les uns par rapport aux autres. La cohérence est alors déterminée sur la base de la nature des relations qui existent entre les *produits* ainsi que les actions des développeurs sur ces *produits*. En effet, si deux *produits* sont liés, leurs évolutions respectives doivent être maintenues « synchronisées ».

La littérature comporte également d'importants travaux sur la synchronisation de *produits-modèles*. Ces travaux visent tous le même objectif qui est de s'assurer que deux *produits-modèles* (respectivement source et cible d'une opération de transformation, par exemple) restent toujours synchronisés (cohérents entre eux) même après un changement de l'un d'entre eux. Certaines approches de synchronisation que nous avons pu trouver comme [Bottoni et al. 2008; Johann & A Egyed 2004] obligent, à chaque transformation, l'écriture explicite d'un code spécifique aux éléments de modèles changés et à la transformation elle-même pour atteindre l'objectif. D'autres comme [Xiong et al. 2007] proposent une génération du code de la synchronisation à partir de celui de la transformation, mais obligent à ré-exécuter la transformation à chaque modification de la source. Dans [Ráth et al. 2009] est également présentée une approche « orientée changement » qui se base sur une capture des changements effectués sur un des deux *produit-modèles* suivie de leur application à l'autre en s'appuyant sur un modèle de changement généré à l'effet.

Nous remarquons ainsi que ces travaux sous-considérent les relations qui existent entre les *produits-modèles* et donc ne mettent pas en valeur la sémantique de ces relations. Nous reviendrons plus en détail sur cet aspect dans le chapitre 3 qui présente l'état de l'art sur le sujet.

2.2.4 Granularité de produits

Un *modèle de procédé* est une représentation abstraite, construite au moyen d'un *LMP*, des différents éléments considérés importants dans le développement d'un système logiciel. Les niveaux d'abstraction peuvent aller d'une description avec un haut niveau de granularité des phases du cycle de développement empêchant toute automatisation, à une fine description des étapes de développement permettant alors une entière exploitation et exécution du *modèle* par une machine. Notre définition de la granularité est inspirée de celle proposée dans [Feiler & G. E. Kaiser 1987].

> Définition 16 : **Granularité d'un produit** : niveau selon lequel un *produit* est décomposé en entités pouvant être elles aussi considérées comme des *produits* en relation ou non entre eux et/ou avec d'autres produits, mais pouvant être manipulées et/ou stockées les unes indépendamment des autres.

La majeure partie des *LMPs* fournit une relation de composition entre les *produits*. Une telle relation permet de définir le niveau de granularité voulu pour les *produits* selon leur nature. On peut par exemple exprimer avec une telle relation qu'un programme se compose de modules, les différents modules de procédures, etc., ou encore qu'un répertoire se compose de fichiers, et ainsi de suite. Il faut noter que cette relation de composition entre *produits* d'un *procédé* n'est pas adaptée pour des *produits-modèles*. En effet, aucune sémantique n'est aujourd'hui définie pour la relation de composition dans le cas des *modèles*. Nous reviendrons pour préciser cela dans le chapitre 3 qui traite de l'état de l'art sur la question.

2.2.5 Typage de produits-modèles

Les *modèles* constituent les entités de première classe en IDM. Dans ce contexte actuel de l'ingénierie logicielle, des services de plus en plus nombreux et orientés *modèles* se développent dans le monde de la recherche et de plus en plus dans l'industrie du logiciel. La particularité de tels services est que pour l'essentiel ils consomment et produisent des *modèles*. Prenant des *modèles* en entrées et en fournissant d'autres en sortie, un service a besoin de raisonner sur la nature de ces *modèles*. C'est cet état de fait qui correspond au fondement de l'émergence de l'idée de la notion de typage de *modèles*. De façon générale, un type peut être

considéré comme un ensemble de valeurs et des opérations associées et applicables à ces valeurs [Steel & Jézéquel 2005]. Une fois un ou plusieurs types définis, il devient possible de les utiliser pour spécifier des opérations en terme de la nature (donc du type) de leurs entrées et sorties.

Le typage des *modèles* permet donc, dans le contexte qui nous intéresse, de raisonner sur la nature ainsi que la structure interne de ces *modèles*. Par exemple, il permet de raisonner sur les conditions selon lesquelles un service peut être appliqué à un *modèle*. Il faut noter qu'en plus et surtout, le typage des *modèles* peut servir de base de raisonnement sur les relations possibles entre eux ainsi que sur la nature de celles-ci.

Les éléments précédents constituent la motivation principale de notre proposition qui est de définir le type des *produits-modèles* au moment de la conception des *modèles de procédé*. Il nous a dès lors fallu adopter l'approche de typage la plus adéquate. En effet, un parcours de la littérature nous a permis de relever l'existence de deux principales approches de typage de *modèles* que nous présenterons dans la suite. Ces approches ont en commun de s'appuyer sur les mécanismes de typage utilisés dans les systèmes orientés objet (OO) dont nous donnons un bref aperçu dans un premier temps.

2.2.5.1 Le typage orienté objet

Dans un système OO, un type définit des propriétés communes d'un ensemble d'objets de mêmes caractéristiques. Il correspond à la notion de *type abstrait de données* et comprend deux parties décrites ci-après [Atkinson et al. 1992].

- L'interface, c'est un ensemble d'opérations avec leurs signatures. Elle est visible pour les utilisateurs du type.
- L'implémentation, elle contient, à son tour, deux parties. La première correspond aux données décrivant la structure interne des objets du type. La seconde partie de l'implémentation d'un type consiste en un ensemble d'opérations/procédures qui permettent de réaliser l'interface du type. L'implémentation n'est visible que pour le seul concepteur du type.

En programmation OO [Anderson 1988], les types sont utilisés pour accroître la productivité des programmeurs. En effet, le système trouve une possibilité de raisonnement sur la correction ou non de tel ou tel programme, et ce à travers les systèmes de typage qu'il utilise

et les informations de typage des données des programmes. Dans ce paradigme (OO) par ailleurs, la notion de type n'est pas à confondre à celle de classe. La spécification d'une classe est en effet la même que celle d'un type. Mais en plus de pouvoir être utilisées par le système pour raisonner sur la nature des objets, une classe contient une *fabrique* et un *entrepôt* d'objets (respectivement *object factory* et *object warehouse*). La fabrique est utilisée pour construire de nouveaux objets (à travers une toute nouvelle création, avec un opérateur **new** par exemple, ou un clônage d'objets spécifiques de la classe). L'entrepôt quand à lui correspond à une définition en extension de la classe; il consiste en effet à l'ensemble des objets instances de la classe. De façon générale, sans être le citoyen de première classe, le type a un statut spécial dans certains systèmes [Atkinson et al. 1992], et n'est pas modifiable à l'exécution. Par ailleurs, dans [Atkinson et al. 1992], les auteurs soulignent qu'à la base les classes ne sont pas utilisées pour raisonner sur la nature (la correction ou la non correction) des programmes, mais plutôt pour créer et manipuler des objets. Néanmoins, dans les systèmes OO actuels les classes sont les citoyens de première classe. Et pour plus de flexibilité et d'uniformité, ces classes peuvent être manipulées en runtime, c'est à dire mises à jour, passées en paramètre, etc. Il y a donc d'énormes similarités entre les notions de classes et types. La différence entre les deux notions est d'ailleurs très subtile dans certains systèmes de typage.

L'intérêt que nous avons porté au système de typage OO se justifie par le fait qu'un *modèle* soit essentiellement vu dans la communauté de l'IDM comme un ensemble d'objets dont chacun correspond à un élément du *modèle*. Ces éléments de modèles sont liés entre eux pour former un graphe arbitrairement complexe décrit par un *méta-modèle*. Une telle situation a conduit à une forte influence du typage des *modèles* par celui OO. Nous présentons, dans la sous-section suivante, deux approches de typage de *modèles* dans ce contexte.

2.2.5.2 Le typage de modèles

La première approche est l'approche traditionnelle, elle n'introduit aucune nouveauté relative aux *modèles*. Dans cette vision, un *modèle* n'est pas vu comme une entité globale, mais plutôt comme un ensemble d'objets. Les applications qui manipulent les *modèles* ne s'intéressent pas à eux en tant qu'entités, mais s'intéressent plutôt aux différents objets qu'ils contiennent. Aucun typage spécifique aux *modèles* n'est donc défini dans cette approche; les applications y orientent leur raisonnement vers les types des objets qui alors sont considérés comme des

objets habituels et sont manipulés comme tel. Cette approche traditionnelle est utilisée dans les premières applications de transformations de *modèles* définies dans le cadre IDM comme [Gerber et al. 2002; Qvt-Merge-Group 2005; Sendall 2003]. En guise d'illustration, considérons un *modèle* M qui contient la classe UML nommée *Display* avec son attribut *size* et son opération *draw()*. Pour un programme, dans cette approche, les types utilisés sont *TypeC, TypeA,* et *TypeO* considérés respectivement comme ceux associés aux méta-classes UML *Class*, *Attribute* et *Opération.* Dans [Edwards et al. 2004; J Steel & M Lawley 2004], les auteurs indiquent que l'utilisation d'une telle approche de typage rend les programmes fragiles face aux changements sur les *méta-modèles* qui structurent les *modèles* traités. En l'occurrence, ils changent de comportement suite à des modifications (de ces *méta-modèles*) qui ne devraient, en principe, pas affecter leur résultats.

La deuxième approche marque une rupture. Elle pose les bases d'un système de typage spécifique aux *modèles*. Elle est essentiellement sous-tendue par les travaux de Steel et al. [Jim Steel & Jézéquel 2007; Jim Steel & Jézéquel 2005]. C'est une approche qui, plutôt que de se rabattre exclusivement sur le typage OO, l'utilise certes mais pour définir un vrai typage orienté *modèles* (OM). En effet, Steel et al. soulignent la nécessité parfois accrue pour les applications de raisonner suivant le type des *modèles* manipulés plutôt que sur les objets qu'ils contiennent. Les auteurs remarquent donc l'inadéquation de l'approche précédemment présentée pour permettre à une application de raisonner sur le type des *produits* et proposent un système de typage considéré plus adéquat, c'est dire qui permette à un utilisateur de spécifier ses programmes en terme de *modèles* et de types de *modèles*.

Par ailleurs, aucune spécification du MOF ne contient une définition formelle de la notion de *modèle* ou de *méta-modèle*. Dans MOF 1.x en effet, un *modèle* est considéré comme une « instance d'un paquetage » pour intuitivement désigner un ensemble d'objets instances de classes d'un paquetage MOF. Bien que le terme « instance de paquetage » ne soit pas défini par MOF 1.x, il est très rapidement devenu inadapté à l'arrivée de MOF 2.0 qui introduit des mécanismes permettant de construire des *modèles* dont les objets sont instances de classes de paquetages différents et conférant une structure plus complexe aux *modèles* qui alors lient des ensembles d'objets de paquetages différents. Dans UML 2, un *modèle* est considéré comme l'instance d'une classe à laquelle s'ajoute l'ensemble de tous les objets contenus ou

pouvant être atteints à partir de cette instance racine. Une telle définition trouve sa faiblesse dans le fait qu'elle ne prend pas en compte certaines catégories de *modèles* ne disposant pas d'un unique élément racine. Ainsi la définition qui semble la plus générale est, comme souligné dans [Jim Steel & Jézéquel 2005] celle qui définit un *modèle* comme un ensemble d'objets correspondant à ses éléments de *modèles*. C'est cette définition est à la base du système de typage de Steel et al.. Si nous reconsidérons l'exemple de la première approche, un programme traitant le modèle *M* dans cette nouvelle approche va considérer le type comme étant *TypeM={TypeC,TypeA,TypeO}*.

Cette deuxième approche offre une possibilité de raisonnement sur un *modèle*. On peut, par exemple, avec l'aide de *TypeM*, faire les déductions suivantes:

- *M* contient une ou des classe(s), attribut (s), etc.,
- *M* ne peut contenir une même classe qu'un autre *modèle M1* tel que *TypeM1* ne contient pas *TypeC*,
- etc.

En revanche aucune possibilité de raisonnement sur un modèle sur la base du typage n'est possible dans la première approche.

Nous retenons, ainsi, la définition suivante du type d'un *modèle*.

> Définition 17 : **Type de modèle**: ensemble composé des méta-classes qui composent le *méta-modèle* dont le *modèle* est une instance.

Cette définition ne prend pas en compte les associations et multiplicités et les autres éléments possibles d'un *méta-modèle*. En effet, l'adjonction de ces éléments dans un type ne contribuent pas à accroître la possibilité d'une *conformité* entre les types. Celle-ci permet qu'un même type puisse être (ré)utilisé pour typer le maximum possible de *modèles*. Signalons, enfin, que de ce point de vue, il est montré dans [Jim Steel & Jézéquel 2005] la différence entre un type de *modèle* et un *méta-modèle*.

2.3 Contexte, problématique et illustration

L'avènement de la vision IDM fait que les *produits* d'un *procédé* sont des *produits-modèles,* pour la plupart. De plus, ces *produits-modèles* sont reliés entre eux par des relations qui

peuvent être de nature diverse. Dans ce même contexte, les *produits-modèles* d'un *procédé* évoluent du fait de l'exécution d'opérations dont ils sont des entrées et sorties au cours des activités du *procédé*. Ces opérations sont implémentées par les outils du *procédé* et peuvent elles aussi être de diverse nature. Devant une telle situation, se pose le problème de la gestion des *produits-modèles* au cours de l'exécution d'un *procédé*. En effet, devant la situation précédemment décrite, les *moteurs d'exécution* se heurtent à des limites dans la gestion de l'évolution de ces *produits-modèles* et, en l'occurrence, de leur *cohérence relationnelle* au cours de cette évolution. Cette faiblesse des *moteurs d'exécution* est strictement liée à un non support par les *LMPs* de la spécification des relations entre les *produits-modèles*.

Dans cette section, nous allons d'abord présenter un certains nombre de relations qui souvent existent entre les *produits-modèles* d'un *procédé* en exécution. Nous présenterons, ensuite, des opérations définies dans la littérature pour être appliquées à ces *produits-modèles* au cours du déroulement des activités d'un *procédé*. C'est ainsi que nous allons expliciter la problématique précédemment énoncée avant de l'illustrer à travers des exemples. Nous finirons la section par une synthèse.

2.3.1 Relations entre produits-modèles

Plusieurs relations peuvent exister entre les *produits-modèles* d'un *procédé* en cours d'exécution. Cette situation est accentuée dans notre contexte d'étude par la logique IDM selon laquelle le cycle de développement est vu comme une suite de transformations successives des *produits-modèles*. Les *produits-modèles* sont alors souvent fortement dépendants les uns des autres dans un tel contexte.

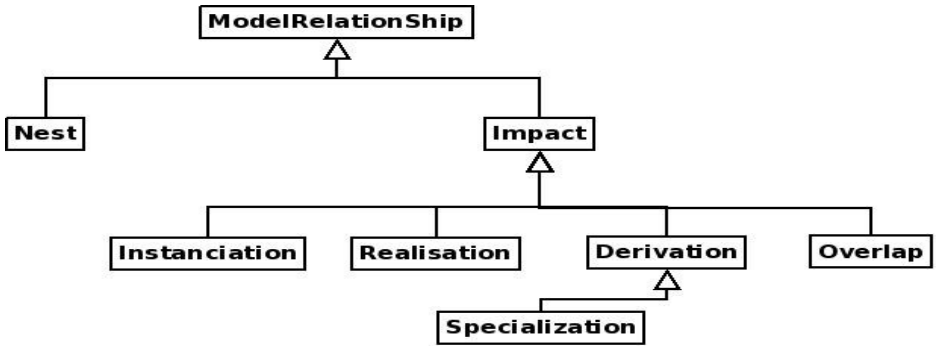


Figure 2.2: Exemples de Relations entre *Produits-Modèles*

La **Figure 2.2** représente des relations possibles entre *produits-modèles* d'un *procédés* en exécution. Nous donnons ci-après leurs sémantiques respectives en nous appuyant sur des travaux de la littérature présentés dans [Hailpern & Tarr 2006; Pons et al. 2000; Spanoudakis & Finkelstein 1998; Spanoudakis et al. 1999].

- Le *nest*: est une relation définie d'un *produit-modèle* source vers une ou plusieurs autres cibles. Un lien *nest* permet de spécifier que même si chacune des cibles est physiquement séparée des autres, elle reste logiquement une partie intégrante de la source. C'est l'exemple d'un modèle UML composé de classes et de cas d'utilisation d'un système tel que les classes ainsi que les éléments de modèles liés soient définis dans un fichier séparé de celui dans lequel les autres éléments de modèles (les use case, etc.) sont définis. Nous reviendrons plus amplement sur la sémantique de ce lien au chapitre 4.
- L*'instanciation*: elle se définit d'une source vers une cible pour indiquer que les éléments de modèles de la cible sont des instances d'éléments de la source. Une classe UML est, par exemple, une instance de la méta-classe **Class** du *méta-modèle* UML 2 [OMG-UML2 2007].
- La *dérivation*: elle se définit d'une source vers une cible pour spécifier que les éléments de modèle de la cible sont la conséquence (ou sont générés à partir) d'éléments de la source. La relation de dérivation permet d'exprimer une transformation quelconque d'éléments de la source en un ou plusieurs éléments de la cible. Il faut ainsi noter une particularité significative de cette relation par rapport aux autres. En effet, dans cette relation, un élément de la cible peut être associé à plusieurs éléments de la source (et non forcément à un seul) et inversement. Exemple: dans le cadre d'un *procédé* s'exécutant dans un contexte IDM, soit la classe *C2* qui porte une opération *O2* et un attribut *A2* d'un *modèle* UML *M2*. Considérons ensuite que les éléments du modèle *M2* sont générés à partir d'une classe *C1* d'un autre modèle UML *M1* via une transformation de *modèle* [Bézivin et al. 2006; T Mens & Gorp 2006]. Une relation de *dérivation* peut alors être définie entre M1 (la source) et M2 (la cible).
- La *spécialisation* ou l'*extension* ou encore le *raffinement*: elle se définit d'une source vers une cible et est un cas particulier de *dérivation.* Elle représente un enrichissement

de la source à travers l'adjonction d'éléments de modèle additionnels à la cible. Les éléments de la cible sont donc des instances spécifiques d'éléments plus génériques de la source. Dans [Pons et al. 2000], les auteurs proposent, par exemple une définition formelle de l'*extension* qui prend en compte la condition de l'extension et une référence à un point d'extension de la cible (point par lequel l'extension est possible).

- La *réalisation:* les éléments de la cible représentent une implémentation des éléments de la source. Exemple, dans le cas de modèles UML, d'une classe *C* (portant l'attribut *a1*) qui implémente une interface *I* (qui à son tour déclare les opérations *o1* et *o2)*. Tel que nous venons de la définir, la *réalisation* ressemble fortement à un cas particulier de la relation de spécialisation. Ce n'est tout de même pas le cas; dans l'exemple précédent en effet, l'interface I, qui est un élément de la source et qui est impliquée dans la relation, ne se retrouve pas dans la cible où elle est « remplacée » par un autre élément qui est la classe *C*.
- L'*overlap*: c'est une relation non orientée, contrairement aux relations précédentes. Elle se définit entre deux ou plusieurs *produits-modèles* pour spécifier le fait qu'ils contiennent des éléments de modèles communs. Exemple: soit deux *modèles* UML *M1* et *M2* tels que certaines (ou toutes les) classes, attributs et opérations de *M1* se retrouvent aussi dans *M2*. Une telle situation se produit souvent dans le contexte de l'exécution de *procédés*, notamment entre les *produits-modèles* de l'activité d'analyse et ceux de conception. *M1* et *M2* sont alors liés par une relation d'*overlap*. C'est une relation qui est non orientée. Comme pour le *nest*, l'étude que nous avons faite de cette relation va être présentée au chapitre 4. Elle constitue, en effet, le coeur de la contribution de cette thèse.

2.3.2 Opérations sur les produits-modèles

Dans la sous-section précédente, nous avons montré l'importance des relations pouvant exister entre les *produits-modèles* d'un *procédé* en exécution. Nous avons également remarqué leur non support par les *LMPs* existants. Nous soulignions également, en introduisant cette section, que dans le contexte IDM, ces *produits-modèles* évoluent suite à l'exécution d'opérations dont ils sont des entrées et sorties au cours des activités du *procédé*. Les opérations en question sont celles implémentées au sein des outils du *procédé* et peuvent être

de diverse nature. Par ailleurs, contrairement aux relations, des travaux significatifs trouvés dans la littérature ont permis de les formaliser [H. Oliveira et al. 2005; Bézivin et al. 2006; Benoit Baudry et al. 2005; Reddy et al. 2005; Farah et al. 2009; Murta et al. 2007]. Nous avons classé certaines de ces opérations à l'aide de la **Figure 2.3**. Ci-après nous décrivons brièvement trois d'entres elles qui nous paraissent principales.

- La *composition de produits-modèles.*

  Désigne l'opération de fusion de deux ou plusieurs *modèles* pour obtenir un seul *modèle* [Benoit Baudry et al. 2005]. Cette opération peut être utilisée pour avoir une vue synthétique d'un système représenté à travers plusieurs vues partielles correspondant chacune à un *modèle*. Plusieurs variantes de cette opération sont développées dans la littérature [Reddy et al. 2005; Bézivin et al. 2006; Benoit Baudry et al. 2005]; on distingue, par exemple le *merge* et l'*override* [Benoit Baudry et al. 2005], et bien d'autres approches présentées dans [Reddy et al. 2005], ou référencées dans [Bézivin et al. 2006]. Ces variantes diffèrent essentiellement par les conditions d'intégration des *modèles* à composer. Ces conditions d'intégration sont à leur tour dépendantes de conditions d'équivalence ou de correspondance définies entre éléments des différents *modèles*, mais aussi de la façon suivant laquelle ces éléments de modèle sont fusionnés.

- La *transformation de produits-modèles.*

  Elle peut être définie de la façon générique suivante, synthèse de deux définitions trouvées dans [Bézivin et al. 2006] et [T Mens & Gorp 2006]. C'est une opération qui prend un ou plusieurs *produit(s)-modèle(s)* en entrée, exécute de façon automatique un ensemble de règles sur les éléments de *modèles* contenus avant de produire un ou plusieurs autre(s) *produits-modèle*(s) en sortie. En ce sens, en reprenant Mens et al. dans [T Mens & Gorp 2006], une composition de deux *produits-modèles* peut être vue comme une transformation les prenant en entrée et fournissant le *produit-modèle* résultat de la composition comme sortie.

- Le *versionning* de *produits-modèles.*

  Il peut être défini comme le fait de déterminer et de stocker des informations de version (numéro, date, etc.) pour un *produit-modèle*. Comme souligné dans

[H.Oliveira et al. 2005], de nombreux travaux existent aujourd'hui pour réaliser cette opération de versionning pour du code source ou des données textuelles de façon générale, mais beaucoup reste à faire pour une adaptation des systèmes résultants au contexte des *produits-modèles* de *procédés* dans le contexte IDM. Des travaux non négligeables sur le versionning de *modèles* sont tout de même disponibles dans la littérature [Farah et al. 2009; Murta et al. 2007; Sriplakich et al. 2008; Sriplakich et al. 2006].

Après avoir montré, en section 2.3.1, l'existence de relations variées entre *produits-modèles*, Nous venons dans cette section de préciser que dans le contexte IDM, les *produits-modèles* sont fortement reliés entre eux et que leur évolution est le résultat de l'exécution d'opérations de mieux en mieux définies dans la littérature. Cette situation, couplée à l'existence décrite en section 2.3.1 de relations variées entre *produits-modèles*, est à l'origine de la problématique que nous décrivons et illustrons dans la section suivante.

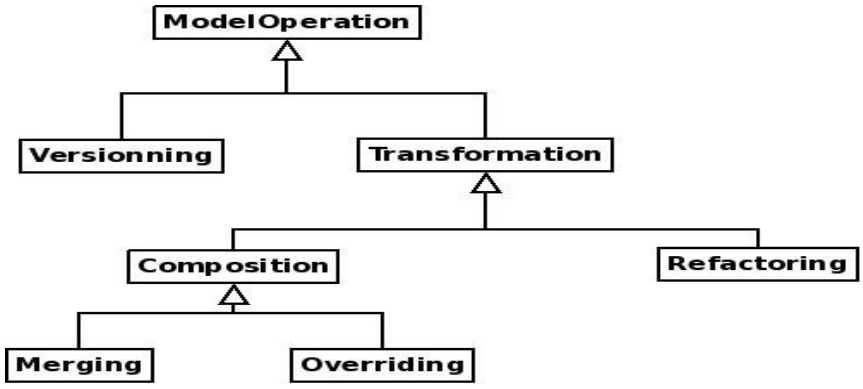


Figure 2.3: Quelques Opérations sur les *Modèles*

2.3.3 La problématique: description et illustration

Les deux sections précédentes ont décrit le contexte dans lequel se place notre travail. Il s'agit, dans la vision IDM, de l'existence de relations variées entre les *produits-modèles* d'un *procédé* en cours d'exécution. Cette situation s'associe au fait que des opérations bien connues, et également de diverses sortes, sont appliquées à ces *produits-modèles* au cours du déroulement des activités de *procédés*. C'est ainsi que se pose la problématique que nous explicitons à travers les deux facettes suivantes: l'expressivité des relations entre *produits-modèles*, et

l'exploitation de ces relations au cours de la gestion des *produits-modèles*. Les deux sous-sections suivantes fournissent les développements respectifs.

2.3.3.1 Expressivité des relations entre *produits-modèles*

Les relations décrites en section 2.3.1 sont existantes entre *produits-modèles* de *procédés* s'exécutant dans le contexte IDM. Il faut noter par ailleurs qu'à travers une étude présentée dans [Pons et al. 2000] sur des relations de dépendance entre *produits-modèles* d'un *procédé* de type processus unifié, Pons et al. soutiennent la nécessité pour un *modèle de procédé* de fournir une définition précise de la syntaxe et de la sémantique de ses différents *produits-modèles* ainsi que de leurs relations. Au vu des résultats présentés par les auteurs dans cet article, en l'occurrence ceux relatifs à l'importance du nombre et de la nature des relations entre les *produits-modèles*, nous considérons que cette idée est valable pour un *procédé* de type quelconque et n'est donc pas seulement spécifique au processus unifié considéré par les auteurs dans leur cas d'étude.

Il faut noter que notre parcours de l'état de l'art que nous présenterons au chapitre 3 ne nous a pas permis de trouver un *LMP* offrant un support pour la modélisation des relations. On constate néanmoins que presque tous les *LMPs* rencontrés supportent les relations de composition et de dépendance (souvent avec des appellations différentes). Toutefois, ces *LMPs* considèrent ces relations entre des *produits* de façon générale, c'est à dire sans prendre en compte la nature de ceux-ci.

Il faut également noter que la relation d'*overlap* a été largement adressée par Spanoudakis et al. [Spanoudakis & Finkelstein 1998; Spanoudakis et al. 1999] qui en donnent une définition assez générale pouvant se résumer en *« une relation entre les interprétations des composants de deux spécifications »*. Avec une telle définition, ces travaux appliqués à des *produits-modèles* fournissent des résultats similaires à ceux sur la synchronisation de *modèles* que nous avons présentés précédemment. De plus, ils ne définissent aucun cadre d'expression pour la relation de partage d'éléments entre *produits-modèles*.

2.3.3.2 Exploitation des relations dans la gestion de l'évolution des *produits-modèles*

Les limites des *LMPs* soulignées ci-haut impliquent que les relations entre *produits-modèles*

ne peuvent pas être décrites et qu'aucune sémantique ne peut alors leur être associée lors de la description d'un *procédé*. Un *moteur d'exécution* ne peut donc pas exploiter de telles relations (alors inexistantes pour lui) dans le cadre de la gestion de l'évolution des *produits-modèles* qu'il est chargé d'assurer. Par ailleurs, divers systèmes sont utilisés par ces *moteurs d'exécution* pour assurer la gestion de l'évolution des *produits* d'un *procédé* lorsque celui-ci s'exécute. Altmanninger et al. [Altmanninger et al. 2009] indiquent que dans le cadre de *procédés* représentant des projets d'envergure, l'utilisation des *CMSs* est incontournable pour assurer cette gestion. A ce niveau, il est donc important de signaler l'existence de plusieurs *CMSs* dans la littérature. Certains de ces *CMSs* [Reidar Conradi & Westfechtel 1998; W. F. Tichy 1985; Lampson & Schmidt 1983; Fowler et al. 1995; Prieto-Díaz Rubén 1986; Estublier & Casallas 1994] s'appuient, pour assurer la gestion de l'évolution de *produits*, sur des modèles de données qui prennent en compte les liens de composition et de dépendance. Ils sont utilisés dans des domaines très variés comme la gestion des versions de fichiers et répertoires [W. F. Tichy 1985; Fowler et al. 1995], de modules de programmes [Lampson & Schmidt 1983; Prieto-Díaz Rubén 1986], et des objets de bases de données [Estublier & Casallas 1994]. Dans le contexte initial où les *produits de procédés* sont décrits comme des fichiers sans aucune considération de leur structure interne, ces *CMSs* supportent avec aisance la gestion de leur évolution en s'appuyant sur leur modèles de données de base. Seulement, comme nous le montrerons également dans l'état de l'art, même si des efforts sont en train de donner leurs fruits [H. Oliveira et al. 2005; Farah et al. 2009; Murta et al. 2007; Prawee Sriplakich et al. 2008], nous n'avons pu trouver dans l'existant aucun système *CMS* capable de prendre en charge des *produit-modèles* en considérant entre eux les relations que nous avons identifiées ci-haut.

Les *moteurs d'exécution*, parce que associés à des *LMPs* limités, ne disposent d'aucune possibilité leur permettant d'exploiter les relations que nous avons identifiées. Ils ne peuvent ainsi pas assurer la gestion optimale des *produits-modèles* à l'exécution des *procédés*. Une exploitation de ces relations permettrait, en effet, une évolution automatique et dynamique et une bonne *cohérence relationnelle* pour certains *produits-modèles*, une meilleure modularité pour d'autres, etc. Nous illustrons cela à travers les exemples de la section suivante.

2.3.3.3 Exemple et Illustration

2.3.3.3.1 Le procédé-exemple

Le procédé-exemple que nous décrivons ici sera utilisé dans toute la suite du document. Cet exemple est tiré de [Alexander Egyed 2007]. C'est un extrait d'un *procédé* plus global correspondant à celui de développement d'un système de vidéo à la demande (VOD, pour video-on-demand). Il contient les deux activités d'*analyse* (*Analysis*) et de *conception* (*Design*) qui sont contenues dans une itération. L'*analyse* produit un *produit-modèle* (le *modèle d'analyse*) qui est pris en entrée par l'activité de *conception*, qui elle en fournit un autre (le modèle de conception). La **Figure 2.4** contient une vue schématique de ce *procédé* simple.

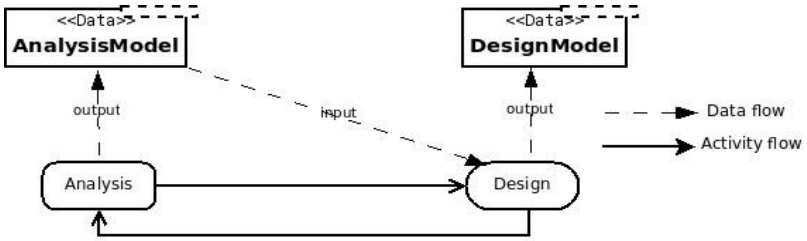


Figure 2.4: Vue Schématique du Procédé-Exemple

Les *produit-modèles* du système VOD sont des *modèles* UML 2. Ils sont présentés Figure 2.5. Le *modèle d'analyse* (Figure 2.5.a) est composé des cas d'utilisation du système et de ses classes d'analyse. Le *modèle de conception* (Figure 2.5.b) comprend à son tour les classes de conception du système ainsi que des éléments d'interaction.

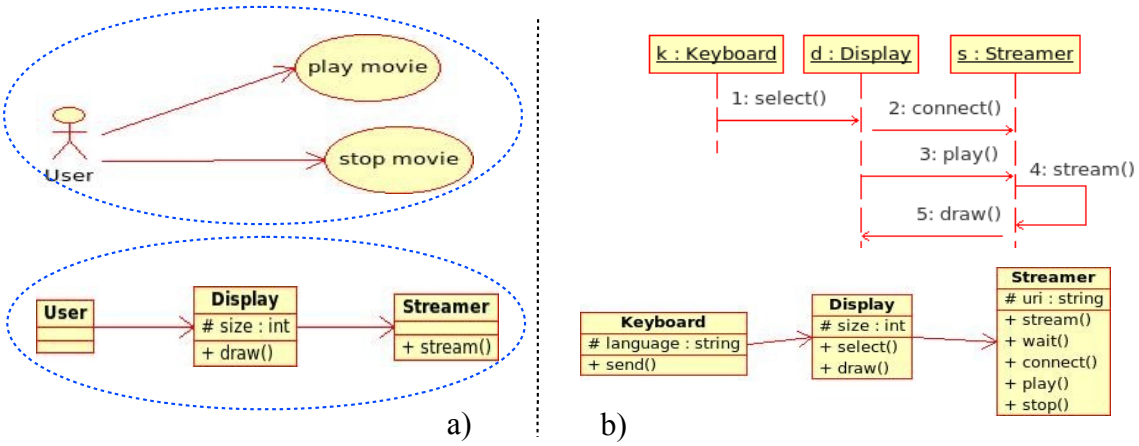


Figure 2.5: Modèles d'Analyse et de Conception du Système VOD

2.3.3.3.2 Exemples illustrateurs

Dans cette section nous utilisons le procédé-exemple précédemment défini pour présenter quelques illustrations pouvant aider à une meilleure compréhension de la problématique soulevée.

Tel que sont structurés les *produits-modèles* du procédé-exemple, nous avons les deux constatations suivantes les concernant.

- Considérant le *modèle d'analyse*, on peut distinguer la partie correspondant aux éléments relatifs aux cas d'utilisation et l'autre correspondant aux classes d'analyse et leurs associations. Sur le schéma de la **Figure 2.5.a**, ces deux parties sont délimitées à l'aide des ellipses. Le *moteur d'exécution* n'a aucune information lui permettant d'accéder à une de ces deux parties sans accéder à l'autre à la fois. Or, dans le cas d'une relation précise comme celle de partage d'éléments que nous décrirons dans la suite, tout le *modèle d'analyse* n'est pas impliqué, mais seulement sa partie correspondant aux classes d'analyse. Nous avons donc là un problème de « modularité » du *produit-modèle* lié à la *granularité* avec laquelle il a été spécifié. Nous reviendrons sur ce point avant de finir cette section.
- Considérant le *modèle de conception*, certains de ses éléments de modèles sont également des éléments du *modèle d'analyse*. C'est le cas des classes *Display* et *Streamer*, des attributs *size*, et de la méthode *stream()*. Cette même constatation est

également valable entre les deux parties du *modèle de conception* correspondant respectivement aux interactions et classes de conception; en effet les messages contenus dans le diagramme d'interactions correspondent également à des éléments du diagramme de classe de conception. Ainsi, à chaque fois que, par exemple, la classe *Display* du *modèle de conception* est modifiée, la classe *Display* du *modèle d'analyse* peut aussi l'être. Une synchronisation de ces deux *produits-modèles* est donc nécessaire pour maintenir la *cohérence relationnelle* qui doit exister entre eux.

De façon plus générale, au cours du déroulement de l'activité d'*analyse*, lorsqu'un élément de modèle du *modèle de conception* est modifié, il faut que le *moteur d'exécution* du *procédé* réalise les étapes suivantes.

- Vérifier si une telle modification a un impact et doit ou non être répercutée sur d'autres éléments de modèles du *modèle d'analyse*.
- Si oui, exécuter les actions permettant de répercuter la modification sur les éléments de modèle concernés.

Les étapes précédemment décrites en 1) et 2) peuvent se réaliser manuellement sur des *produits-modèles* de très petite taille. Dans le cas de projets industriels avec des *produits-modèles* de taille importante par contre, les actions correspondantes explosent facilement de façon exponentielle, leur gestion manuelle devient alors irréaliste. Il faut donc une exécution automatique de ces actions par le *moteur d'exécution* alors qu'aucune information n'est disponible pour lui à cet effet.

Toujours dans le but d'illustrer l'intérêt de disposer d'informations détaillées sur les relations entre *produits-modèles* d'un *procédé* en cours d'exécution, nous considérons une opération de transformation *T* quelconque, de même que le *modèle d'analyse* du procédé-exemple. Nous considérons toujours l'hypothèse suivant laquelle le *modèle d'analyse* et le *modèle de conception* sont liés par une relation d'*overlap* dont la sémantique associée est la suivante: *«les classes du modèle d'analyse sont aussi présentes dans le modèle de conception».*

Des hypothèses précédentes peuvent être déduites les règles de gestion suivantes:

- Si *T* appliquée au *modèle d'analyse* ne modifie aucune de ses classes, alors l'action correspondante n'a pas d'effet sur le *modèle de conception* au vu de la relation entre les deux *produits-modèles*.

- Si, par contre, *T* modifie une ou plusieurs classe(s) d'analyse, alors les mêmes modifications doivent être exécutées sur les éléments correspondants du *modèle de conception*.

Nous pensons que de telles règles de gestion sont importantes pour un *moteur d'exécution* pour jouer son rôle de maintien de la *cohérence relationnelle* entre les deux *produits-modèles* considérés du procédé-exemple. Par ailleurs la sémantique de l'*overlap* entre les deux *produit-modèles* permet également au *moteur d'exécution*, au moment de la création du *modèle de conception*, de proposer une aide au développeur responsable puisque durant son activité il a, de toutes les façons, soit à recréer des éléments de modèles (les classes) déjà présentes dans le *modèle d'analyse*, soit à les copier/coller.

Pour finir, considérons le *modèle d'analyse*. Celui-ci contient deux partitions distinctes que nous allons, pour simplifier, appeler des *diagrammes*: d'une part le d*iagramme de cas d'utilisation* correspond aux éléments de modèle qui sont des cas d'utilisation, acteurs, etc. D'autre part, le *diagramme de classes* contient les classes, attributs, associations, etc. Reprenons la transformation *T* et considérons maintenant qu'elle ne s'intéresse qu'aux classes du *modèle d'analyse* lorsqu'elle est invoquée sur celui-ci. Dans cette nouvelle configuration, le *moteur d'exécution* n'a besoin de passer à *T* que la partition du *modèle d'analyse* correspondant au *diagramme de classes*. Afin de permettre cela au *moteur d'exécution*, il est naturel de penser à une spécification du *produit-modèle* avec une *granularité* plus fine qui alors implique, à la place de celle du *modèle d'analyse*, une spécification des deux *diagrammes* sous la forme de deux *produits-modèles*. Seulement avec une telle solution on perdrait le niveau de *granularité* permettant de définir le modèle-somme correspondant au *modèle d'analyse* auquel le *moteur d'exécution* ne pourrait plus accéder en cas de besoin.

Avec l'existant en terme de modélisation et d'exécution de *procédé*, un *moteur d'exécution* ne dispose donc d'aucune possibilité d'avoir, à la fois, les deux niveaux de *granularité* permettant d'accéder aux parties (les diagrammes) et à la somme du *modèle d'analyse*.

2.3.3.3.3 Synthèse

Il apparaît, à la lumière des exemples précédents les deux éléments décrits suivants.

- Les relations entre *produits-modèles* de *procédés* et les détails sur la structure interne de ceux-ci sont actuellement sous-considérés par les *LMPs* existants de même que par

les *moteurs d'exécution* associés à ces langages.

- Une fois des relations établies et bien formalisées entre les *produits-modèles* d'un *procédé*, elles deviennent un très bon support productif pour la gestion de la *cohérence relationnelle*, et pour l' évolution automatique des *produits-modèles*. Elles rendent également possible une gestion de ces *produits-modèles* avec une plus grande flexibilité sur leur niveau de *granularité*.

Par ailleurs les *produits* sont des *produits-modèles* dans le contexte IDM, il devient alors possible de décrire leur structure interne. En effet un *produit-modèle* est un ensemble cohérent d'éléments de modèles et est structuré par un *méta-modéle* qui à son tour est un ensemble de méta-classes reliées entre elles. Cette situation est favorable à la définition d'une sémantique claire pour les relations entre les *produits-modèles* d'un *procédé* dans ce contexte. Par exemple, un *produit-modèle* peut être subdivisé en différentes parties de modèles correspondant chacune à une partition d'éléments de modèle logiquement liés entre eux selon le *modèle du procédé*. De même, il est possible de spécifier un *overlap* entre *produits-modèles* différents.

2.4 Conclusion

Dans ce chapitre nous avons donné les définitions des concepts fondamentaux pour la compréhension de cette thèse. Nous avons mis l'accent sur le contexte IDM qui constitue celui dans lequel le travail se place. Nous avons ensuite défini la problématique de la thèse qui est relative à l'absence d'une prise en compte des relations entre *produits-modèles* de *procédés* à la modélisation et à l'exécution. Notre objectif est donc de fournir une solution de modélisation des relations entre *produits-modèles*, exploitable à l'exécution par un *moteur d'exécution*. Notre travail se focalise sur deux relations: le *nest* et l'*overlap*.

Dans le chapitre suivant, nous présentons l'état de l'art sur le support de la modélisation des *produits* de *procédés* et de leurs relations, ainsi que celui sur la gestion de l'évolution et de la cohérence de ces *produits*.

Chapitre 3

Relations entre *Produits de Procédés*: Etat de l'Art

3.1 Introduction

Le chapitre précédent a défini les concepts fondamentaux utilisés tout le long du document. Il a également indiqué le contexte ainsi que la problématique de la thèse. Le problème identifié est relatif au manque de considération, à la modélisation comme à l'exécution des *procédés*, des relations entre les *produits* dans un contexte IDM où ce sont essentiellement des *produits-modèles*. Cette situation n'est pas favorable à une gestion optimale des *produits-modèles* à l'exécution des *procédés*. Les problèmes de gestion de *produits-modèles* identifiés sont mis en exergue au chapitre 2. Ils sont relatifs au maintien de la *cohérence relationnelle* entre les *produits-modèles* de même qu'à leur modularité et à leur évolution dynamique et automatique.

Dans le but d'illustrer la problématique soulevée, mais aussi de nous servir de l'existant comme base pour construire notre contribution, ce chapitre présente le support des relations entre *produits* dans différents domaines connexes à notre thématique. C'est ainsi que nous nous sommes intéressés aux *LMPs* et aux *moteurs d'exécution* qui leur sont associés, ainsi qu'aux systèmes de gestion d'évolution de *produits* que sont les *VCSs*/*CMSs*. Nous identifions chaque fois les relations supportées et leur prise en charge à l'exécution, notamment en terme de gestion de la *cohérence relationnelle*, de l'évolution dynamique et automatique, et de la modularité de *produits-modèles*.

3.2 La modélisation et l'exécution de *procédés*

Plusieurs types d'informations doivent être intégrés pour décrire adéquatement un *procédé*. A cet égard, un *modèle de procédé* décrit ce qui doit être fait, qui doit le faire, où et quand, etc. Il décrit également les dépendances entre ses différents éléments. Il existe plusieurs *LMPs* qui diffèrent, entre autres, par le fait qu'ils couvrent une ou plusieurs perspectives. La perspective

à laquelle nous nous intéressons ici est celle relative aux entités informationnelles produites ou manipulées au cours du déroulement d'un *procédé*. En effet, c'est cette perspective qui s'intéresse à la représentation de la structure des *produits* ainsi que des relations entre eux. Cette représentation se fait avec un niveau de détail plus ou moins important selon les possibilités et orientations du langage considéré. Dans cette section, nous allons présenter les *LMPs* que nous jugeons représentatifs de la littérature sur la modélisation de *procédés*. En effet, ces approches se basent presque chacune sur une philosophie différente de celle des autres pour modéliser et exécuter un *procédé*. Pour préciser cet aspect, nous faisons suivre ci-après la classification des approches que nous allons présenter.

- 1 approche orientée « *programmation de procédés »*: APPL/A [Sutton et al. 1990]. La notion de « *programmation de procédés »* a été développée par Osterweil dans [Leon Osterweil 1987]:
- 1 basée sur des règles : MSL/MARVEL [G. Kaiser et al. 1990].
- 1 basée sur des réseaux de Petri : SLANG/SPADE [Bandinelli et al. 1994].
- 1 basée sur un *CMS*/*VCS* : TEMPO/ADELE [Belkhatir et al. 1992].
- 2 basées sur UML : Di Nitto et al. [Di Nitto et al. 2002], Chou [Chou 2002]. Ces approches utilisent UML (versions 1.3 et 1.4, respectivement) pour construire le moyen d'une modélisation des *procédés* avec un haut niveau d'abstraction. Elles définissent ensuite un autre niveau de représentation constitué d'un langage de programmation. Les *modèles de procédés* conçus en UML y sont alors projetés pour être exécutés.
- 2 basées sur UML 2.0 et les techniques de la méta-modélisation : UML4SPM [Bendraou et al. 2006] et SPEM4MDE [Diaw et al. 2010].

De plus, certaines de ces approches sont anciennes, qualifiées de première génération (APPL/A, MSL/MARVEL, SLANG/SPADE, TEMPO/ADELE) et les autres plus récentes (Di Nitto et al., Chou, UML4SPM).

Pour chacune des approches présentées, nous décrivons succinctement la philosophie qui sous-tend la modélisation et, le cas échéant, l'exécution d'un *procédé*. Nous mettons ensuite l'accent sur la prise en compte des *produits* eux mêmes, et des relations inter-*produits* au sein de l'approche à la modélisation comme à l'exécution. Nous finissons l'étude de chaque approche par une mise en exergue de son adaptabilité ou non au contexte IDM où les *produits* sont des *produits-modèles*. Pour rappel, nous nous focalisons dans cette étude aux relations

nest et *overlap* définies au chapitre 2.

3.2.1 APPL/A

3.2.1.1 Présentation générale de l'approche

APPL/A [Sutton et al. 1990] est un *LMP* orienté *« programmation de procédés logiciels »*, un paradigme introduit par Osterweil dans [Leon Osterweil 1987]. APPL/A est une extension du langage Ada auquel l'auteur ajoute des concepts spécifiques aux *procédés* et qui sont relatifs à la modélisation des données, la gestion des changements dans un *procédé*, l'expression des dépendances entre éléments d'un *procédé*, la persistance ainsi que la gestion de la cohérence de données.

La motivation principale de la création d'APPL/A est strictement liée aux nombreux changements qui très souvent se produisent au cours du déroulement d'un *procédé*. Mais ces changements sont considérés dans la conception de ce langage comme pouvant provenir des *produits*, du *procédé* lui-même (ses activités), ou de l'environnement d'exécution du *procédé*. Pour venir à bout des problèmes relatifs aux changements qui surviennent au cours du déroulement d'u *procédé*, APPL/A propose une représentation la plus explicite et exhaustive possible des *procédés* de développement en utilisant des programmes directement exécutables.

3.2.1.2 Les produits et les relations entre eux

APPL/A propose:

- une représentation explicite des *produits* et des relations entre eux,
- une représentation explicite de la sémantique des *relations* entre les *produits*, et entre les *produits* et les autres éléments de *procédés*,
- une automatisation du processus de changement incluant une propagation des données et une gestion de la cohérence.

Pour réaliser son objectif, APPL/A s'appuie sur un modèle de donnée selon lequel chaque *produits* est représenté à travers une relation qualifiée de programmable et persistante. Une relation peut également être utilisée pour traduire un lien de dépendance entre *produits* différents. Chaque relation est définie sous la forme d'une unité de programme qui contient deux parties, une spécification et un corps, que nous présentons ci-après.

La spécification d'une relation contient les définitions des types des éléments impliqués. Ces

éléments sont des *produits* ou parties de *produits* et peuvent ou non être dérivés les uns des autres. Une spécification de relation contient également des déclarations d'opérations (insertion, modification, suppression, recherche, etc.) sur la relation, ainsi que de liens de dépendances entre les différents éléments. La **Figure 3.1** montre le code correspondant à la spécification APPL/A d'une relation nommée *Word_Count*. Cette spécification définit une relation de dérivation entre des objets *text* et leurs nombres de lignes (*lines*), mots (*words*) et caractères (*characters*) calculés à l'aide d'un outil *wc*. Le type ***tuple*** détermine les noms et types des éléments de la relation sous la forme d'attributs. Les opérations sont matérialisées sous forme d'entrées (***entry***). Enfin, la spécification déclare une dépendance (*dependencies*) entre les éléments *text* et leurs nombres de lignes (*lines*), mots (*words*) et caractères (*characters*) obtenus avec l'aide d'un outil *wc*.

```
with WC;    – separate "word-count" tool
with text_def; use text_def;

Relation Word_Count is
    type wc_tuple is tuple
        text: in text_type;
        lines, words, characters: out natural;
    end tuple;
entries
    entry insert(text: in text_name);
    entry delete(text: text_type;
        lines: natural;
        words: natural;
        characters: natural);
    entry find(iterator: in out integer := 0;
        found: out boolean;
        first: boolean;
        t: out wc_tuple;
        select_text: boolean; text: text_type;
        select_lines: boolean; lines: natural;
        select_words: boolean; words: natural;
        select_characters: boolean;
            characters: natural);
dependencies
    determine lines, words, characters
        by wc(text, lines, words, characters);
End Word_Count;
```

Figure 3.1 : Spécification APPL/A de la Relation Word_Count [Sutton et al. 1990]

Contrairement à la spécification qui correspond à la partie abstraite (ou la définition en intention) réservée à la description de la relation, le corps quant à lui désigne sa partie concrète (ou sa définition en extension). Il contient et gère la persistance des données de la relation ainsi que des implémentations des différentes opérations déclarées dans la spécification.

APPL/A supporte également la définition de déclencheurs. Un déclencheur est associé à une opération d'une relation et permet de propager des changements d'une relation à une autre, envoyer des messages en réponse aux changements de données, exécuter d'autres programmes, etc.

3.2.1.3 Adaptabilité au contexte IDM

L'étude que nous avons faite de APPL/A et qui est résumée ci-avant nous a permis de faire les conclusions suivantes quant à l'adaptabilité ou non de ce langage au contexte IDM.

- La relation de dépendance qui est prise en compte par ce langage peut s'adapter au contexte des *produits-modèles* pour exprimer des liens de *dérivation* entre eux.
- Lors de la définition du type d'un *produit* sous la forme d'un tuple, il est possible de définir des attributs dont chacun peut être atomique ou composé. Il est donc possible d'exprimer une relation *nest* entre différents *produits-modèles*. Toutefois, aucune possibilité d'associer une caractéristique additionnelle à la relation dans le but de mieux la spécifier n'est offerte dans APPL/A.
- APPL/A ne supporte aucune possibilité permettant de définir une relation de type *overlap*.
- Une gestion de la *cohérence relationnelle* entre les *produits-modèles* d'un *procédé* modélisé avec APPL/A peut être basée sur l'exploitation de la relation de dépendance dans ce contexte ainsi que sur les déclencheurs. Cette cohérence n'est par contre pas celle que nous adressons ici ; elle est de niveau *produit-modèle* et est semblable à celle ciblée par les approches de synchronisation de *modèles* que nous avons évoquées au niveau du chapitre 2.
- Aucune forme de *cohérence relationnelle* relative au partage d'éléments entre *produits-modèles* ne peut être garantie lors de l'exécution de programmes APPL/A.
- L'utilisation de déclencheurs rend possible une automatisation de l'évolution des *produits-modèles*. Par contre, celle-ci ne peut pas tirer profit de l'existence d'éléments communs. Elle n'est donc pas dynamique.

3.2.2 MSL/MARVEL

3.2.2.1 Présentation générale de l'approche

MSL (Marvel Strategy Language) est un langage qui permet de modéliser un *procédé* en

utilisant des règles [G. Kaiser et al. 1990]. Cette philosophie est très répandue dans le monde des systèmes experts et est à la base de langages comme Prolog. L'idée du langage MSL est que chaque activité du *procédé* à modéliser est représentée par une règle qui est composée de trois parties:

- une pré-condition: c'est une expression logique composée de clauses qui renseignent sur l'état du *procédé* à travers celui de tous ses éléments (activités, *produits*, agents, etc.). La pré-condition doit être vérifiée pour que l'activité puisse commencer.
- l'activité elle-même: elle correspond à l'invocation d'outils, à la description des tâches à faire, ou à la définition (ou la sélection) d'autres règles nécessaires à l'activité en cours de description ou à celles qui la suivent.
- une ou plusieurs post-conditions composées de clauses qui correspondent à des assertions sur les éléments du *procédé* et dont la vérification marque la fin de l'activité.

Le langage MSL est textuel. L'unité de modularité dans MSL est appelée *stratégie*. La **Figure 3.2** montre la forme générique d'une stratégie qui contient des définitions d'une interface en terme d'imports et d'exports, des classes, des relations entre les classes, des outils, et enfin des règles. Un *modèle de procédé* représente une collection d'unités en interaction entre elles, à la manière des modules d'un programme écrit dans un langage de programmation conventionnel. L'idée est de représenter chaque phase du cycle de développement à l'aide d'un ensemble de stratégies, avec la possibilité que certaines stratégies soient partagées entre différentes phases. Dans MSL, des classes sont utilisées pour modéliser tous les éléments d'un *procédé*.

```
strategy name
imports list of imported strategies
exports list of exported classes, tools, relations and rules

objectbase
  classes...
  relations...
  tool_name :: superclass TOOL ;
     operation_name : string=file_name ;
     operations....
  end
  tools...
end_objectbase
rules
  rules...
```

Figure 3.2: Code Générique d'une Stratégie dans MARVEL [G. Kaiser et al. 1990]

MSL est associé à l'environnement d'exécution appelé MARVEL [G. Kaiser et al. 1990; Barghouti 1992]. MARVEL accède aux éléments d'un *procédé* via des objets qui leur correspondent et qui sont décrits sous forme de classes MSL. L'architecture de MARVEL fait ressortir deux composants essentiels : l'un correspond à une base d'objets appelée *objectbase*, et l'autre correspond à celui de gestion de *modèle de procédé*. Ce dernier est un ensemble de règles qui sont relatives au déroulement et à la coordination des activités d'un *procédé*. Il tire ses données de *l'objectbase* qui alors représente, sous forme d'objets (au sens orienté objet du terme), l'ensemble des éléments du *procédé*. L'*objectbase* correspond ainsi à une représentation (logique) en mémoire centrale des éléments du *procédé* qui sont physiquement stockés dans le système de fichiers. Chaque objet de l'*objectbase* est l'instance d'une classe qui modélise l'élément de procédé qu'il représente.

3.2.2.2 Les *produits* et les relations entre eux

Comme nous l'avons déjà souligné, des classes sont utilisées pour modéliser les *produits*. Certaines relations entre *produits* sont représentées sous forme d'attributs dans les classes qui les modélisent. D'autres relations sont définies séparément par rapport aux classes comme cela apparaît dans la **Figure 3.2**. Les relations supportées par l'approche sont:

- la composition, le fait qu'un *produit* soit un composant d'un autre,
- la dérivation, le fait qu'un *produit* (par exemple, un module) soit dérivé d'un autre (par exemple, une unité compilée), ou de plusieurs autres. Une relation de dérivation entre deux (ou plus) objets représentants des *produits* peut faire intervenir un autre objet (par exemple, un compilateur) représentant un outil du *procédé*.

Le problème de la granularité des objets définis avec MSL est très bien adressé dans [Feiler & G. E. Kaiser 1987] qui souligne l'importance d'une modularité adéquate des *produits*. En effet cette modularité permet, selon les auteurs, de gérer les *produits* en tant qu'éléments séparés les uns des autres pour avantager les activités de contrôle de l'exécution de même que pour améliorer la performance de leur gestion.

MARVEL supporte un accès séparé, aux modules, procédures, types, et variables globales pour ce qui est des programmes, aux sections et sous-sections des documents, aux plans, tâches, agents pour ce qui est des plans de gestion, etc. Pour MARVEL, en effet, la modularité des *produits* est indispensable pour une performance des activités de contrôle de l'exécution, de même que du point de vue de la gestion de ces *produits* [Feiler & G. E. Kaiser 1987]. Les

relations de dérivation quant à elles sont exploités par MARVEL pour construire certains *produits* à partir d'autres avec l'aide des outils impliqués. Elles sont également utilisées pour maintenir les *produits* liés synchronisés les uns par rapport aux autres.

3.2.2.3 Adaptabilité au contexte IDM

Nous indiquons ci-après les possibilités, mais aussi les limites qui caractérisent cette approche dans le cas des *produits-modèles*.

- Un lien de type *nest* peut être défini entre deux *produits-modèles* sous la forme d'un lien classe-attribut sous MSL. De plus, si cela est nécessaire, il faut que cette définition soit complétée par une autre relation impliquant les mêmes *produits-modèles* et définie dans le seul but de spécifier les caractéristiques du lien.
- Il est possible de spécifier un lien de type *overlap* entre *produits-modèles* en définissant une relation MSL qui implique les classes qui les modélisent.
- Du point de vue de l'exécution, en ne prenant en compte que des types de *produits* prédéfinis tels que « module », « procédure », « manuel d'utilisateur », etc., l'environnement MARVEL est plutôt conçu pour gérer des *produits* relatifs aux dernières phases d'un *procédé* (codage, recette, etc.). Pour les autres phases (analyse, conception), l'approche définit des types inadaptés (exemple : « description conceptuelle ») selon lesquels les *produits-modèles* impliqués sont considérés comme des fichiers dont la structure interne est inaccessible. Entre l'*objectbase* et le système de fichier, MARVEL dispose de mécanismes adaptés à la sauvegarde/restauration de *produits* dont les types sont supportés. Ces mécanismes n'étant pas adaptés aux *produits-modèles*, l'environnement ne peut assurer une gestion de leur modularité, ni leur évolution dynamique, encore moins une *cohérence relationnelle* entre eux.

3.2.3 SLANG/SPADE

3.2.3.1 Présentation générale de l'approche

SLANG (**S**pade **LANG**uage) [Bandinelli et al. 1994] est un *LMP*. C'est un formalisme de haut niveau basé sur les réseaux de Petri. Il permet de structurer de façon modulaire des *modèles de procédés*. Du point de vue architectural, SLANG se compose de deux parties principales: le noyau (Kernel SLANG) qui définit les éléments de base du langage, et l'extension (Full SLANG ou SLANG) dans laquelle sont définis les concepts spécifiques aux *procédés*.

Kernel SLANG s'appuie sur le modèle orienté objet pour définir les éléments de base du langage. Il offre, par exemple, la possibilité de définir des jetons (*tokens*) et des *places*. Les jetons représentent des objets typés sur lesquels il est possible d'exécuter des opérations définies dans les types correspondants. La hiérarchie des types SLANG fournit ainsi le type prédéfini **Token** qui alors représente la racine de tous les types représentant des jetons d'un *modèle de procédé*. Une *place* est définie pour contenir des jetons. Les *places* servent de base à la construction de réseaux. En effet, le noyau de SLANG permet également la définition de réseaux impliquant des *tokens* et des *places* déjà définis.
En s'appuyant sur les éléments définis dans Kernel SLANG, Full SLANG définit un ensemble d'éléments permettant de représenter l'évolution de *procédés* mais aussi des notations dont le but est d'accroître l'utilisabilité du langage pour ses utilisateurs. C'est ainsi qu'un *modèle de procédé* dans Full SLANG se présente sous la forme de la paire suivante: *SLANGProcessModel=(ProcessTypes, ProcessActivities)*. Dans ce couple, *ProcessTypes* représente un ensemble de définitions (textuelles) de types utilisant le système défini dans le noyau. *ProcessActivities* est, quant à lui, un ensemble de définitions d'activités. Les activités correspondent à un ensemble de réseaux définis de la manière indiquée dans le noyau.
SLANG est associé à SPADE [Bandinelli et al. 1994]. SPADE est un environnement intégré de modélisation, d'analyse et d'exécution de *procédés* définis à l'aide du langage SLANG.

3.2.3.2 Les *produits* et les relations entre eux

Dans Kernel SLANG toutes les données sont typées. En effet Kernel SLANG définit un système de typage de style orienté objet très riche et qui permet la modélisation des *produits* d'un *procédé*. Le système de typage utilisé prend en compte le fait que ces *produits* peuvent être de nature complexe et variée : objets graphiques, rapport de bugs, données de test, code exécutables, etc. C'est à cet effet que SLANG offre la possibilité de s'appuyer sur des types de base pour définir des structures de données complexes qui de plus permettent de définir des relations entre les *produits*. La définition d'un type contient le nom du type, une spécification de la hiérarchie qui contient le type, ainsi qu'une liste d'attributs et d'opérations. La modélisation d'une relation de composition entre *produits* est faite à l'aide des attributs de types complexes. Celle des relations de dépendance entre différents *produits* est, quant à elle, assurée à travers des réseaux impliquant les *tokens* qui les modélisent dans le *procédé*.
L'environnement SPADE, gère tous les éléments d'un *procédé* (*produits*, activités, rôles, etc.)

en exécution à l'aide d'une base de données orientée objet. Il est capable d'accéder aux *produits* selon les différents niveaux de granularité avec lesquels ils ont été définis à la modélisation. Il permet également de satisfaire les dépendances entre *produits* en interprétant les réseaux qui définissent ces dépendances.

3.2.3.3 Adaptabilité au contexte IDM

- La relation de composition qu'il est possible d'exprimer avec SLANG à travers le lien classe-attribut est adaptée à la définition d'une relation de type *nest* entre des *produits-modèles*. De plus, la définition d'une composition exprimée sous la forme d'un lien type-attribut peut être complétée à l'aide d'une opération définie avec le type.
- Le formalisme SLANG ne supporte pas la spécification d'une relation de type *overlap* entre deux *produits-modèles*.
- SPADE utilise un système de base de données O2 [O2 Technology 1992] pour le stockage des *produits* et ne définit aucun mécanisme lui permettant de supporter une gestion de *produits-modèles*. Il ne peut, par conséquent, ni identifier des éléments de modèles communs à deux ou plusieurs *produits-modèles*, ni se servir d'une relation *nest* spécifiée, etc. au cours de l'exécution d'un *procédé*.
- Du fait que SPADE interprète les réseaux correspondant à un modèle de procédé et que ceux-ci détermine les actions à effectuer sur les produits, les *produits-modèles* peuvent évoluer automatiquement même si leur structure interne est inaccessible au système de gestion.

3.2.4 TEMPO/ADELE

3.2.4.1 Présentation générale de l'approche

TEMPO [Noureddine Belkhatir et al. 2007; Noureddine Belkhatir, Walcelio L Melo, et al. 1992; N Belkhatir et al. 1993] est un formalisme de modélisation de *procédés* créé au dessus du noyau du *CMS* ADELE [Noureddine Belkhatir, Walcélio L Melo, et al. 1992; Estublier & Casallas 1994]. Il est orienté objet, et s'appuie sur une modélisation du comportement contextuel des éléments d'un *procédé*. En effet, le formalisme se base sur le concept de *rôle*, qui est analogue à celui plus connu de *vue*. Un *rôle* est défini ici comme étant une *nouvelle définition des propriétés et du comportement d'un objet-type dans un sous-procédé donné*. TEMPO définit n'importe quelle étape d'un *procédé* comme une liste d'objets y jouant chacun

un rôle précis. Selon cette philosophie, le comportement d'un objet n'est pas statique mais est contextuel à l'étape considérée du *procédé* et du *rôle* qu'il y joue. Un même objet peut être simultanément utilisé dans plusieurs espaces de travail différents, avec des *vues* différentes. TEMPO définit, les mécanismes de synchronisation et de communication nécessaires à une telle possibilité [Noureddine Belkhatir, Walcélio L Melo, et al. 1992].

Dans le formalisme, un *procédé* est vu comme une combinaison de sous-procédés dont chacun correspond à un ensemble défini d'activités du *modèle de procédé* et fait intervenir un nombre précis d'agents, d'objets, et de règles dont certaines assurent la communication et la synchronisation du sous-procédé avec les autres du même *procédé*.

Le moteur de procédé associé à TEMPO est une extension de ADELE dont les auteurs soutiennent qu'*une bonne gestion de configuration a besoin d'un bon support de la notion de procédé* [Estublier & Casallas 1994] pour motiver l'extension. A côté de son noyau, ADELE dispose d'une couche de « services ». C'est cette dernière couche qui intègre le support de *procédé* à travers trois principaux éléments dont un moteur de procédé. Le rôle principal du *moteur d'exécution* est d'exécuter des règles dites Événement/Condition/Action (ECA) qui sont intégrées dans les définitions de types sous forme de déclencheurs, en s'appuyant sur ADELE.

3.2.4.2 Les *produits* et les relations entre eux

Avec TEMPO, chaque sous-procédé est décrit par un procédé-type. La définition d'un procédé-type contient son nom et la définition des *rôles* qui lui correspondent (voir la première partie de la **Figure 3.3**). Celui-ci contient la définition des *produits* sous la forme des *rôles* qui leur correspondent dans le sous-procédé. Chaque *rôle* a un nom, un type de référence, des attributs et des méthodes locales, ainsi qu'une liste de règles. La deuxième partie de la **Figure 3.3** contient un exemple illustratif de définition de *rôles* tiré de [Noureddine Belkhatir, Walcelio L Melo, et al. 1992]. Elle montre deux sous-procédés dont un (*WS-change*) dans lequel les objets *module* ont un rôle appelé *view*. Ils sont isolés durant toute la durée du sous-procédé. Ces objets sont alors stockés dans l'espace de travail dédié au sous-procédé. Les modifications qui leurs sont opérées ne seront visibles des autres sous-procédés que si leurs auteurs décident de leur propagation. Un mécanisme **d'alerte** permet alors d'avertir les autres sous-procédés.

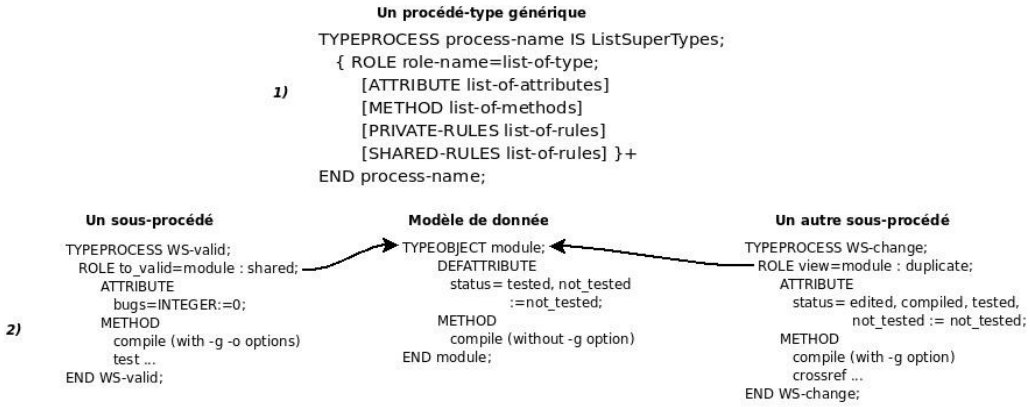


Figure 3.3: Exemple de Définition de deux Rôles avec TEMPO

TEMPO utilise le modèle de donnée de ADELE pour modéliser les *produits*. ADELE quant à lui repose sur un modèle de données entité-relationnel, enrichi de concepts orientés objets. Dans ADELE, les entités-types et relations-types sont définies séparément les unes des autres. Les objets (instances des entités-types) et les relations (instances des relations-types) sont traités de la même manière. Les objets peuvent être des fichiers, activités, fonctions, ou des éléments simples comme des chaînes de caractères, dates, etc. Une relation est établie entre un objet d'origine (O) et un objet de destination (D) et peut correspondre à une dérivation, une dépendance, ou une composition. Comme chaque objet, chaque relation dispose d'un nom externe, ce nom correspond à la chaine de caractère (O|R|D) qui est la concaténation de trois autres chaines de caractères correspondant respectivement au nom de l'objet source (le O), celui de la relation-type (le R), et celui de l'objet cible (le D).

Un type définit les attributs et méthodes des instances de ce type. Les relation-types peuvent en plus définir des informations supplémentaires relatives au domaine de la relation. Le domaine d'une relation correspond à un ensemble de contraintes sous lesquelles des entité-types peuvent être liées par la relation. Un attribut peut être de n'importe quel type y compris celui d'un objet. Il peut également être de type ensemble. Un attribut de type complexe correspond à une composition spéciale; il est à chaque fois associé à une relation de même nom qui en précise les éventuels éléments structurels et comportementaux.

ADELE est vu comme une machine virtuelle de *procédés* spécifiés avec TEMPO. En effet, le *moteur d'exécution* contenu dans son extension peut lire une spécification TEMPO et translater ses concepts orientés objet vers ceux orientés entité-relation de ADELE. C'est ainsi que les procédé-types sont représentés par des entité-types, les *rôles* par des relations dites

actives (car réagissant aux évènements qui se déroulent au sein des objets origines et destinations) [Noureddine Belkhatir et al. 2007], les règles par des déclencheurs associés aux entités et relations, et enfin les méthodes et actions par du code ADELE. Le noyau ADELE est une base de données relationnelle étendue par des concepts orientés-objet et un gestionnaire d'activités qui se base sur un mécanisme de déclencheurs.

3.2.4.3 Adaptabilité au contexte IDM

Nous avons vu que TEMPO permet de modéliser les relations entre *produits* de deux façons différentes et parfois complémentaires (pour le cas de la composition). En effet, une relation entre deux *produits* peut s'exprimer à travers un lien de type objet-attribut. Elle peut également être explicitement définie avec l'aide d'une relation TEMPO de nom O|R|D. Ainsi, dans le cas de *produits-modèles*, il est bien possible de représenter avec ce langage des liens de type *nest* en utilisant, par exemple, un lien objet-attribut complété, au besoin, par la définition d'une relation explicite (O|R|D) pour lui ajouter les détails de sa spécification. Le problème est qu'une fois une telle relation spécifiée, l'environnement ADELE est incapable (à l'exécution) de l'exploiter pour assurer une modularité dans la gestion des *produits-modèles*. En effet, ADELE ne peut représenter un *produit-modèle* qu'avec un fichier et n'est pas doté de services spécifiques lui permettant une gestion de ces fichiers en tant que *modèles*, c'est à dire avec la possibilité d'accès aux éléments de modèles, de réaliser des opérations de *modèles* (par exemple, ADELE ne dispose d'aucun moyen lui permettant de composer deux *modèles* pour créer le modèle-somme correspondant), etc.

Le fait que TEMPO ne permette de modéliser de relations qu'entre deux *produits* seulement (une source, et une cible) constitue une limite de taille pour cette approche quant au support de la modélisation des relations identifiées au chapitre 2 entre *produits-modèles*. En effet, il arrive souvent que ces dernières, comme c'est le cas de l'*overlap*, impliquent plus de deux *produits-modèles*. En revanche, le langage permet de modéliser une relation d'*overlap* entre deux *produits-modèles* avec une relation explicite (O|R|D). Toutefois, l'environnement d'exécution ne peut pas tirer profit de cette relation pour assurer une *cohérence relationnelle* entre les *produits-modèles* ou leur évolution dynamique, pour les mêmes raisons que celles évoquées dans le cas du *nest*.

3.2.5 L'approche de Di Nitto et al.

3.2.5.1 Présentation générale de l'approche

L'approche dite de Di Nitto et al. est présentée dans [Di Nitto et al. 2002]. Les auteurs y proposent de représenter un *modèle de procédé* à l'aide de diagrammes UML (version 1.3). La **Figure 3.4** montre un diagramme de classes UML qui représente les différentes entités considérées à la modélisation d'un *procédé* dans cette approche ainsi que les liens entre elles.

Dans cette approche, un *modèle de procédé* se compose de trois types de diagrammes UML 1.3 : un diagramme de classes, plusieurs diagrammes d'états-transitions et un diagramme d'activités.

Le diagramme de classes a pour objectif de représenter l'ensemble des éléments du *procédé* de même que les liens entre eux. Ces éléments sont représentés sous la forme de sous-classes des classes qui apparaissent sur le diagramme de la **Figure 3.4**. Un diagramme d'états-transitions est associé à chaque élément du *procédé*. Il permet de décrire les règles du *procédé* relatif à l'élément associé. Chaque élément du *procédé* est associé à un ensemble d'états. Un diagramme d'activité complète le *modèle de procédé*. Il montre les différents liens de précédence et de d'imbrication entre toutes les activités du *procédé* (représentées sous forme de classes de mêmes noms respectifs dans le diagramme de classes).

Une fois un *procédé* modélisé, il est ensuite présenté à un *moteur d'exécution* du nom d'ORCHESTRA Process Support System (OPSS) [Cugola et al. 2001; Cugola et al. 1998]. OPSS permet de coordonner les activités d'agents répartis intervenant dans l'exécution d'un *procédé*. Il gère l'évolution des éléments du *procédé* qui est guidée par des règles qui décrivent tous les changements d'états possibles et qui sont écrites sous la forme de machines à états associées à chaque entité du *procédé*. Dans ces machines à états, une transition est semblable à une règle ECA telle que définie dans ADELE [Estublier & Casallas 1994].

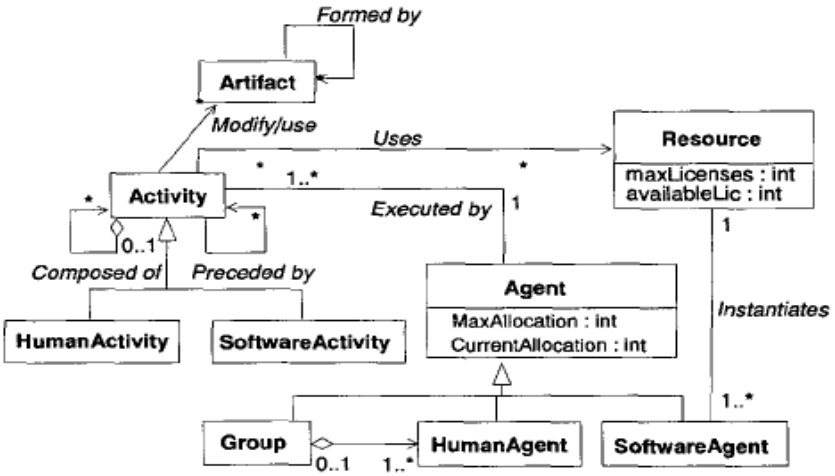


Figure 3.4: Les Entités d'un *Procédé* et leurs Relations sous OPSS

3.2.5.2 Les *produits* et les relations entre eux

La classe **Artifact** (**Figure 3.4**) modélise les *produits* dans cette approche. La seule relation entre *produits* qu'il est possible de modéliser ici est spécifié par le lien *Formed by.* Ce lien indique qu'un *produit* peut être composé d'autres *produits*. Comme indiqué plus haut, la spécification de chaque *produit* est complétée par une machine à états décrivant ses règles d'évolution. Par défaut, un *produit* est associé aux états *Created*, *OnEdit*, *Edited*, et *Distroyed* [Cugola et al. 2001].

Pour l'exécution d'un *procédé*, OPSS s'appuie sur un *modèle* formé de classes Java qui correspond au noyau sur lequel il se base pour représenter un *procédé* et son évolution. Un composant appelé *State Server* de son architecture se charge de stocker l'état courant du *procédé* à travers une représentation sous forme d'objets Java de l'état de toutes ses entités. C'est ainsi qu'avec l'aide des règles décrivant chaque *produit*, ce composant va s'occuper de gérer l'évolution (de l'état) de celui-ci. Le *State Server* ne prend en compte que la relation de composition entre *produits*. Toute autre relation modélisée entre *produit* doit alors être gérée de façon manuelle durant la phase de translation du *modèle de procédé* vers le modèle objet Java OPSS. La gestion effective des *produits* est donc laissée au soin des agents impliqués dans le *procédé*.

3.2.5.3 Adaptabilité au contexte IDM

Cette approche s'appuie sur UML 1.3 pour modéliser un *procédé* et sur la génération du code Java qui va guider l'exécution de celui-ci. Il se caractérise toutefois par une très faible

considération des relations entre les *produits*. La relation de composition qu'elle propose peut être utilisée pour modéliser un lien de type *nest* entre *produits-modèles*, sans possibilité de la spécifier complètement dans le cas où elle nécessite des caractéristiques supplémentaires.
L'approche ne permet de modéliser aucune autre relation entre *produits-modèles*. Elle ne permet pas, non plus, de prendre en compte des relations entre *produits-modèles* lors de l'exécution d'un *procédé* pour améliorer leur gestion.

3.2.6 L'approche dite de Chou de modélisation de procédé

3.2.6.1 Présentation générale de l'approche

Cette approche [Chou 2002] est très proche de l'approche de Di Nito et al. présentée précédemment. Elle préconise l'utilisation d'un langage de haut niveau pour décrire un *procédé*, et à partir de cette description de générer un *programme de procédé* [Estublier & Casallas 1994] correspondant. L'auteur propose UML (version 1.4) pour la représentation de haut niveau. En effet, dans cette approche, un *procédé* est modélisé à l'aide d'un *diagramme de P-activity* [Chou & Jason Chen 2000] et d'un *diagramme de P-class* [Chou & Jason Chen 2000]. Ces diagrammes sont utilisés pour les mêmes objectifs dans dans l'approche de Di Nitto et al.. Pour les construire, les notations UML utilisée pour la construction de diagrammes de classes et d'activités sont réadaptées puis utilisées (respectivement).
A côté de ce haut niveau, un bas niveau de représentation d'un *procédé* est proposé sous la forme d'un programme orienté objet. L'approche propose, ensuite des règles permettant de transformer une spécification de haut niveau en un programme de bas niveau exécutable.

3.2.6.2 Les *produits* et les relations entre eux

Les *produits* d'un *procédé* sont représentés au sein d'un *diagramme de P-class* par des classes et les relations entre elles par des relations entre ces classes. Toutes les classes modélisant des *produits* sont des sous-classes, directes ou indirectes, d'une classe prédéfinie **Document**. L'approche permet de modéliser des relations de composition, d'héritage, et de dépendance entre *produits*. La **Figure 3.6** montre un exemple de représentation de trois *produits* (*Design*, *Specification* et *Requirement*) ainsi que les relations de dépendance et de composition entre eux.

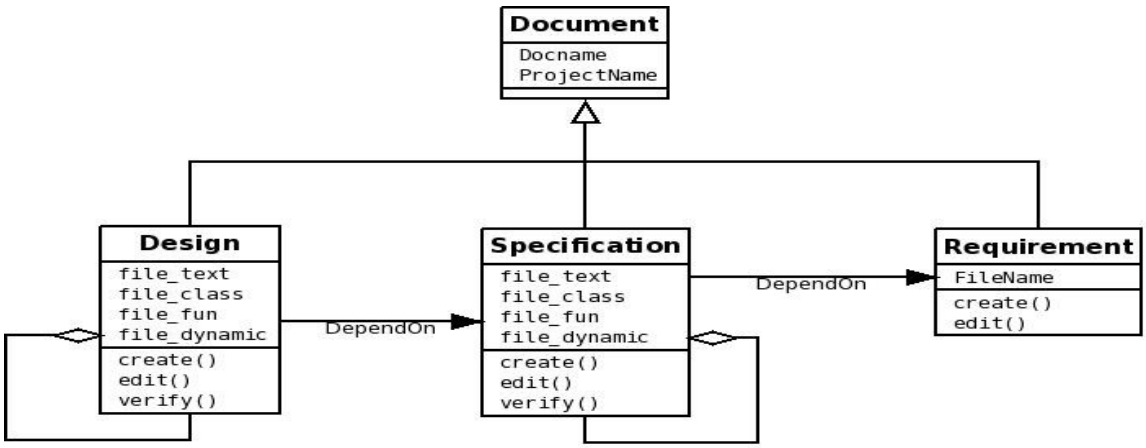


Figure 3.5: Un Exemple Modélisation de *Produits* avec les Relations entre eux selon Chou.

Le langage de bas niveau est structuré par une grammaire dont un extrait apparaît à la **Figure 3.6.a**. La **Figure 3.6.b** présente la définition des symboles utilisés dans cet extrait de la grammaire. Comme il apparaît dans cette grammaire, les *produits* sont représentés par des classes (**ProductClass**, ligne 4). Ces classes portent des attributs et opérations. Les attributs sont utilisés pour définir les caractéristiques du *produit*; par exemple, le nom du fichier contenant les données du *produit* est représenté par un attribut. Une classe (**PartOf**) permet de modéliser un lien de composition entre deux *produits*. Elle est une sous-classe d'une autre plus générique (**Relationship**, ligne 11) qui elle désigne n'importe quelle relation entre éléments d'un *procédé*. Il n'existe pas, dans ce langage, un concept pour représenter une autre relation entre *produits*. Les relations de dépendances entre *produits* restent implicites; il n'y a aucune possibilité de leur associer une caractéristique pouvant être exploitée à l'exécution. Par exemple, la syntaxe suivante est fournie par le langage pour exprimer, dans le corps d'une opération, l'assignation d'un traitement à un développeur.

```
developer_name develops product_name referring to reference_list
    with limits schedule_name, budget_name;
```

Cette instruction indique en même temps que *« pour construire le produit de nom **product_name**, le développeur **developer_name** a besoin du (des) produit(s) dont la (les) référence(s) est (sont) précisée(s) dans la liste **reference_list** »*. Elle exprime ainsi une relation de dépendance implicite du ***product_name*** aux *produits* représentés dans ***reference_list***.

(a)

```
1. ProcessProgram ::= {Class}  ProcessClass
     /* A process program is composed of one "Process" class and one or more other classes. */
2. Class ::= ProductClass | RoleClass
     /* Only product classes and role classes can be defined as customized classes. */
     /* Other classes such as tools, schedules, and budgets are built-in classes. */
3. ProcessClass ::= "class Process"  "{"  (Data)  StartTask  {Task}  (ExceptionBlock)  "}"
     /* "Task" defines a task, which is a collection of related activities */
     /* "StartTask" is the entry point of a process program. "ExceptionBlock" defines an exception handler. */
4. ProductClass ::= ClassDef  "{"  (Attribute)  Constructor  (Operation)  "}"
     /* A class is composed of attributes, a constructor, and operations. */
5. RoleClass ::= ClassDef  "{"  (Attribute)  Constructor  (Operation)  "}"
     /* An operation of a role class corresponds to an activity of a developer */
6. ClassDef ::= "class"  ClassName  ["extends"  ClassName]
     /* "extends" defines inheritance relationships*/
7. Constructor ::= ClassName  "("  (Parameter)  ")"  "{"  {Statement}  "}"
8. Operation ::= [DataType]  OperationName  "("  (Parameter)  ")"  "{"  {Statement}  "}"
9. StartTask ::= "start()"  "{"  {Statement}  "}"
     /* The entry point of a process program. */
10. Task ::= [DataType]  TaskName  "("  (Parameter)  ")"  "{"  {Statement}  "}"
11. Statement ::= Data | Relationship | ClassInstance| GeneralStatement| SyncStat
     /* "ClassInstance"  instantiates an instance from a class. */
     /* "Relationship" define a relationship among process components, such as products and tools */
12. GeneralStatement ::= ObjectOperationInvocation | WorkAssignment | Branch |Loop
     /* "SyncStat" is for synchronizing activities, including synchronous and asynchronous communication */
13. SyncStat ::= ConcurrencyBlock | EventStat
14. ExceptionBlock ::= "exception" ExceptionName "{" {Statement} "}"
```

(b)

Symbol	Meaning
::=	is defined as
|	alternative
[X]	zero or one instance of X
(X)	zero or more instance of X
{X}	one or more instance of X
/* . . . */	comments
un-quoted symbols	non-terminals
quoted symbols	terminals

Figure 3.6: Un Sous-Ensemble de la Grammaire BNF du Langage de Bas Niveau de l'Approche de Chou [S.-C. Chou 2002]

3.2.6.3 Adaptabilité au contexte IDM

L'approche de Chou fournit la classe **PartOf** qui permet de spécifier une relation de type *nest* entre *produits-modèles* de *procédés*. Par contre, elle ne supporte pas la définition complète d'une autre relation entre *produits-modèles*.

L'absence de support de la gestion de la modularité des *produits-modèles* à l'exécution de *procédé* est liée au non support par le langage de bas niveau d'opérations spécifiques aux *modèles* comme la composition de modèles.

3.2.7 UML4SPM

3.2.7.1 Présentation générale de l'approche

UML4SPM [Bendraou et al. 2005; Bendraou et al. 2006] est un langage qui s'appuie sur les techniques de méta-modélisation pour définir et exécuter les *modèles de procédés*. Il est défini sous la forme du *méta-modèle* présenté à **Figure 3.7.a**. UML4SPM utilise des concepts UML 2.0 qui sont étendus pour les besoins de la modélisation de *procédés*. Sa création fait suite à une évaluation du standard SPEM 1.1 [OMG-SPEM1 2002] et correspond ainsi à une réponse à une RFP [OMG-SPEM2-RFP 2005] de l'OMG visant l'amélioration du standard de modélisation de *procédés* en faisant usage des possibilités offertes par UML2.

Un *modèle de procédé* UML4SPM vient sous la forme d'un diagramme d'activités qui étend celui du standard UML 2.0 en lui rajoutant les concepts propres à la modélisation de *procédé*. Il définit l'ensemble de toutes les activités du *procédé* ainsi que les actions qui composent celles-ci [Bendraou et al. 2005]. Le diagramme d'activités peut être complété par d'autres diagrammes qui permettent de modéliser les détails des éléments du *procédé*. Ces derniers peuvent être des diagrammes de classes pour définir, entre autres, plus finement les liens entre les *produits*, ou des diagrammes d'états-transition afin de préciser les états possibles des composantes du *procédé* tels que les activités. UML4SPM utilise une notation graphique qui est également une extension de celle utilisée pour faire des diagrammes d'activités UML 2.0.

UML4SPM propose deux approches d'exécution de *procédés*. La première [Bendraou et al. 2007] préconise une utilisation du standard d'orchestration de Services Web WS-BPEL (Web Service Business Process Execution Language) [OASIS-WSBPEL 2007]. La seconde approche d'exécution de *procédé* préconisée dans UML4SPM est basée sur un modèle d'exécution présenté dans [Reda Bendraou et al. 2008].

3.2.7.2 Les *produits* et les relations entre eux

L'objectif de la **Figure 3.7** est de faire ressortir les concepts du langage relatifs aux *produits* et aux relations qui peuvent s'établir entre eux, avec l'aide du *méta-modèle*. Le concept utilisé pour représenter les *produits* est celui de **WorkProduct**. Un **WorkProduct** est défini ici comme étant n'importe quelle information consommée, produite ou modifiée durant le cycle de développement d'un système logiciel [Reda Bendraou et al. 2005]. Les propriétés de cette méta-classe permettent d'exprimer si un *produit* est un livrable ou non, sa date de création et

de dernière modification, sa version, et l'URI permettant de le localiser. Un *produit* peut être associé à une catégorie modélisée par la méta-classe **WorkProductKind**. Ainsi, il est possible d'exprimer le fait qu'un *produit* soit de type « Code », « Document », « Modèle », etc. La relation ***impacts*** est utilisée pour modéliser des liens d'impact entre *produits*, traduisant le fait qu'une modification sur un *produit* donné va impacter sur un autre. Du fait du lien ***nestedArtifacts*** hérité de la méta-classe **Artifact** de UML2 (voir **Figure 3.7.b**), le langage permet également d'exprimer des relations de composition entre *produits*. Dans le paragraphe qui suit, nous montrons comment tous ces éléments relatifs aux *produits* sont pris en compte à l'exécution d'un *procédé* UML4SPM.

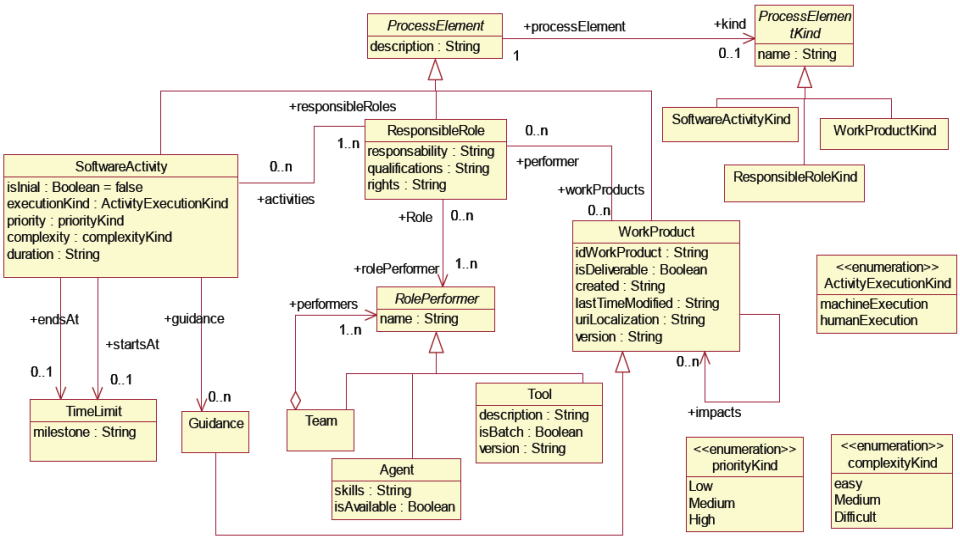
Selon la première approche d'exécution de UML4SPM, un *modèle de procédé* est transformé en une description de procédé correspondante en BPEL. La transformation s'appuie sur des règles définies dans [Bendraou et al. 2007] et selon lesquelles un **WorkProduct** est transformé en une variable BPEL, et son type en un attribut ***MessageType*** associé à la variable. Les attributs d'un **WorkProduct** sont représentés dans un fichier XML Schema. Le résultat de la transformation d'un *modèle de procédé* est alors passé à un moteur de procédé BPEL pour exécution. Cette approche ne définit aucun mécanisme permettant de prendre en compte les relations modélisées entre les **WorkProducts**. L'autre approche d'exécution de UML4SPM repose entièrement sur une idée inspirée d'une RFP de l'OMG parue en 2005 et qui décrit les exigences de base pour une exécutabilité de UML et qui devait conduire à la définition d'une machine virtuelle UML de base comme souligné dans [OMG-UML-Exe 2008]. Cette approche, comme la première, ne définit aucun mécanisme spécifique permettant la prise en charge des *produits* et de leurs relations modélisées.

3.2.7.3 Adaptabilité au contexte IDM

L'approche UML4SPM permet de modéliser un lien de type *nest* entre *produits-modèles* d'un *procédé*, mais ne donne pas la possibilité d'associer des caractéristiques supplémentaires à une telle relation même si celles-ci sont souvent nécessaires. Le lien d'***impacts*** qui est proposé peut être utilisé pour déclarer une relation de dépendance entre *produits-modèles*. Ce lien reste tout de même simplement descriptif dans la mesure où le langage ne supporte aucune possibilité de lui ajouter des éléments pour le caractériser au besoin. Or, une telle précision est nécessaire pour rendre automatique l'évolution des *produits-modèles* sur la base de la relation à l'exécution.

Au niveau de l'exécution de *procédé*, l'approche ne définit aucun mécanisme spécifique de gestion de *produits-modèles*.

a)



b)

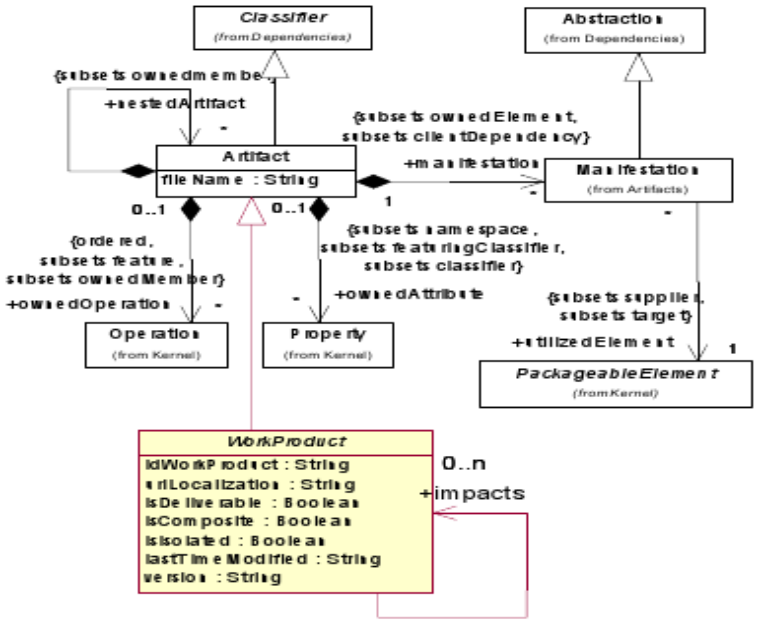


Figure 3.7: Deux Extraits [Reda Bendraou et al. 2006; Reda Bendraou et al. 2008] du *Méta-Modèle* UML4SPM

3.2.8 Le standard SPEM

SPEM (**S**oftware **P**rocess **E**ngineering **M**eta-Model) est le standard de modélisation de *procédés*. SPEM est proposé sous la forme d'un *méta-modèle* par l'OMG et est aujourd'hui disponible en deux versions nous nous sommes intéressés à la dernière appelée SPEM 2

[OMG-SPEM2 2008]. Le lecteur intéressé par SPEM 1 peut se référer à [OMG-SPEM1 2002]. Nous précisons que d'autres standards traitent également de la représentation de procédés, mais du fait qu'ils ne sont pas spécifiques aux *procédés* logiciels et pour des raisons de place nous avons préféré nous limiter à juste les lister ci-après.

- La notation BPMN (Business Process Model and Notation) [OMG-BPMN2 2011], une initiative de la BPMI (Business Process Management Initiative, un consortium d'entreprises rallié à l'OMG depuis juin 2005 pour former la Business Modeling & Integration (BMI) Domain Task Force (DTF)). Elle vise à définir une notation graphique commune permettant de modéliser les processus métier.
- BPEL4WS (Business Process Execution Language for Web Services) [OASIS-WSBPEL 2007] ou simplement BPEL (Business Process Execution Language) est également une initiative de la BPMI dont le but est de proposer une représentation XML des activités liées à l'exécution d'un processus métier. Là où la notation BPMN s'attache à décrire statiquement les processus, le langage BPEL décrit leur dynamique d'ensemble.
- XPDL (XML Process Definition Language) [WFMC 2008] est un standard proposé par la WfMC (Workflow Management Coalition). Son but et de représenter des modèles de procédé sous forme de fichiers XML.

3.2.8.1 Présentation générale de l'approche

SPEM 2.0 [OMG-SPEM2 2008] est la dernière version en date du standard, elle date de mars 2007 et est sortie sous la forme d'un *méta-modèle* conforme au MOF 2.0. Il comprend également un profil définissant des stéréotypes aux éléments de modèle de UML 2 Superstructure [OMG-UML2 2007] pour permettre une représentation des *modèles de procédé* de façon non ambigüe à l'aide diagrammes UML 2. SPEM est autant un *méta-modèle* de modélisation de processus qu'un cadre conceptuel qui met à disposition les concepts nécessaires pour modéliser, documenter, présenter, gérer, échanger, et (indirectement) exécuter des *procédés* et méthodes de développement logiciels. SPEM 2.0 définit une base minimale d'éléments nécessaires à la définition de *procédés*, sans aucune spécialisation à un quelconque domaine ou une quelconque discipline. SPEM 2.0 ne se veut pas être un langage générique de modélisation de *procédés*, mais à la place fournit un cadre à partir duquel tout *procédé* spécifique peut être bâti. Le *méta-modèle* est organisé en sept (7) packages qui

divisent le *modèle* en des unités logiques interdépendantes. Pour des raisons d'espace, ces packages ne seront pas décrits dans ce document. Le lecteur pourra se référer à la spécification du standard pour plus de détails [OMG-SPEM2 2008].

3.2.8.2 Les *produits* et les relations entre eux

Puisque SPEM 2.0 est représenté sous la forme d'un *méta-modèle*, nous décrivons dans cette section ses concepts relatifs aux *produits* et à leurs relations.

- **WorkProductDefinition** et **WorkProductDefinitionRelationship**: ils permettent de représenter des définitions générales et réutilisables de *produits* ainsi que des relations entre eux.
- **WorkProductUse** et **WorkProductUseRelationship**: ils permettent de représenter des *produits* spécifiques à une activité précise d'un *procédé* donné.

Malgré cette différenciation faite par le standard entre ces éléments, **WorkProductDefinitionRelationship** et **WorkProductUseRelationship** ont la même sémantique qui est d'exprimer une relation entre un *produit* (source) et un autre (cible). Les relations peuvent être de différents types exprimés à l'aide d'instances de **Kind**. Le concept générique de **Kind** s'applique à tout élément de procédé et remplace un ensemble de concepts de SPEM 1 dont celui de **WorkProductKind** qui lui est spécifique aux *produits*. Les relations considérées par le standard comme étant les plus utilisées sont celles listées ci-après.

- La **composition** qui traduit le fait que le *produit* cible soit une partie du *produit* source.
- L'**agrégation** qui ajoute à la définition précédente le fait que la cible puisse en même temps être partagée par deux ou plusieurs *produits* différents.
- La **dépendance** (ou **impactedBy**) qui exprime qu'une modification de la source entraine une invalidité de la cible tant que la modification n'est pas répercutée sur elle.

Il est important de mentionner l'attention qui est accordée aux relations entre *produits* dans le standard SPEM 2.0 qui décrit, à travers un exemple dans sa section 9.12 [OMG-SPEM2 2008], la possibilité d'une construction de *modèles de procédés* guidée par les *produits*. En effet, l'exemple montre qu'un *procédé* peut être modélisé en suivant les étapes suivantes.

- Commencer par identifier l'ensemble de tous les *produits* utilisés, produits et mis à jours.
- Ensuite, représenter, à l'aide d'un diagramme, l'ensemble des dépendances entre les

produits identifiés.

- Définir les différents états possibles pour chaque *produit*.
- Enfin, associer chaque transition entre deux états à une activité (avec les autres éléments de procédé impliqués comme les rôles, agents, etc).

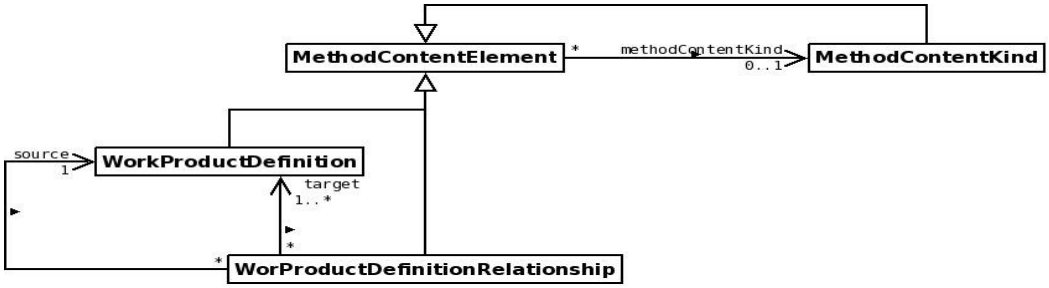


Figure 3.8: La Méta-Class WorkProductDefinitionRelationship de SPEM 2

3.2.8.3 Adaptabilité au contexte IDM

La **Figure 3.8** montre la méta-classe **WorkProductDefinitionRelationship** du *méta-modèle* SPEM 2. Elle montre également que selon le standard, toute relation entre *produits* est orientée (obligation d'indiquer une source et la (les) cible(s)).

A l'aide de SPEM, il est possible de définir une relation de type *nest*. Pour la spécifier, il faut définir, si cela est nécessaire, une instance indirecte de **MethodContentKind** qui correspond au type de lien souhaité (composition, agrégation, etc.).

Le fait que chaque relation soit orientée sous SPEM 2 pose un problème pour la définition d'une relation non orientée comme l'*overlap*. Il est tout de même possible de contourner ce problème. En effet, pour chaque lien à définir, on peut fixer n'importe laquelle des instances de **WorkProductDefinition** à lier comme la source de la relation, et alors toutes les autres comme des cibles. Comme dans le cas du *nest*, il faut également définir une instance indirecte de **MethodContentKind** à l'aide de laquelle ajouter les caractéristiques du lien.

SPEM ne propose pas d'approche d'exécution de *procédé*, nous ne l'évaluons pas de ce point de vue.

3.2.9 L'approche SPEM4MDE

3.2.9.1 Présentation générale de l'approche

SPEM4MDE [Diaw et al. 2010; Diaw et al. 2011] a fait l'objet d'une thèse [Diaw 2011] dans l'équipe MACAO de l'IRIT[1]. C'est une approche de modélisation et d'exécution de *procédés* dans un contexte IDM. Elle est conçue sous la forme d'un *méta-modèle* MOF permettant de décrire la structure de *procédés* IDM. Ce *méta-modèle* correspond à une extension du standard SPEM 2 (voir section 3.2.8) auquel sont ajoutés les concepts nécessaires au support des activités d'un *procédé* dans lequel les *produits* sont des *produits-modèles*. Les principaux concepts ajoutés sont relatifs aux transformation appliquées aux *produits-modèles* lors de l'exécution de procédés, de même qu'aux relations entre les *produits-modèles* et leurs *méta-modèles*. De plus, l'approche offre la possibilité d'exprimer la sémantique comportemental d'un *procédé* et définit sur sa base une approche d'exécution mise en oeuvre à travers un environnement appelé SPEM4MDE-PSEE qui supporte l'exécution de *modèles de procédés* définis avec le langage. SPEM4MDE utilise des machines à états UML2 pour la spécification de l'aspect comportemental des *procédés*. Elle s'appuie également sur la standard MOF QVT [OMG-QVT 2008] pour associer une sémantique exécutable aux transformations modélisées.

3.2.9.2 Les *produits* et les relations entre eux

Du fait que SPEM4MDE s'appuie sur le standard SPEM 2 pour définir la structure des *procédés*, elle supporte tous les concepts présentés à la section 3.2.8.2. De plus, le *méta-modèle* SPEM4MDE définit le concept de **Model** pour représenter les *produits-modèles* de *procédé* de même que les *méta-modèles* associés. Du fait que cette méta-classe est une spécialisation de **WorkProductUse** de SPEM2.0::ProcessStructure, SPEM4MDE supporte les même possibilités que le standard SPEM quant à la modélisation des relations entre *produits-modèles*. De plus, SPEM4MDE définit les méta-classes **TransformationDefinition** et **TransformationImpl** pour respectivement représenter des définitions et implémentations de relations de transformation entre *produits-modèles*.

3.2.9.3 Adaptabilité au contexte IDM

En ce qui concerne le support de la modélisation des relations de types *nest* et *overlap* entre *produits-modèles*, SPEM4MDE s'aligne avec le standard SPEM 2.0. Il faut noter qu'en plus de

[1] Voir www.irit.fr

cela, SPEM4MDE supporte la modélisation avec les précisions requises des relations entre *produits-modèles* sources et cibles telles que les cibles soient obtenues des sources par transformations de *modèles*. Ce sont ces dernières relations qui sont exploitées et gérées au cours de l'exécution d'un *procédé* dans SPEM4MDE-PSEE. L'approche d'exécution de *procédés* ne permet tout de même pas de gérer des relations de types *nest* et *overlap* pour assurer une gestion optimisée des *produits-modèles*.

3.2.10 Synthèse sur les approches de modélisation et d'exécution de procédé

A la lumière de l'étude précédente, nous tirons les éléments de synthèse qui suivent. Il a été reconnu que la première génération de langages, dont la plupart de ceux présentés ci-avant, n'a remarquablement pas connu un succès dans le monde industriel [Reda Bendraou et al. 2010; Di Nitto et al. 2002]. Elles sont caractérisées par le fait que leurs politiques respectives de gestion de données sont plus adaptées pour des *produits* manipulés essentiellement durant les dernières phases d'un *procédé* comme des descriptions ou programmes textuels. En effet les modèles de données sur lesquels ces approches se basent à l'exécution d'un *procédé* prennent bien en compte la structure interne de ces types de *produits* de même que les principales relations entre eux comme la composition et la dépendance. Ce sont donc des approches qui sont conçues et bien adaptées à ces types de *produits*. En revanche, aucune d'entre elles ne s'adapte aux *produits-modèles*. En effet, ceux-ci sont vus comme de simples fichiers et ces approches ne définissent aucun mécanisme de traitement spécifique qui permette l'accès à leur structure interne.
Les approches plus récentes quant à elles s'appuient sur UML et mettent plus l'accent sur l'expressivité que sur l'exécutabilité des *procédés*. Elles sont caractérisées par le fait de sous-considérer les aspects des *procédés* relatifs aux *produits*, et surtout par le fait que, pour l'essentiel, elles ne proposent pas de véritables environnements d'exécution. La gestion des *produits*, dans ces approches, est plus sous la responsabilité des développeurs.

Nous avons représenté une synthèse de cette étude à travers le **Tableau 3.1** qui permet de voir qu'aucune des approches étudiées ne supporte à la fois la modélisation des relations entre *produits-modèles* et la prise en compte de ces relations à l'exécution. Dans ce tableau,

- un *Oui* indique le support de la propriété par l'approche considérée,

- un ***Non*** indique le non support, de la propriété par l'approche considérée,
- et un *Oui** indique que l'approche considérée ne supporte que partiellement la propriété étudiée. Par exemple, nous avons conclu que l'approche MSL/MARVEL supporte une gestion modulaire de *produits-modèles*. En effet cette approche voit les *produits-modèles* comme des fichiers et n'a aucun mécanisme lui permettant d'accéder à leur structure interne. Elle ne peut donc pas réaliser des opérations comme la composition de deux *modèles*. Toutefois, lorsqu'une relation *nest* est spécifiée au sein d'un *modèle de procédé*, l'approche permet de gérer les *produits-modèles* impliqués par la dite relation séparément les uns des autres.

APPROCHES	**MODELISATION**	**EXECUTION**			
	Relations	**Cohérence Relationnelle**	**Evolution Automatique**	**Evolution Dynamique**	**Modularité**
APPL/A	*Dérivation* *Nest*	**Non**	Oui	**Non**	Oui *
MSL/MARVEL	*Dérivation* *Nest* *Overlap*	**Non**	Oui	**Non**	Oui *
SLANG/SPADE	*Nest*	**Non**	Oui	**Non**	Oui *
TEMPO/ADELE	*Dérivations* *Nest* *Overlap* (**seulement entre 2 *produits***) ...	**Non**	Oui	**Non**	Oui *
Di Nito et al.	**Aucune**	**Non**	**Non**	**Non**	**Non**
Chou	*Nest*	**Non**	**Non**	**Non**	**Non**
UML4SPM	**Aucune**	**Non**	**Non**	**Non**	**Non**
SPEM2	*Nest* *Dérivation* *Overlap* etc.	**Non**	**Non**	**Non**	**Non**
SPEM4MDE	*Nest* *Dérivation* *Overlap* etc.	**Non**	**Non**	**Non**	**Non**

Tableau 3.1 : Synthèse de l'Etude sur les Approches de Modélisation et d'Exécution de *Procédés*

3.3 La gestion de l'évolution de *produits-modèles*

A côté des approches de modélisation et d'exécution de *procédés*, nous avons estimé que d'autres domaines sont également intéressés par les relations entre *produits*, comme par la prise en compte de ces relations au cours de l'évolution des *produits*. C'est ainsi que nous nous sommes intéressés aux *VCS*/*CMSs*. Nous présentons dans les sous-sections qui suivent quelques unes des approches qui existent dans ce domaine. Les approches présentées ont été choisies d'une part sur la base de leur architecture centralisée. D'autre part, certaines d'entres elles sont classiques et orientés *produits* (Subversion et Adele), et les autres plus récentes sont spécifiques aux *produits-modèles* (Modelbus et Odyssey-VCS). Ces approches sont donc représentatives de l'existant qui nous intéresse dans notre étude. Notre but est le même que dans la section précédente et consiste à mettre en exergue les relations prises en compte ainsi que l'adaptabilité ou non de ces approches aux *produits-modèles*.

Pour réaliser son rôle, un *VCS* s'appuie sur un modèle de données et un modèle de version. Le premier définit la structure des données gérées ainsi que les relations entre elles. Le second décrit la manière suivant laquelle les versions successives des *produits* gérés sont construites et stockées de même que les informations nécessaires à ces versions. Pour chacune des approches auxquelles nous nous intéresserons dans cette section, nous présenterons l'orientation de base, les relations supportées par le modèle de donnée et leur prise en compte durant l'évolution des *produits*, et enfin son adaptabilité au contexte IDM.

3.3.1 Subversion

3.3.1.1 Présentation générale de l'approche

Subversion* (encore appelé SVN) [Collins-Sussman et al. 2008] est un *VCS* libre. Il correspond à une amélioration du système CVS [Cederqvist & Pesch 1993]. Il gère des données sous la forme de fichiers et de répertoires, ainsi que les changements qu'on leur applique au fil du temps. Il offre la possibilité de revenir à d'anciennes versions d'un *produit* ou d'examiner la façon dont il a évolué. Subversion est un système généraliste qui peut être utilisé pour gérer n'importe quel ensemble de fichiers qui peuvent alors représenter du code source, des images ou vidéos, etc.

L'architecture de Subversion montre qu'il est composé d'un dépôt (repository), un ensemble de

*Voir aussi http://subversion.tigris.org

services d'accès au repository et d'une interface ouverte à des clients qui peuvent être dispersés à travers le réseau. Le repository a pour rôle d'assurer un stockage centralisé des données. Subversion utilise un mécanisme *copy-modify-merge* [Collins-Sussman, Fitzpatrick & Pilato 2008a; Shen & Sun 2002] comme mécanisme de gestion des accès concurrents aux données du repository.

3.3.1.2 Les *produits* et les relations entre eux

Le modèle de donnée utilisé par Subversion est simple. En effet, les *produits* sont organisés sous la forme d'une arborescence de fichiers, c'est-à-dire une hiérarchie classique de fichiers et de répertoires. Hormis la relation de composition qui existe entre un répertoire et les fichiers qu'il contient, aucune autre n'est définie dans le modèle de donnée de Subversion.

3.3.1.3 Adaptabilité au contexte IDM

L'OMG recommande l'utilisation du standard XMI [OMG-XMI 2005] pour l'encodage de *modèles* sous la forme de fichiers textes. Le standard XMI lui-même n'inclut pas de spécification relative aux relations entre différents *modèles*. Tout de même, différents mécanismes utilisant des URIs HTPP ou FTP, et des spécifications telles que Xpath, Xquery, etc. sont utilisés au sein des différentes implémentations du standard pour désigner des éléments précis d'un document XMI à partir d'un autre. Il existe donc le moyen d'encoder des relations entre *produits-modèles* sous la forme de fichiers XMI séparés ou non de ceux utilisés pour encoder les *produits-modèles* en question; xlinkit [Nentwich et al. 2002] en est un exemple.

Toutefois, aucun mécanisme n'est défini au sein de Subversion pour exploiter de tels liens entre fichiers. De plus, il ne peut accéder à aucun élément de modèle au sein d'un fichier XMI représentant un *produit-modèle*. Pour ces raisons, le modèle de donnée de Subversion ne supporte aucune relation entre *produits-modèles*. Par conséquent, Subversion ne supporte ni une gestion de la modularité de *produits-modèles*, ni leur évolution dynamique, encore moins une gestion d'une *cohérence relationnelle* entre eux.

3.3.2 Adele

ADELE [Estublier & Casallas 1994] a été initialement conçu en tant que un *CMS*. Il a, par la suite, servi de socle à l'approche TEMPO/ADELE de modélisation et d'exécution de

procédés que nous avons présentée à la section 3.2.4 de ce chapitre. Son modèle de donnée ainsi que son évaluation par rapport au contexte IDM ont été présentés dans cette même section.

3.3.3 ModelBus

3.3.3.1 Présentation générale de l'approche

ModelBus [Prawee Sriplakich, Xavier Blanc, et al. 2006; Prawee Sriplakich et al. 2008; P Sriplakich, X Blanc, et al. 2006] est un environnement collaboratif et réparti de gestion de *produits-modèles*. Il a fait l'objet d'une thèse [SRIPLAKICH 2007] au sein de l'équipe Move du LIP6[*]. ModelBus permet à un ensemble de développeurs travaillant dans différents endroits de manipuler des *produits-modèles* communs de manière concurrente. C'est également un environnement qui permet l'intégration d'outils hétérogènes du point de vue des données et du contrôle.

Pour gérer des *produits-modèles*, ModelBus se base sur le paradigme *copy-modify-merge* qu'il applique sur une architecture de type workspace-repository [Altmanninger et al. 2009]. C'est en gros une adaptation des *VCS* traditionnels (typiquement CVS) aux *produits-modèles*. Sous ModelBus, les *produits-modèles* sont encodés au format XMI.

Le repository de ModelBus assure l'évolution des *produits-modèles* qu'il gère en tenant compte des actions des développeurs sur eux. En effet, un ensemble d'opérations est mis à la disposition des développeurs à travers leurs espaces de travail respectifs.

3.3.3.2 Les *produits* et les relations entre eux

L'approche ModelBus représente les *produits-modèles* dans le repository central sous forme d'ensembles de *fichiers de modèle*. Ceux-ci sont des fichiers XMI. Ils sont organisés de façon arborescente, exactement comme sous Subversion. En plus du lien de composition de type répertoire-fichier, le modèle de donnée de ModelBus introduit une autre relation entre les fichiers. Il est possible, en effet, de faire référence à un élément de modèle se trouvant dans un fichier donné à partir d'un autre *fichier de modèle*.

3.3.3.3 Adaptabilité au contexte IDM

ModelBus gère le maintien de la cohérence des *fichiers de modèle,* même s'ils sont liés entre

[*] LIP6 – Laboratoire d'Informatique de Paris 6 - www.lip6.fr

eux, par le biais d'un mécanisme de résolution automatique de conflits basé sur des règles de cohérence. En outre, ModelBus offre un accès aux éléments de modèles contenus dans un *fichier de modèle*. Ces deux éléments combinés offrent la possibilité de représenter, sous la forme d'un fichier XMI séparé des autres, une relation de type *nest* ou *overlap* entre *produits-modèles* avec toutes leurs caractéristiques. Par contre, ModelBus utilise une approche de stockage hiérarchique des *fichiers de modèle* semblable au mécanisme fichier-répartoire employé dans CVS, mais il n'indique pas comment les *fichiers de modèle* sont créés, ni la base sur laquelle leur organisation hiérarchique est construite. Cette situation impose une construction manuelle (ou par un moyen externe) des *fichiers de modèle*, des fichiers qui représentent les relations, de même que de la hiérarchie. Il est évident que dans le cas de grands projets avec des *produits-modèles* de taille importante, cela devient facilement irréaliste.

En définitive, nous retenons que du fait de l'absence d'un mécanisme qui permette la construction des *fichiers de modèle* et de ceux représentant les relations, ModelBus ne peut intégrer aucun *procédé*. Même si son modèle de donnée supporte la représentation de relations comme *nest* et *overlap*, l'environnement ne peut pas se servir de leurs spécifications respectives contenues par un *modèle de procédé* pour assurer une amélioration de la gestion des *produits-modèles* du *procédé* correspondant au cours de son exécution.

3.3.4 Odyssey-VCS

3.3.4.1 Présentation générale de l'approche

Odyssey-VCS [Murta et al. 2007; H. Oliveira et al. 2005] est un système de contrôle de version développé en Java. Il a été développé ex nihilo pour lui éviter toute dépendance vis à vis des systèmes qui l'ont précédé et qui, selon ses auteurs, ne permettent de judicieusement gérer que les versions de *produits* de bas niveau comme le code. Une des caractéristiques principales d'Odyssey-VCS est le fait de rendre possible la définition d'*unités de version* avec une granularité fine et, surtout, configurable selon les projets. C'est ainsi qu'il se spécialise dans la gestion des versions de *produits-modèles* de plus haut niveau comme ceux d'analyse et de conception. Odyssey-VCS est un *CMS* pour des éléments de modèles UML. Son architecture qui est composée de trois couches principales (*client*, *transport*, et *serveur*) définit un fonctionnement de style workspace-repository. La couche *client* (workspace)

correspond aux outils de modélisation des développeurs. Ces outils utilisent une notation UML et encodent les *produits-modèles* en XMI. La couche *transport* utilise des Web Services comme protocole de transport. Enfin la couche *serveur* (repository) stocke et traite le XMI correspondant aux *produits-modèles*.

3.3.4.2 Les *produits* et les relations entre eux

Odyssey-VCS ne supporte que la gestion de *produits-modèles*. Il les représente sous la forme de fichiers XMI. Il repose sur un modèle de donnée qui ne reconnaît que le concept d'élément de modèle. Ce modèle de donnée définit également une relation de composition entre éléments de modèle qui tire son origine du *méta-modèle* UML. C'est cette relation qui par exemple, permet de décrire qu'une classe UML se compose de méthodes et d'attributs. L'approche supporte, d'un projet à un autre, la redéfinition des *unités de version* et des *unités de comparaison*. A cet effet, elle préconise l'utilisation d'un fichier XML appelé *descripteur de comportement*. C'est sur ce fichier que se base le système pour savoir si un éléments de modèle est ou non une *unité de version* ou une *unité de comparaison*. Le **Tableau 3.1** montre un exemple de configuration relative aux éléments de modèles UML de type **Model**, **Package**, **Class**, **Attribute**, **Operation**, **Use Case**, et **Actor**. Cette configuration met aussi en exergue les relations de composition considérées entre:

- **Model** et **Package**,
- **Package** et **Class** et **Use Case**, et enfin
- **Class** et **Attribute** et **Operation**.

En fonction de la configuration choisie qui dépend du projet, et suivant un algorithme de gestion de la cohérence défini par l'approche, les informations de version sont déterminées et stockées pour les éléments du *modèle*. Avec l'aide des relations de composition entre les différents éléments de modèles sous contrôle, une évolution de version se produit pour chaque élément dont un composant a changé de version.

Nous soulignons, pour finir, qu'à côté de la composition précédemment décrite, le modèle de donnée utilisé dans cette approche ne supporte aucune autre relation.

Eléments	Unité de Version (UV)	Unité de Comparaison (UC)
Model	True	False
Package	True	False
Class	True	True
Attribute	True	True
Operation	True	True
Use case	True	True
Actor	False	True

Tableau 3.2 : Un Exemple de Configuration Odyssey-VCS

3.3.4.3 Adaptabilité au contexte IDM

Nous avons souligné dans la sous-section précédente que cette approche ne supportait que la relation de composition entre éléments de modèles. Elle ne supporte donc pas l'expression de la relation *overlap*. Elle ne supporte pas, non plus, l'expression d'une relation de type *nest*. En effet, malgré la possibilité de représenter tous les éléments d'un *produits-modèles* comme des composants d'un seul élément UML de type **Model** qui peut contenir d'autres éléments de même type que lui, aucun moyen ne permet d'associer, au besoin, des caractéristiques à une composition.

En conséquence, l'approche ne supporte pas une gestion améliorée des *produits-modèles* d'un *procédé* en exécution sur la base des critères que nous avons considérés.

3.3.5 Synthèse sur l'étude sur les VCSs/CMSs

Le choix des approches que nous venons de présenter se justifie principalement par les deux éléments suivants.

- D'une part, Subversion est le plus connu des *VCSs* de type centralisé actuels [94]. De plus, les autres approches de sa famille reposent sur le même modèle de donnée que lui.
- D'autre part, après un parcours de la littérature, ModelBus et Odyssey-VCS constituent les seules approches de gestion d'évolution de modèles que nous avons pu recensées.

De l'étude présentée ci-haut nous retenons, comme dans le cas des approches de modélisation et d'exécution de procédés, que les *VCSs* se subdivisent eux aussi en deux grandes familles. La première famille est plus ancienne et contient des systèmes, comme SVN, dont le modèle

de donnée n'est pas adapté au contexte des *produits-modèles*. La deuxième famille quant à elle se compose de systèmes conçus pour gérer des *produits-modèles*. Ces systèmes se caractérisent malheureusement par une incapacité d'intégrer les *procédés* ou par une inadaptation de leurs modèles de donnée à la gestion des relations entre *produits-modèles*. Le **Tableau 3.3** représente une synthèse de cette étude. Ce tableau se lit d'une même façon que le **Tableau 3.1**.

APPROCHES	MODELISATION	EXECUTION			
	Relations	Cohérence Relationnelle	Evolution Automatique	Evolution Dynamique	Modularité
SUBVERSION	**Aucune**	**Non**	**Non**	**Non**	**Non**
ADELE	*Dérivations* *Nest* *Overlap* (**seulement entre 2 *produits***)	**Non**	Oui	**Non**	Oui *
MODELBUS	*Nest* *Overlap*	**Non**	**Non**	**Non**	**Non**
ODYSSEY-VCS	**Aucune**	**Non**	**Non**	**Non**	**Non**

Tableau 3.3: Synthèse de l'Etude sur les VCSs/SCMs

3.4 Conclusion

Dans ce chapitre nous avons, dans un premier temps, réalisé une étude des approches de modélisation et d'exécution de *procédés*. Elles se classent en deux générations. Du point de vue des *produits*, la première génération était composée d'approches qui s'adaptaient à leur contexte. En effet, elles supportaient la modélisation des *produits* ainsi que les relations qui existaient entre eux et dont les plus importantes sont la composition et la dépendance. Aussi, elles supportaient une prise en compte des ces relations au cours de l'exécution des *procédés*. Cependant, ces approches ne sont pas adaptées aux *produits-modèles* du contexte IDM actuel. Elles ne supportent, en effet, pas la spécification complète des nouveaux types de relations (voir chapitre 2) entre ces *produits-modèles*. De plus, elles manquent de mécanismes spécifiques à la gestion des *produits-modèles* à l'exécution des *procédé*. La seconde

génération quant à elle comprend des approches qui ne supportent ni la spécification complète des relations entre *produits-modèles*, ni leur prise en compte lors de l'exécution des *procédés*. Il n'existe donc pas dans la littérature parcourue une approche de modélisation et d'exécution de procédé qui prenne en compte de façon satisfaisante les relations entre *produits-modèles* des *procédés* du contexte IDM comme l'illustre le **Tableau 3.2** qui représente une synthèse de l'étude présentée dans ce chapitre portant sur la modélisation et l'exécution de *procédés*.

Nous avons également réalisé, dans un deuxième temps, une étude d'un autre domaine qui lui aussi s'intéresse la gestion de l'évolution de *produits*. C'est ainsi que nous nous sommes intéressés aux VCSs/CMSs. La majeure partie des *VCSs*/*CMSs* traditionnels ne prennent pas en compte des relations qui mettent en jeu la structure internes des *produits* qu'ils gèrent du fait qu'ils considèrent le fichier comme unités de version et de comparaison (*UVs* et *UCs*). D'autres parmi ces systèmes utilisent des *UCs* de plus fine granularité (ligne, paragraphes, mots, etc.) mais qui ne sont pas adaptées aux *produits-modèles*. Enfin, certains des *VCS*/*CMSs* orientés *modèles* commencent à voir le jour, certains d'entre eux supportent certaines relations mais se caractérisent malheureusement par une incapacité d'intégrer les *procédés*. Le **Tableau 3.3** représente une synthèse de l'étude présentée dans ce chapitre portant sur les *VCSs*/*CMSs*.

Il faut noter qu'à l'issue de cette étude, notre attention a particulièrement été marquée par certaines approches de modélisation et d'exécution de *procédés* comme ADELE, Di Nito et al. et Chou, et UML4SPM. Nous donnons les raisons de cet attrait ci-après.

Concernant ADELE.

Nous retenons l'importance des éléments suivants de cette approche.

- **L'intégration des *procédés* à la gestion de la configuration.** Dans [Estublier & Casallas 1994], les auteurs de ADELE soutiennent qu'*« une configuration n'est valide seulement que si tous les tests nécessaires sont exécutés avec succès et sont concluants »*. Ils précisent également que satisfaire une telle contrainte suppose tout de même une connaissance très précise des activités à mener, leurs lieux d'exécution, ainsi que les différentes collaborations entre agents pouvant conduire à leur réalisation. Cette intégration se justifie donc bien, mais en plus elle avantage le support de la prise en compte de relations entre *produits(-modèles)*.
- **La représentation des relations entre *produits* suivant la même forme que les *produits* liés.** Cela donne, en effet, beaucoup de possibilités quant à une spécification

exhaustive des relations.

- **Les mécanismes (synchronisation, alertes, etc.) utilisés pour mettre en oeuvre le concept de rôle (ou vue)**. Ils assurent, à l'exécution d'un *procédé*, un suivi partagé de l'évolution des *produits* qui représentent des vues différentes d'une même chose. Nous estimons que de tels mécanismes sont intéressants s'ils sont adaptés au contexte des *produits-modèles*, en l'occurrence dans le suivi de l'évolution d'éléments de modèles partagés.

Concernant Di Nito et al. et Chou.

Ces approches s'appuient sur une philosophie intéressante qui repose sur une représentation d'un procédé sur deux niveaux. Le premier est un niveau d'abstraction élevé et est réservé à la modélisation des procédés. Le deuxième niveau définit quant à lui une représentation du procédé plus adaptée à son exécution. La jonction entre les deux niveaux est assurée par une transformation basée sur un mapping entre les différents concepts.

Concernant UML4SPM.

Nous retenons l'importance des deux éléments suivants après l'étude de cette approche.

- **L'utilisation d'un *méta-modèle* pour décrire le langage**. Cela donne la possibilité de définir un *modèle de procédé* sous la forme d'un *modèle* instance de ce méta-modèle et, ainsi, de profiter des atouts de la métamodélisation pour l'exécuter.
- **Le typage des *produits***. Dans le cas des *produits-modèles*, le type est très important et permet d'aider à la spécification des relations. Dans UML4SPM, le typage est tout de même juste descriptif dans la mesure où il permet juste de dire qu'un *produit* est un *modèle*, ou un texte, etc. Dans le cas où on travaille avec des *produits-modèles*, le type peut permettre, par exemple, de connaître les méta-classes des éléments de modèles de chacun d'eux et de s'en servir pour spécifier les relations qui l'impliquent.

En conséquence, la philosophie qui sous-tend chacun de ces éléments de ces différentes approches a fortement guidé la solution que nous avons proposée et qui est présentée au chapitre suivant.

Chapitre 4

Prise en Compte des Relations entre *Produits-Modèles* à la Modélisation et à l'Exécution des *Procédés*

4.1 Introduction

Dans le chapitre 2, nous présentions la problématique de la thèse ainsi que ses principaux objectifs. Pour rappel, cette problématique se résumait au fait que les *LMPs* existants ne prennent pas en compte les relations entre les *produits-modèles* lors de la modélisation de *procédés*, de même que les *moteurs d'exécution* au moment de l'exécution de ces *procédés* que nous considérons dans le contexte IDM. Cette situation, nous l'avons déjà soulignée dans les chapitres précédents, est liée à l'absence d'une spécification précise de ces relations et à un non support automatisé de l'évolution de ces *produits-modèles* par les *moteurs d'exécution* de *procédés* existants. Cette problématique a été illustrée et précisée au chapitre 3 à travers un état de l'art dans lequel nous avons présenté plusieurs approches de modélisation de *procédés* et de gestion de *produits* en général et de *produits-modèles* en particulier ainsi que leurs supports respectifs des relations entre *produits* (ou *produits-modèles*). Dans ce chapitre, nous allons présenter notre proposition de solution à cette problématique. Elle consiste à permettre la modélisation des relations existantes entre *produits-modèles* afin d'optimiser leur gestion. L'optimisation en question repose sur deux piliers principaux:

- d'abord sur une analyse, par les *moteurs d'exécution de procédés*, des relations existantes entre *produits-modèles* afin de faciliter l'évolution et le maintien de la cohérence entre ces derniers par des mécanismes de propagation automatique (ou semi-automatique) des modifications d'un *produit-modèle*.
- et ensuite sur une plus grande flexibilité dans la *granularité* des *produit-modèles* qui assure une plus grande modularité dans leur gestion. En effet, comme nous l'avons déjà montré au chapitre 2, il peut arriver que des parties de *produits-modèles* puissent être vues comme des entités séparées, liées éventuellement entre elles, mais individuellement manipulables par le *moteur d'exécution*.

Notre contribution se subdivise en deux principales parties.

La première est relative à la modélisation des *procédés* et correspond à la modélisation des *produits-modèles* utilisés durant le *procédé* de développement ainsi que des relations entre eux. Dans le but de rendre cette modélisation possible, nous avons proposé un *méta-modèle* prenant en compte les relations entre les *produits-modèles* ainsi que la sémantique associée. Celui-ci est nommé *méta-modèle* de *WorkProducts*.

La deuxième partie de notre solution est quant à elle relative à l'exécution des *procédés* et correspond à une gestion améliorée et (semi) automatique des *produits-modèles* durant cette exécution en s'appuyant sur les relations spécifiées à la phase de modélisation. Par cette gestion améliorée nous entendons une évolution (semi) automatique, une gestion de la *cohérence relationnelle*, ainsi qu'une modularité des *produits-modèles*. Afin de structurer les différentes entités concrètes correspondant aux *produits-modèles* tels qu'ils sont gérés par le *moteur d'exécution* au cours de l'exécution du *procédé*, nous avons également proposé un *méta-modèle*. Ce dernier a pour but de structurer les entités à travers lesquelles un *moteur d'exécution* d'un *procédé* va gérer ses *produits-modèles* une fois ceux-ci modélisés par le *méta-modèle* de *WorkProducts*.

Dans ce chapitre de la thèse, nous présentons d'abord une synthèse décrivant succinctement la solution que nous avons proposée afin d'en donner une vue globale au lecteur. Les détails de la solution sont ensuite fournis à travers la présentation des deux *méta-modèles* explicités à travers un exemple présenté à l'effet. Nous présenterons également les correspondances établies entre les concepts de ces deux *méta-modèles*. Ces parties seront suivies d'une

conclusion du chapitre.

4.2 Synthèse de la solution proposée

Comme énoncée plus haut en introduction de ce chapitre, notre solution permet de prendre en compte les relations entre les futurs *produits-modèles* d'un *procédé* dès la modélisation de celui-ci. Elle offre, ensuite, la possibilité de prendre en compte ces relations pour améliorer la gestion des *produits-modèles* à l'exécution du *procédé*. Avant d'en donner une description détaillée dans les sections suivantes de ce chapitre, dans le reste de cette section, nous décrivons succinctement cette solution d'abord du point de vue de la modélisation et, ensuite, du point de vue de l'exécution.

Du point de vue de la modélisation, nous proposons un *méta-modèle* permettant de structurer les *WorkProducts* utilisés lors de la description d'un *procédé*. Ce *méta-modèle* contient donc les concepts nécessaires à la définition des *WorkProducts* d'un *procédé* i.e. des concepts permettant de définir des relations entre ces différents *WorkProducts* et d'associer à ces relations la sémantique nécessaire qui permettra leur exploitation par un moteur d'exécution de procédé. Ce *méta-modèle* supporte pour le moment les relations de **Nest** et d'**Overlap** qui respectivement permettent de spécifier les relations d'inclusion et de partage d'éléments entre deux ou plusieurs *produits-modèles* d'un *procédé* et que nous présenterons plus amplement dans la suite avec le *méta-modèle*. En effet, ces deux relations, et nous allons le montrer tout au long de ce chapitre, offrent une plus-value significative à la gestion des *produits-modèles* au cours de l'exécution d'un *procédé*. Par exemple, le **Nest** permet d'assurer une meilleure gestion de la modularité en offrant la possibilité d'exprimer qu'un *produit-modèle* peut être composé ou fait partie d'un ou de plusieurs autres *produit(s)-modèle(s)*. Il contribue ainsi à une meilleure réutilisation, un stockage unique et sans redondance, et une granularité de verrouillage plus fine des *produits-modèles*. L'**Overlap** quant à elle permet d'assurer une évolution automatique et dynamique des *produits-modèles* durant l'exécution du *procédé*, une gestion de la *cohérence relationnelle* entre eux et l'évitement de la redondance d'information à travers un stockage unique des éléments de modèles dupliqués. Nous reviendrons plus en détails sur tous ces aspects dans la suite du chapitre.

Une fois les *produits-modèles* d'un *procédé* spécifiés lors de la phase de modélisation à travers

les concepts du *méta-modèle* de *WorkProducts*, ceux-ci doivent être gérés à l'exécution en tenant compte des détails contenus dans leur spécification. Or la description faite au niveau de la modélisation sous la forme de *WorkProducts* contient exclusivement des éléments caractéristiques des *produits-modèles* comme leur structure interne et les éventuelles relations entre eux ainsi que des détails sur ces relations. Cette modélisation est donc faite à un niveau que nous qualifions de conceptuel. Elle ne comporte aucune information concrètement exploitable à l'exécution sur la façon d'organiser les *produits-modèles*.

Par exemple, considérons deux *WorkProducts* *WP1* et *WP2* ayant un lien d'**Overlap**. Les *produits-modèles* qu'ils modélisent ont donc des éléments communs. Lors de l'exécution du procédé par le *moteur d'exécution*, ce dernier va agir sur ces *produits-modèles* (les créer, les modifier, etc.). Ainsi à l'exécution, ces *produits-modèles* sont donc des entités concrètes stockées et utilisées par le *moteur d'exécution*. Il est alors nécessaire de déterminer comment organiser le stockage des éléments communs entre plusieurs *produits-modèles*. Ils peuvent, en effet être stockés:

1. soit dans l'un des *produits-modèles* puis référencés dans l'autre,
2. soit dans les deux *produits-modèles* avec l'utilisation d'une troisième structure contenant des annotations indiquant les éléments communs,
3. soit dans un troisième *produit-modèle* avec des références dans les deux premiers *produits-modèles.*
4. etc.

Il est ainsi nécessaire de préciser au moteur d'exécution comment il doit utiliser les informations du *modèle de procédé* pour organiser le stockage des entités concrètes correspondant aux *produits-modèles*.

Pour cela, et en conformité avec le principe de séparation des préoccupations [Hürsch & Lopes 1995; Mili et al. 2004] largement adopté en Génie Logiciel, nous choisissons de ne pas inclure ces informations dans le *méta-modèle* de *WorkProducts* qui est dédié à la modélisation de niveau conceptuel.

Ainsi nous proposons que ces informations relatives à l'organisation d'entités concrètes et relevant d'un niveau logique soient modélisées conformément à un *méta-modèle* de **SoftwareProcessObjects** (du nom que nous attribuons aux entités concrètes stockées et

manipulées par le *moteur d'exécution*). Celui-ci a donc pour rôle de prescrire la façon dont les spécifications contenues dans un *modèle de procédé* sont utilisées pour aider le *moteur d'exécution* à assurer la cohérence, l'évolution et la modularité des *produits-modèles*. Le *méta-modèle* contient le concept de **SoftwareProcessObject**. Ce dernier désigne tout ou partie d'un *produit-modèle* selon qu'il est composé ou non, ou qu'il partage ou non des éléments de modèle avec un autre *produit-modèle*. Il contient également le concept de **SoftwareProcessObjectRelation** qui désigne à son tour une relation entre deux ou plusieurs **SoftwareProcessObjects**. Les deux spécialisations (**NestSPOR** et **OverlapSPOR**) de ce dernier concept correspondent respectivement aux relations d'inclusion et de partage d'éléments de modèles entre *produits-modèles* actuellement supportés. Le *méta-modèle* introduit enfin le concept de **SPOObserver** qui joue un rôle d'observateur pour les **SoftwareProcessObjects**. A la section 4.4, nous reviendrons sur chacun de ces concepts du *méta-modèle* de **SoftwareProcessObjects** lors de la présentation détaillée de celui-ci.

Un *moteur d'exécution* prend en entrée un *modèle de procédé* puis l'utilise comme base pour construire la structure des **SoftwareProcessObjects** correspondant aux **WorkProducts** qu'il contient. Il est donc nécessaire de définir des règles de correspondance entre les concepts des deux *méta-modèles* précédemment nommés. Notre proposition comporte également une définition de telles règles qui ainsi la complètent.

Dans les sections suivantes de ce chapitre, nous présentons dans les détails chacun des éléments annoncés de notre proposition. Nous présentons d'abord le *méta-modèle* de *WorkProducts* ainsi que chacun des concepts qu'il contient puis l'utiliserons pour modéliser le procédé-exemple défini précédemment au chapitre 2. S'en suivra une présentation du *méta-modèle* de **SoftwareProcessObject** avec celle des concepts qu'il introduit. Enfin seront présentées les correspondances qui s'établissent entre les concepts de ces deux *méta-modèles*.

4.3 Le *méta-modèle* de *WorkProducts*

Dans les précédents chapitres, nous avons évoqué la nécessité pour un *moteur d'exécution* de *procédé* de disposer de plus de détails sur la structure des *produits-modèles* dans le but d'assurer leur modularité et leur *cohérence relationnelle*, et d'automatiser leur évolution. De tels détails ne peuvent être déduits ex nihilo par un *moteur d'exécution* car ils sont relatifs au

procédé [Curtis et al. 1992; Scacchi 2001] et doivent donc être précisés lors de la définition de celui-ci. Les *LMPs* couramment utilisés ne supportent pas une telle possibilité, comme démontré dans le chapitre 3. Ils ont alors besoin d'être étendus afin de permettre aux personnes chargées de modéliser un *procédé* de décrire les futurs *produits-modèles* avec une précision pouvant être exploitée lors de l'exécution du *procédé*. A ce niveau l'idée soutenue par notre proposition est de modéliser toutes les relations possibles entre *produits-modèles* en utilisant des concepts structurés à travers le *méta-modèle* de *WorkProducts* présenté ci-après. Pour rappel, les *WorkProducts* correspondent aux éléments d'un *modèle de procédé* qui sont spécifiques aux *produits-modèles* de ce *procédé* et à leurs relations (voir définition 7, chapitre 2). Le *méta-modèle* proposé et présenté ci-après peut ainsi être vu comme une réadaptation au contexte IDM des concepts utilisés pour représenter les *produits* dans les *LMPs* étudiés dans l'état de l'art (chapitre 3).

La suite de cette section est consacrée à la présentation du *méta-modèle* de *WorkProducts* proposé et, en l'occurrence, de celle de la sémantique associée à chacun de ses concepts.

4.3.1 Le méta-modèle

La **Figure 4.1** présente le *méta-modèle* de *WorkProducts* proposé. Par souci de clarté, nous avons choisi d'utiliser le même style que celui employé dans le standard UML2.0 de l'OMG [OMG-UML2 2009; OMG-UML2 2007] pour décrire les méta-classes du *méta-modèle*. Ainsi, pour chaque méta-classe sont donnés sa description et/ou sa sémantique, ses éventuels liens de généralisation avec d'autres méta-classes du *méta-modèle* ou d'autres *méta-modèles*, ses attributs, ses associations avec les autres méta-classes du *méta-modèle*, les contraintes qui lui sont appliquées ainsi que la notation qui lui est associée.

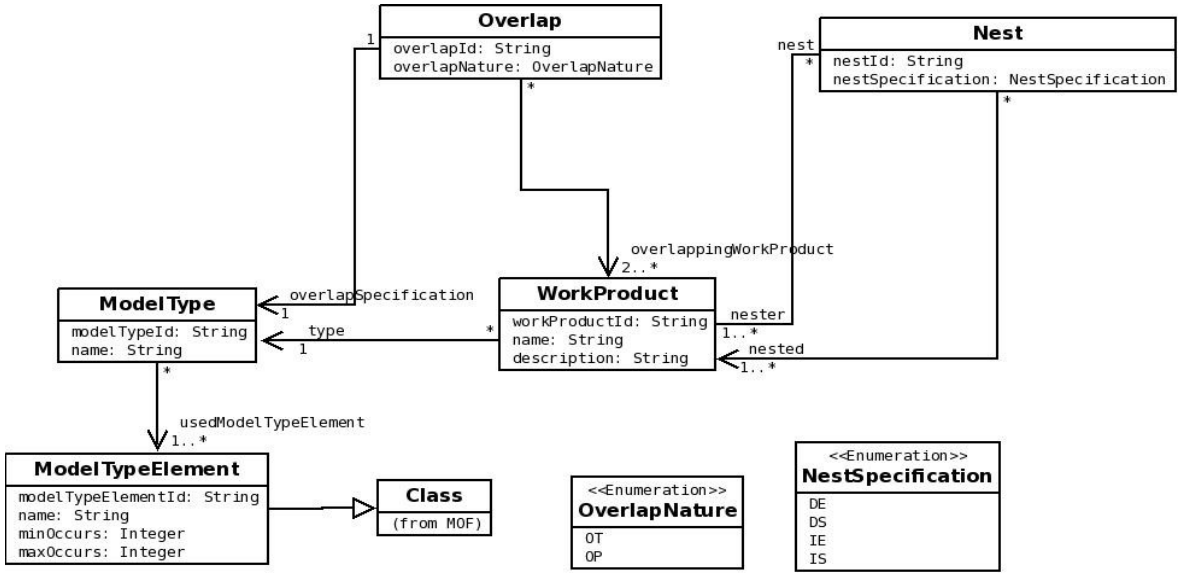


Figure 4.1: Le *Méta-Modèle* de *WorkProducts*

4.3.1.1 WorkProduct

4.3.1.1.1 Description et sémantique

Nous utilisons le concept de **WorkProduct** pour représenter les *WorkProducts* d'un *procédé*. En référence à la définition 7 du chapitre 2, le concept de **WorkProduct** du *méta-modèle* représente le *modèle* d'un *produit-modèle*; il correspond donc à la spécification d'un *produit-modèle*, et non le *produit-modèle* lui-même. Il est l'élément à partir duquel le raisonnement du *moteur d'exécution* sur le *produit-modèle* va se fonder lors de l'exécution du *procédé*.

4.3.1.1.2 Généralisations

Aucune généralisation définie pour le moment.

4.3.1.1.3 Attributs

workProductId: String

Un identifiant unique du *WorkProduct* durant la modélisation et l'exécution du *procédé*.

name: String

Un nom utilisé pour faire référence au *WorkProduct* dans le *modèle de procédé*.

description: String

Une description du *WorkProduct* pouvant être utilisée au cours de l'exécution du *procédé*.

4.3.1.1.4 Associations

nest: Nest [*]

Permet de désigner, si elle(s) existe(nt), la ou les relations(s) *Nest* dans (les)laquelle(s) un *WorkProduct* est impliqué. Chaque élément *Nest* associé permet de spécifier une relation mettant en jeu un ou plusieurs autre(s) *WorkProducts* (dit(s) ***nested***) de *granularité* plus fine et inclus dans le *WorkProduct* considéré (appelé ***nester***). Ce concept est plus amplement défini dans la sous-section 4.3.1.5 de ce chapitre.

type: ModelType [1]

Le type du *WorkProduct*. Il est défini sous la forme d'un **ModelType**. Le concept de **ModelType** est défini dans la suite, à la sous-section 4.3.1.2.

4.3.1.1.5 Contraintes

Aucune contrainte définie pour le moment.

4.3.1.1.6 Notation

Aucune notation définie pour le moment.

4.3.1.2 ModelType

4.3.1.2.1 Description et sémantique

Cette méta-classe représente la spécification d'un type de *WorkProduct*. Elle décrit le *type de modèle* (voir la *définition 17*, chapitre 2) qui est associé à un ou plusieurs *produit(s)-modèle(s)*. Notre proposition de l'introduire dans le *méta-modèle* est motivée par le fait que la définition du concept d'**Overlap** s'appuie fortement sur le typage des *produits-modèles*. L'**Overlap** est le concept du *méta-modèle* utilisé pour modéliser la relation de partage d'éléments de modèles entre deux ou plusieurs *produits-modèles* (voir sa définition détaillée dans la suite, section 4.3.1.4). Un **ModelType** est décrit par son identifiant et son

nom qui sont des chaines de caractères, il est défini sous la forme d'un ensemble de **ModelTypeElements**. Ce dernier est un autre concept du *méta-modèle* et est défini en section 4.3.1.3.

4.3.1.2.2 Généralisations

Aucune généralisation définie pour le moment.

4.3.1.2.3 Attributs

modelTypeId: String

Un identifiant unique du **ModelType** durant la modélisation et l'exécution du *procédé*.

name: String

Un nom utilisé pour faire référence au **ModelType** dans le *modèle de procédé*.

4.3.1.2.4 Associations

modelTypeElements: ModelTypeElement[1..*]

Permet de désigner les méta-classes qui forment le **ModelType**.

4.3.1.2.5 Contraintes

Aucune contrainte définie pour le moment.

4.3.1.2.6 Notation

Aucune notation définie pour le moment.

4.3.1.3 ModelTypeElement

4.3.1.3.1 Description et sémantique

Un **ModelTypeElement** représente une méta-classe utilisée dans la construction d'un **ModelType**. En effet, un **ModelType** est, comme précédemment expliqué ci-haut en section 4.3.1.2 de ce chapitre, une collection de **ModelTypeElements** créés pour cet unique besoin. Un **ModelTypeElement** est décrit par son identifiant et son nom qui sont deux chaines de caractères. Il est également décrit par deux entiers (***minOccurs*** et ***maxOccurs***) qui permettent de représenter respectivement les nombres minimal et maximal d'instances de la méta-classe représentée dans un *produit-modèle*. En effet, si un **WorProduct** *wP* est typé par un **ModelType** *mT* dont *mTE* est un **ModelTypeElement**, alors pour un *produit-modèle pM*

spécifié par *wP* il est possible d'indiquer, par le biais de ces deux nombres entiers, les nombres minimal et maximal d'éléments de modèles de *pM* instances de la méta-classe correspondant à *mTE*. Ces entiers offrent la possibilité lors de la création d'un *modèle de procédé* de mettre des contraintes sur la présence ou non d'éléments de modèles instances de telle ou telle méta-classe. Par exemple, supposons que *mTE* correspond à un élément de *ModelType* qui correspond à la méta-classe *UseCase* de UML2, supposons en plus que *mTE.minOccurs* et *mTE.maxOccurs* valent, respectivement, 1 et -1. A l'exécution du *procédé*, le moteur d'exécution va s'assurer que le *produit-modèle pM* contient au moins un *use case* UML 2 parmi ses éléments de modèles.

4.3.1.3.2 Généralisations

L'idée de base du système de typage utilisé dans [Jim Steel & Jézéquel 2007], et que nous avons exploitée, est que le type d'un *modèle* est un ensemble de méta-classes, la méta-classe **ModelTypeElement** dispose d'une généralisation qui est la méta-classe **MOF::Class**.

4.3.1.3.3 Attributs

modelTypeElementId: String

Un identifiant unique du **ModelTypeElement** durant la modélisation et l'exécution du *procédé*.

name: String

Un nom utilisé pour faire référence au **ModelTypeElement** dans le *modèle de procédé*.

minOccurs: Integer

Le nombre minimal d'instances de la méta-classe représentée par le **ModelTypeElement** dans un *produit-modèles* spécifié par un **WorkProduct** typé par un **ModelType** dont le **ModelTypeElement** est un ***usedModelTypeElement***.

maxOccurs: Integer

Le nombre maximal d'instances de la méta-classe représentée par le **ModelTypeElement** dans un *produit-modèles* spécifié par un **WorkProduct** typé par un **ModelType** dont le **ModelTypeElement** est un ***usedModelTypeElement***.

4.3.1.3.4 Associations

Aucune association n'est définie pour le moment.

4.3.1.3.5 Contraintes

Aucune contrainte définie pour le moment.

4.3.1.3.6 Notation

Aucune notation définie pour le moment.

4.3.1.4 Overlap

4.3.1.4.1 Description et sémantique

Le concept d'**Overlap** est utilisé pour exprimer le fait que deux ou plusieurs *WorkProducts* se partagent des éléments communs. Etant donné qu'un *WorkProduct* représente une spécification de *produit-modèles*, l'intersection entre deux (ou plus) *WorkProducts* est donnée sous la forme d'un ensemble de méta-classes (représentées dans le *modèle de procédé* sous forme de **ModelTypeElements**) qui alors sont communes aux spécifications considérées. Cet ensemble de méta-classes est appelé *spécification de l'Overlap* et correspond à un **ModelType** du *modèle de procédé*. L'existence d'une relation d'**Overlap** entre des *WorkProducts* signifie que les *produits-modèles* spécifiés par de tels *WorkProducts* peuvent avoir des éléments de modèles communs et que ces éléments communs seront des instances d'éléments de la spécification de l'**Overlap**. En plus de sa spécification, un **Overlap** est décrit par son identifiant et sa nature qui sont des chaines de caractères.
Du point de vue de la modélisation, c'est ce concept d'**Overlap** que nous avons proposé qui permet de répondre à l'objectif de faire évoluer les *produits-modèles* de façon automatique ou semi-automatique lors de l'exécution, et aussi de maintenir à priori la cohérence relationnelle liée au partage d'éléments entre eux. Nous allons montrer cela plus en détail lors de la présentation de la solution du point de vue de l'exécution.

4.3.1.4.2 Généralisations

Aucune généralisation n'est définie pour le moment.

4.3.1.4.3 Attributs

overlapId: String

Un identifiant unique de l'**Overlap** durant la modélisation et l'exécution du *procédé*.

overlapNature: OverlapNature

La nature de l'**Overlap**. Elle permet de dire, par exemple, et en référence aux

définitions de Spanoudakis et al [Spanoudakis et al. 1999], si la relation est totale, partielle ou inclusive. Dans notre étude, nous nous sommes pour le moment limités à la prise en compte des deux formes suivantes:

- **OT** (**Overlap** totale): les éléments modélisés par la *spécification de l'Overlap* sont tous communs à tous les *produits-modèles* liés. Au cours de la modélisation d'un *procédé*, un lien d'**Overlap** de cette nature permet, par exemple d'exprimer que « tous les éléments de type *classe* d'un *produit-modèle pM1* sont aussi des éléments d'un *produit-modèle pM2* ».
- **OP** (**Overlap** partielle): certains éléments modélisés par la *spécification de l'Overlap* sont communs à tous les *produits-modèles* liés, les autres non. On utilisera un **Overlap** de cette nature pour, par exemple lors de la modélisation d'un *procédé*, exprimer que «certaines classes d'un *produit-modèle pM1* sont aussi des classes d'un *produit-modèle pM2* ».

4.3.1.4.4 Associations

overlapSpecification: ModelType [1]

La spécification de l'**Overlap**. Elle est donnée sous la forme de méta-classes regroupées pour former un **ModelType**.

overlappingWorkProduct: WorkProduct [2..*]

Les deux ou plusieurs **WorkProducts** entre lesquels existe et est définie la relation d'**Overlap**.

4.3.1.4.5 Contraintes

Aucune contrainte définie pour le moment.

4.3.1.4.6 Notation

Aucune notation définie pour le moment.

4.3.1.5 Nest

4.3.1.5.1 Description et sémantique

Le concept de **Nest** est utilisé pour la spécification d'une relation entre deux *WorkProducts* traduisant le fait que l'un (le ***nested***) soit inclus dans l'autre (le ***nester***). La relation d'inclusion modélisée à travers le concept de **Nest** fait abstraction des types (**ModelTypes**) des deux *WorkProducts* liés; en effet, aucune contrainte ou condition n'est posée sur les types de ces *WorkProducts*.

En rapport aux objectifs énoncés au chapitre 2 et comme annoncé en synthèse plus haut en section 4.2, le concept de **Nest** a été introduit afin, d'une part, de permettre d'assurer une meilleure gestion de la modularité des *produits-modèles* lors de l'exécution d'un *procédé*. En effet, le fait de spécifier une relation **Nest** entre deux **WorkProducts** d'un *modèle de procédé* fournit au *moteur d'exécution* la possibilité d'avoir simultanément deux vues différentes sur les *produits-modèles* correspondants lors de l'exécution du *procédé*. La première vue (la vue séparatiste) est la plus naturelle. Les deux *produits-modèles* y sont considérés comme des entités séparées les unes des autres. C'est cette vue qui est supportée par les *LMPs*/*moteurs d'exécution* étudiés dans l'état de l'art. La seconde vue est plutôt ensembliste. L'ensemble composé des *produits-modèles* correspondants y est considéré comme une entité globale sous la forme d'un *produit-modèle* composé dont les éléments de modèles sont le regroupement de ceux de ses composants. Avec ces deux vues, le *moteur d'exécution* a la possibilité de voir deux ou plusieurs *produits-modèles* d'un *procédé* en exécution comme (naturellement) des entités séparées les unes des autres tout en gardant le moyen, au besoin, de calculer et obtenir leur « somme » sous la forme d'un autre *produit-modèle* lui aussi accessible, comme ses différentes parties, aux activités du *procédé*. L'introduction du **Nest** permet alors de donner à la relation de composition classiquement descriptible à l'aide des *LMPs* existants une sémantique permettant de l'adapter au contexte des *produits-modèles*. Ce lien offre ainsi, dans le contexte IDM, la possibilité d'une plus grande flexibilité du point de vue de la *granularité* des *WorkProducts* d'un *modèle de procédé* et ainsi de celle des *produits-modèles* spécifiés dont la modularité devient accrue.

D'autre part, le **Nest** offre la possibilité d'un contrôle d'une *cohérence relationnelle* que nous qualifions de niveau « *modèle* » entre les *produits-modèles* d'un *procédé* en exécution. Cet

aspect est développé en détail à la section 4.3.1.5.3, qui présente l'attribut du **Nest** utilisé pour indiquer la spécification d'une telle relation.
Nous reviendrons un peu plus sur les avantages du **Nest** lors de la présentation de la solution du point de vue de l'exécution dans la section 4.5.

4.3.1.5.2 Généralisations

Aucune généralisation n'est définie pour le moment.

4.3.1.5.3 Attributs

nestId: String

Un identifiant unique du **Nest** durant la modélisation et l'exécution du *procédé*.

nestSpecification: NestSpecification

Permet de spécifier la nature, la spécification de la relation **Nest**.

Pour plus de clarté dans la suite, nous utilisons les éléments de *modèles de procédé* définis suivants ainsi que les *produits-modèles* qui leur sont associés. C'est ainsi que nous considérons un lien **Nest** *n* défini dans un *modèle de procédé* donné. Nous considérons en plus que *n* est défini entre un **WorkProduct *nester*** que nous appelons le *nesterWP* et plusieurs autres **WorkProducts *nested*** que nous appelons les *nestedWPs*. A l'exécution du *procédé* considéré, soit *nesterPM* le *produit-modèle* spécifié par le *nesterWP*, et soit *nestedPM* le nom que nous donnons à un *produit-modèle* spécifié par un *nestedWP* pris parmi les *nestedWPs* (le *nestedPM* est alors « inclus dans» le *nesterPM*).

Won et al. ont réalisé d'importants travaux sur la notion d'objet composé; ils sont présentés dans [Kim 1989; Kim et al. 1987]. Par analogie à ces travaux nous qualifions le lien ***n*** de *dépendant* si à l'exécution du *procédé*, l'existence de tout *nestedPM* dépend de celle du *nesterPM*. Autrement dit, un *nestedPM* ne peut être créé que si le *nesterPM* l'a déjà été, et la suppression du *nesterPM* entraine celle de tous les *nestedPM* alors existants. Un lien **Nest** non *dépendant* est alors qualifié d'*indépendant*.
Toujours en référence aux mêmes travaux de Won et al., le lien *n* est dit *partagé* si tout *nestedPM* est en même temps impliqué, avec un rôle similaire, dans au moins un autre lien **Nest** que *n*. Un lien **Nest** non *partagé* est alors qualifié d'*exclusif*.

De la définition des propriétés du **Nest** qui précèdent, nous déduisons les valeurs possibles suivantes pour l'attribut ***nestSpecification***.

- **DE** (Dependent-Exclusive): pour les relations Nest de type dépendantes et exclusives,
- **DS** (Dependent-Shared): pour les relations dépendantes et partagées,
- **IE** (Independent-Exclusive): pour les relations indépendantes et exclusives,
- **IS** (Independent-Shared): pour les relations indépendantes et partagées.

La spécification d'un lien **Nest** de la façon décrite ci-avant donne de nouvelles possibilités à l'exécution du *procédé* dans le *modèle* duquel il est utilisé. Nous résumons ces possibilités de la façon qui suit.

- Il devient possible de, par exemple, justifier l'existence ou non d'un *produit-modèle* en fonction de l'existence ou de l'inexistence d'un ou de plusieurs autres, à l'aide de la propriété de dépendance.
- Également, la propriété de partage/exclusivité rend possible l'indication de l'exclusivité ou non de l' « inclusion » d'un *produit-modèle* par un autre.

En définitive, il en ressort la possibilité d'un contrôle d'une forme de *cohérence relationnelle* entre les *produits-modèles*. En effet, il est illustré dans cette section que la spécification d'un lien **Nest** à la modélisation d'un *procédé* permet d'assurer une intégrité sémantique des *produits-modèles* impliqués lors de son exécution.

4.3.1.5.4 Associations

nester: WorkProduct [1..*]

Désigne le ou les **WorkProduct(s)** composé(s) de la relation.

nested: WorkProduct [1..*]

Désigne le ou les WorkProduct(s) composant(s) de la relation.

4.3.1.5.5 Contraintes

- Une relation **Nest** ne peut pas être exclusive si un de ses **WorkProducts** ***nested*** joue déjà le rôle de ***nested*** dans une autre relation **Nest** du *modèle de procédé* en cours de définition.
- Un *modèle de procédé* ne peut comporter de relation **Nest** transitive entre

WorkProducts. Plus clairement, soit *WP1*, *WP2* et *WP3* trois **WorkProducts** tels qu'ils existent une relation **Nest** entre *WP1* et *WP2* et une autre entre *WP2* et *WP3*. La relation **Nest** entre *WP1* et *WP3* qui est implicite ne sera pas définie de façon explicite dans le même *modèle de procédé*.

Object Constraints Language (OCL) est un langage formel, orienté objet et permet l'ajout de contraintes pour exprimer la sémantique statique des modèles et donc des méta-modèles. Elle a fait l'objet d'une spécification officielle de l'OMG présentée dans [OMG-OCL 2010]. Ci-après, nous utilisons OCL pour respectivement décrire les contraintes exprimées ci-haut.

- ```
  context Nest inv :
  if(nested->includes(wp:WorkProduct|
  wp.nest->includes(self))) then
     nestSpecification<>NestSpecification::DE
     and nestSpecification<>NestSpecification::IE
  endif
  ```

- ```
  context WorkProduct inv :
  if self.nest->notEmpty() then
     let wp1 be self.nest.nested in
          if wp1.nest->notEmpty() then
            wp1.nest.nested->excludesAll(wp1)
          endif
   endif
  ```

4.3.1.5.6 Notation

Aucune notation définie pour le moment.

4.3.2 Conclusion

Dans cette section, nous avons présenté le *méta-modèle* proposé pour les *WorkProducts* de *procédés*. Ce *méta-modèle* comporte le concept de **WorkProduct** pour représenter la spécification des *produits-modèles* d'un *procédé* dans MDE. Une spécification est associée à un type sous la forme d'un **ModelType** qui est un ensemble de méta-classes représentées par

le concept de **ModelTypeElement**. Le *méta-modèle* fait ensuite ressortir deux concepts permettant de spécifier les deux relations entre *produits-modèles* supportées pour le moment; ces concepts sont l'**Overlap** et le **Nest**. Chacun de ces concepts est décrit dans cette section et la motivation de sa définition expliquée. C'est ainsi que le **Nest** est proposé pour spécifier une relation d'inclusion entre deux *WorkProducts* d'un *procédé*, offrant ainsi la possibilité d'une gestion modulaire des *produits-modèles* correspondants lors de l'exécution ainsi que celle d'une gestion de leur *cohérence relationnelle* que nous avons qualifiée de niveau «*modèle* ». En offrant la possibilité à l'exécution de gérer le partage d'éléments de modèles entre *produits-modèles* différents, l'**Overlap** quant à lui correspond à une solution qui nous permettra d'assurer un stockage unique des éléments communs et ainsi de leur assurer une *cohérence relationnelle* (cette fois de niveau « élément de modèle ») entre les *produits-modèles* liés mais aussi leur évolution automatique ou semi-automatique comme nous allons le montrer dans les sections suivantes.

Notre proposition comporte également un *méta-modèle* dont le but est de structurer les entités qui, lors de l'exécution d'un *procédé*, assurent la réification des éléments du *modèle de procédé* correspondant. Ce sont ces entités à travers lesquelles le *moteur d'exécution* assure la gestion des *produits-modèles* du *procédé*. Avant de présenter ce *méta-modèle*, et dans le but de faciliter la compréhension des concepts qu'il définit, un exemple d'utilisation du *méta-modèle* de *WorkProducts* proposé est présenté ci-après.

4.4 Exemple: Modélisation et exécution du procédé-exemple

Dans le but d'aider à une meilleure compréhension du *méta-modèle* de *WorkProducts*, nous présentons, dans cette section, une modélisation du procédé-exemple présenté au chapitre 2 à l'aide des concepts nouvellement introduits à travers ledit *méta-modèle*. Nous décrirons ensuite la façon suivant laquelle les éléments de *modèle de procédé* obtenus sont exploités par le *moteur d'exécution*. Cela nous permettra, à la fois, de mettre en exergue la possibilité offerte par le *méta-modèle* de *WorkProducts*, de prendre en compte les problèmes soulevés plus haut, et aider à mieux comprendre les concepts que nous allons introduire dans la section 4.5.

4.4.1 Le modèle du procédé-exemple

Le procédé-exemple est considéré en prenant en compte les deux préoccupations suivantes présentées dans la section 2.3.3.1.1 du chapitre 2: le problème de la *granularité* du modèle d'analyse d'une part, et d'autre part la présence d'éléments communs entre les modèles d'analyse et de conception. Parce que, comme indiqué lors de la présentation du *méta-modèle* de *WorkProducts* ci-haut, nous n'avons pour l'instant pas défini de notation associée aux concepts nouvellement introduits, une forme libre est utilisée pour écrire le *modèle de procédé*. Celui-ci est représenté par la **Figure 4.2**.

Ce modèle définit les deux activités du procédé-exemple (*Analysis* et *Design*) ainsi que trois **WorkProducts**: *AnalysisModel*, *UseCaseModel* et *DesignModel*. Les activités sont contenues dans une iétarion (*anIteration*). Pour chacun de ces **WorkProducts** est défini et associé un type sous la forme d'un **ModelType** (respectivement *AnalysisModelType*, *UseCaseModelType* et *DesignModelType*). Le *modèle* définit également les méta-classes (**ModelTypeElements**) qui constituent chaque **ModelType**. Enfin le *modèle de procédé* contient les définitions d'un **Nest** et d'un **Overlap** (respectivement *NestX* et *OverlapX*). *NestX* (dont le champ *nestSpécification* vaut «IE») est défini pour traduire le fait que le *produit-modèle* spécifié par *UseCaseModel*, bien qu'étant une entité indépendante, est exclusivement une partie intégrante de celui modélisé par l'*AnalysisModel*. *OverlapX* (dont le champ *overlapNature* vaut « OP ») traduit le fait que les *produits-modèles* spécifiés par les **WorkProducts** *AnalysisModel* et *DesignModel* peuvent avoir en commun des instances de *Class*, *Attribute*, et *Operation*.

```
Iteration: anIteration
        activities: Analysis,Design

SoftwareActivity: Analysis
          output: AnalysisModel

SoftwareActivity: Design
           input: AnalysisModel
           output: DesignModel

WorkProduct: AnalysisModel
           type: AnalysisModelType
           nest: NestX

WorkProduct: DesignModel
           type: DesignModel

WorkProduct: UseCaseModel
          type: UseCaseModelType

Nest: NestX
   specification:"IE"
   nester: AnalysisModel
   nested: UseCaseModel

Overlap: OverlapX
      overlapNature:"OP"
      overlappingWorkProduct: AnalysisModel, DesignModel
      overlapSpecification: OverlapXModelType

ModelType: AnalysisModelType
           usedModeTypeElements: Class, Association, Operation, Attribute, Parameter

ModelType: DesignModelType
           usedModelTypeElements: Class, Attribute, Operation, Association, Parameter
                       LifeLine, Message

ModelType: UseCaseModelType
           usedModelTypeElements: UseCase, Actor

ModelType: OverlapXModelType
           usedModelTypeElements: Class, Attribute, Operation
```

Figure 4.2: Le Modèle du Procédé-Exemple Représenté en Forme Libre

4.4.2 L'exécution du modèle de procédé

Dans cette sous-section, il est question de montrer comment le *moteur d'exécution* utilise les éléments du modèle du procédé-exemple. En l'occurrence, l'accent sera mis sur les éléments *NestX* et *OverlapX*, qui sont principalement employés pour apporter des solutions aux préoccupations relatives aux relations entre les *produits-modèles* du procédé-exemple.

Nous employons l'expression *objet de procédé* (ou simplement *SPO*, pour *Software Process Objects*) pour désigner tout élément représentant (réifiant) un élément de modèle de *procédé* à l'exécution. Les *SPOs* sont créés par le *moteur d'exécution* à partir du *modèle du procédé* en exécution. Nous allons nous restreindre, dans toute la suite du document, aux *SPOs* relatifs aux *produit-modèles* et à leurs relations.

Pour assurer la gestion des *produits-modèles* du procédé-exemple à l'exécution de celui-ci, le *moteur d'exécution* crée les SPOs suivants:

- *AnalysisModelSPO*, *DesignModelSPO*, et *UseCaseModelSPO* (correspondent respectivement aux **WorkProducts** *AnalysisModel*, *DesignModel,* et *UseCaseModel*

du *modèle de procédé*),

- *NestXSPOR* et *OverlapXSPOR* (correspondent respectivement aux relations *NestX* et *OverlapX* du *modèle de procédé*).
- *Observer_AnalysisModelSPO_OverlapXSPOR* et *Observer_DesignModelSPO_OverlapXSPOR* (correspondent respectivement aux **SPOObservers** associés aux **SPOs** *AnalysisModelSPO* et *DesignModelSPO* dans le cadre de l'**Overlap** *OverlapXSPO*).

Nous expliquons dans les paragraphes suivants les éléments qui ont guidé la création des *SPOs* qui correspondent aux relations, en l'occurrence *NestXSPOR* d'une part, et *OverlapXSPOR, Observer_AnalysisModelSPO_OverlapXSPOR* et *Observer_DesignModelSPO_OverlapXSPOR* d'autre part. Ces explications seront suivies de la présentation des autres étapes de l'exécution du procédé-exemple.

La création du *SPO OverlapXSPOR* offre la possibilité d'un stockage unique des éléments de modèle qui sont partagés entre les deux *produits-modèles*. Cela offre un certains nombre d'avantages qui sont décrits ci-après.

- **Le maintien d'une *cohérence relationnelle* des *produits-modèles* impliqués**. Ici, la relation concernée est la relation de partage d'éléments (de types classes, attributs, opérations). Comme les éléments de modèles partagés sont stockés en un seul exemplaire, toute modification faite sur eux est systématiquement répercutés sur tous les *produits-modèles*.
- **La propagation automatique des changements**. Puisqu'il y'a une seule copie d'un élément de modèle partagé, s'il est modifié pour un *produit-modèle* il l'est aussi pour tous les autres. Il en résulte une évolution automatique des *produits-modèles*.
- **La création assistée de *produits-modèles***. Il arrive que la possibilité soit offerte à un développeur de créer un *produit-modèle* en important des éléments de modèles déjà existants en lieu et place d'une création ex nihilo de tous les *produits-modèles*. C'est le cas pour le concepteur qui a la possibilité de réutiliser les classes d'analyse.

C'est donc pour tous ces avantages qui contribuent à une croissance de la productivité des développeurs, que nous avons préconisé que, dans le cas d'une relation d'*Overlap* précise, les éléments communs entre deux ou plusieurs *produits-modèles* soient stockés non pas de façon

séparée dans chacun des *produits-modèles* en relation, mais à la place, qu'ils le soient une seule fois dans un *SPO* (*OverlapXSPOR* dans notre exemple) comme indiquée par la solution 3 annoncée en section 4.2. Stockés en un seul exemplaire, ces éléments de modèles seront alors référencés dans chacun des *SPOs* liés à chaque fois que c'est nécessaire. Il faut néanmoins noter que dans cette configuration, un certain nombre de contraintes se posent à une gestion flexible et respectueuse de la sémantique de la relation. En effet,

- tout enregistrement d'un nouvel élément partagé doit être notifié aux SPOs impliqués dans la relation afin de permettre leur prise en compte ou non,
- il faut également que toute action de modification/suppression d'un élément partagé soit notifiée aux développeurs responsables des produits-modèles impliqués dans la relation. Ces derniers pourront alors décider de prendre en compte ou de rejeter l'action.

C'est dans le but de répondre à ces contraintes que nous avons proposé d'adjoindre un élément appelé *observateur* à chaque *SPO* pour lui permettre de se renseigner sur les autres *produits-modèles* représentés par les autres *SPOs* qui lui sont liés, et ainsi permettre aux développeurs concernés d'être avertis au besoin à chaque fois avant qu'ils ne commencent à travailler sur un *produit-modèle*. C'est suivant cet ordre d'idée que les observateurs *Observer_AnalysisModelSPO_OverlapXSPOR* et *Observer_DesignModelSPO_OverlapXSPOR* sont créés.

La **Figure 4.3** schématise une vue contenant les différents *SPOs* créés par le moteur d'exécution au début de l'exécution du procédé-exemple. Les *SPOs* qui vont recevoir les éléments de *produits-modèles* y sont représentés par des disques ovales bleus remplis et à contour foncé, les *observateurs* par des disques ovales gris à contour fin, et les *SPOs* correspondant aux relations *Nest* par des disque ovales gris à contour discontinu. Des flèches sont utilisées pour représenter la possibilité et le sens de la navigation entre les différents *SPOs*. Aucun des *SPOs* représentés ne contient, à ce niveau, d'élément de *produit-modèle* et sont donc tous dits vides.

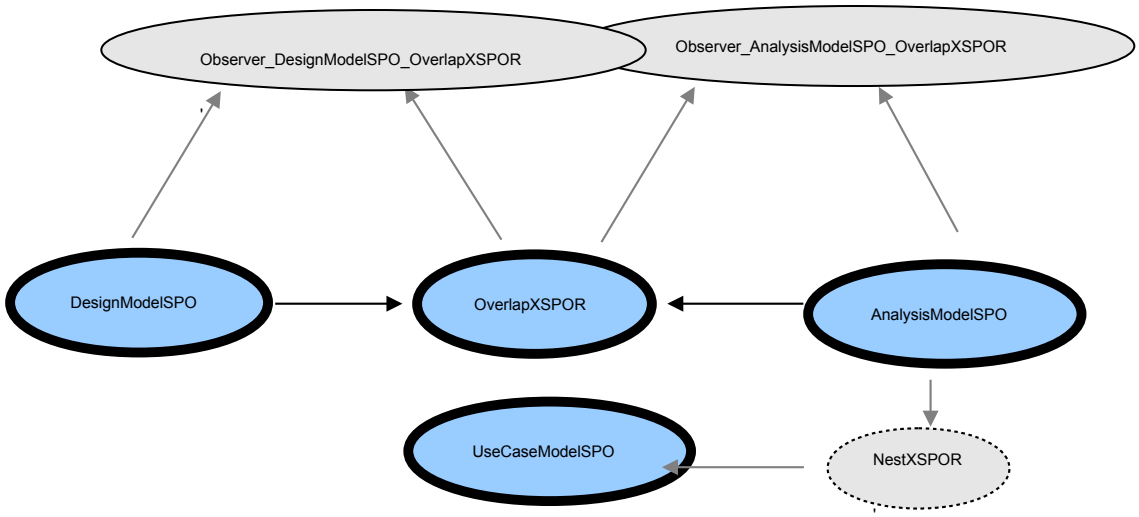


Figure 4.3: Vue Schématique des SPOs du Procédé-Exemple à leur Création

Après l'étape de création des *SPOs*, nous présentons ci-après les autres étapes de l'exécution du procédé-exemple. Ces étapes sont formées par une répétition de la séquence d'activités *Analysis-Design* puisque ces deux activités sont définies dans une itération. C'est ainsi qu'on a la séquence d'évènements présentée ci-après.

Analysis commence...

Etant donné que c'est la première activité du procédé et qu'elle ne prend aucun *produit-modèle* en entrée, le développeur (l'*analyste*) crée *le modèle d'analyse* ex nihilo sur son espace de travail et travaille dessus. Il crée les éléments de modèles correspondants à ce *produit-modèle*. Nous rappelons modélisé que, d'une part, le *modèle d'analyse* comporte les *classes d'analyse*. D'autre part, il inclut (via le lien *NestX*) les *cas d'utilisation*.

Analysis finit...

Quand l'*analyste* déclare avoir terminé son activité, les éléments de modèles qui compose les deux *produits-modèles* créés sont envoyés au *moteur d'exécution* puis enregistrés au niveau des *SPOs* correspondants et schématisés à l'aide de la **Figure 4.3**. Cet enregistrement des éléments du *modèle d'analyse* se fait suivant la façon décrite ci-après et guidée par le choix de stockage explicité précédemment. Pour rappel, la spécification de l'*overlap* défini entre le *modèle d'analyse* et celui de conception contient les méta-classes suivantes: *Class*, *Attribute* et *Operation* (voir **Figure 4.2**). Le schéma de la **Figure 4.5** montre une vue du contenu des *SPOs* après cette étape.

- Toutes les classes, attributs et opérations sont directement enregistrés dans l'*OverlapXSPOR* (1),
- les use case et actor sont enregistrés dans le *UseCaseModelSPO* (4),
- les paramètres d'opérations et les associations sont stockés dans l'*AnalysisModelSPO*, ainsi que des références (2) vers les classes attributs et opérations déjà stockés dans l'*OverlapXSPOR*.
- Au même moment, l'observateur *Observer_DesignModelSPO_OverlapXSPOR* stocke les événements de création (3) ainsi que les détails associés correspondants à chaque nouvel enregistrement d'élément dans l'*OverlapXSPOR*.

La **Figure 4.4** représente, quant à elle, la vue que l'*analyste* a du *modèle d'analyse* à cette étape, via son outil de modélisation.

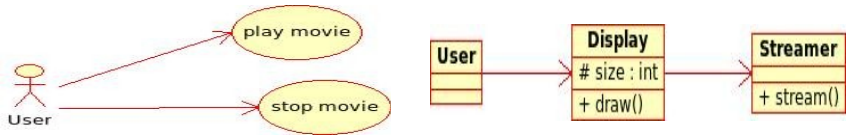


Figure 4.4: Vue Correspondant au Modèle d'Analyse du Procédé-Exemple à la Fin de l'Activité *Analysis*

Les SPOs *UseCaseModelSPO* et *NestXSPO* n'intervenant pas dans la suite de l'exemple, nous les omettrons dans les schémas qui vont suivre.

Design commence...

Au démarrage de l'activité, avec l'aide des informations stockées par l'observateur *Observer_DesignModelSPO_OverlapXSPOR*, le *concepteur* (développeur en charge de l'exécution de l'activité *design*) est averti de la création des nouvelles classes et opérations, et des nouveaux attributs pour lui permettre d'accepter ou d'ignorer ces nouveaux éléments de modèles dans le modèle de conception. C'est ainsi qu'il accepte les classes *Display* et *Streamer* et rejette la classe *User*. Le *modèle de conception* est alors créé dans l'espace de travail du développeur avec les éléments de modèle acceptés comme contenu initial. C'est sur ce *produit-modèle* que le *concepteur* va faire son travail. Les évènements notifiés au *concepteur* sont alors supprimés de l'observateur *Observer_DesignModelSPO_OverlapXSPOR* qui, par ailleurs, enregistre les rejets et acceptations d'éléments.

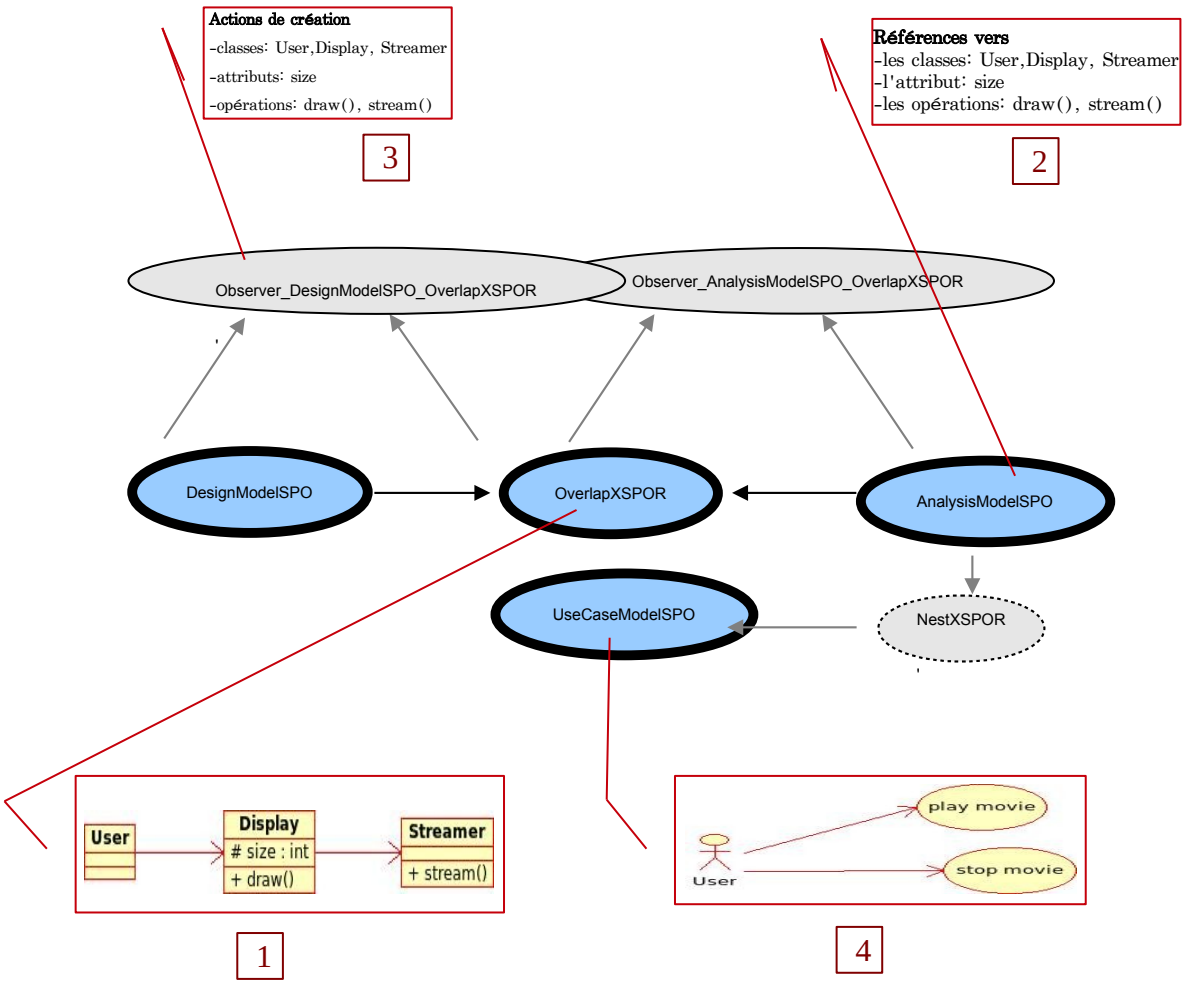


Figure 4.5: Vue Schématique du Contenu des SPOs du Procédé-Exemple à la Fin de l'Activité *Analysis*

Design finit...

Comme pour l'analyse, à la fin déclarée de l'activité de conception, les nouveaux éléments de modèles correspondant au travail du *concepteur* sur le *modèle de conception* sont envoyées au *moteur d'exécution* puis enregistrés au niveau des *SPOs* de la façon décrite ci-après. Le schéma de la **Figure 4.7** montre une vue du contenu des *SPOs* après cette étape.

- Toutes les classes, attributs et opérations nouvellement créés sont directement enregistrés dans l'*OverlapXSPOR* (3). Il s'agit ici de la classe *KeyBoard* avec son attribut *language* et son opération *send()*, et des opéraions *draw()* et *select()* de *Display*.
- Les paramètres d'opérations, les associations et les éléments d'interaction (Lifelines,

Messages) sont stockés dans le *DesignModelSPO* (4).

- Au même moment, l'observateur *Observer_AnalysisModelSPO_OverlapXSPOR* enregistre les événements de création ainsi que les détails correspondant à chaque élément nouvellement enregistré dans l'*OverlapXSPOR* (5). Pour rappel, les actions de rejet et d'acceptation ont déjà été enregistrées à l'aide de ce même observateur (1) au commencement de cette activité de *design.*
- Le contenu de l'*AnalysisModelSPO* ne change pas dans cette activité; il reste tel que décrit plus haut, i.e, les références vers les classes d'analyse (2).

La **Figure 4.6** représente les vues respectives qui correspondent aux *classes d'analyse* d'une part (partie a)), et au *modèle de conception* d'autre part (partie b)), à la fin de l'activité *design*, au sein des outils de modélistion des deux développeurs (*analyste* et *concepteur*).

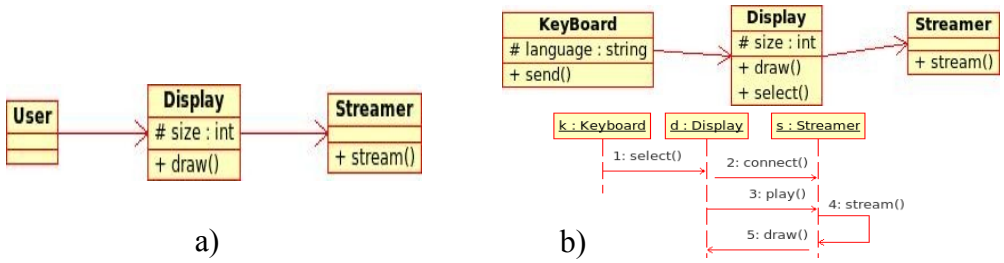


Figure 4.6: Vues Correspondant aux Modèles d'Analyse (*a*)) et de Conception (*b*))

du Procédé-Exemple à la Fin de l'Activité *Design*

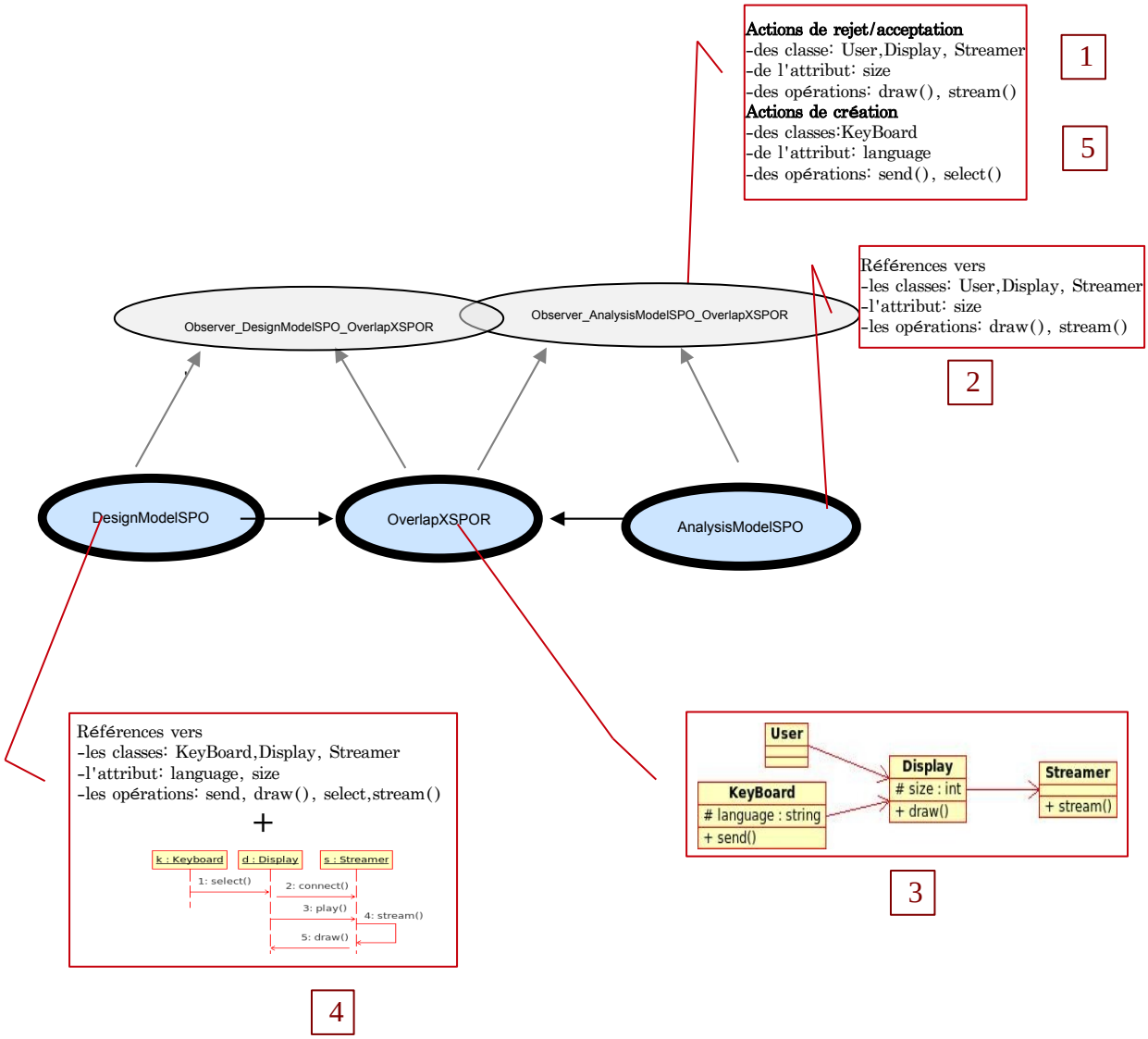


Figure 4.7: Vue Schématique du Contenu des SPOs du Procédé-Exemple à la Fin du *Design*

Analysis recommence...

Au démarrage de l'activité, avec l'aide des informations stockées par l'observateur *Observer_AnalysisModelSPO_OverlapXSPOR*, l'analyste est averti de la création des nouvelles classes (ici *KeyBoard*) et opérations (*draw()* et *select()* de *Display* et des nouveaux attributs (*language* de *KeyBoard*). Il a ainsi la possibilité de les intégrer ou non au *modèle d'analyse*. Le *modèle d'analyse* est alors recréé dans l'espace de travail du développeur avec

les éventuels nouveaux éléments acceptés (ici aucun), les éléments de modèles contenus dans l'*AnalysisModelSPO*, ceux de l'*OverlapXSPO* dont des références sont présentes dans l'*AnalysisModelSPO,* en plus de ceux marqués, au sein de l'observateur *Observer_AnalysisModelSPO_OverlapXSPO,* comme précédemment acceptés par le développeur. C'est sur ce *produit-modèle* que l'analyste fait son travail à nouveau. L'observateur aura auparavant enregistré les acceptations et rejets d'éléments opérés par le développeur.

Ainsi de suite...

C'est suivant des scenarii semblables que l'itération continuera de s'exécuter jusqu'à sa fin. Celle-ci correspondra à la fin de l'exécution du procédé-exemple.

En résumé

L'exemple qui précède donne un aperçu de comment le *moteur d'exécution* utilise les spécifications contenues dans le *modèle* du procédé-exemple pour rendre productives les relations qui existent entre les *produits-modèles*. Cela transparaît à travers les éléments suivants qu'il met en exergue.

- Avec l'aide de l'élément *OverlapX* du *modèle de procédé* (**Figure 4.2**), durant toute l'exécution, le *produit-modèle classes d'analyse* est maintenu cohérent par rapport au *modèle de conception*. Ces deux *produits-modèles* sont spécifiés, respectivement, par les deux *WorkProducts* du *modèle de procédé* qui sont impliqués dans *OverlapX* qui à son tour spécifie la relation de partage d'éléments entre les deux *produits-modèles*. De même, les modifications opérées sur l'un des deux *produits-modèles* sont automatiquement propagées sur l'autre, y compris celles relatives à la création de nouveaux éléments de *produit-modèle*. A cet effet, le *moteur d'exécution* fonctionne de la façon suivante. Il stocke les *produits-modèles* sous forme de SPOs à la fin de l'activité qui les produit ou les modifie, et les restaure en début d'activité. Pour démarrer son activité, il permet à chaque développeur de disposer en local d'une vue exacte de chaque *produit-modèle* nécessaire. Par exemple, la **Figure 4.6** représente les vues respectives correspondant aux *classes d'analyse* d'une part (a), et au *modèle de conception* d'autre part (b), à la fin de l'activité *design*. A ce même moment, au niveau des *SPOs* n'est stocké qu'un seul exemplaire de chaque élément de modèle comme

c'est schématisé à la **Figure 4.7**.

- Du fait de l'élément *NestX* du *modèle de procédé* (voir **Figure 4.2**), on note une modularité du *modèle d'analyse* tout le long de l'exécution. En effet, pendant tout le traitement lié à la relation de partage d'éléments, on ne s'est intéressé qu'à la partie de ce *produit-modèle* appelée *les classes d'analyse*. L'autre partie (les *cas d'utilisation*) de l'ensemble, n'est, en effet, pas impliquée dans cette relation et n'a donc pas besoin que les traitements qui lui sont relatifs lui soient appliqués.

Cet exemple a permis d'utiliser le *méta-modèle* de *WorkProducts* pour modéliser notre procédé-exemple décrit au chapitre 2. Nous avons ensuite décrit la façon suivant laquelle le *moteur d'exécution* utilise le *modèle de procédé* obtenu pour créer et les *SPOs* à travers lesquels les *produits-modèles* sont gérés. En vue d'une capitalisation des résultats mis en exergue par cet exemple, nous avons structuré l'ensemble des concepts qui correspondent aux *SPOs* à travers le *méta-modèle* de *SPOs* précédemment annoncé et que nous présentons dans la section suivante.

4.5 Le méta-modèle de SPOs

Dans cette section, nous allons présenter le *méta-modèle* de *SPOs* ainsi que la sémantique associée à chacun de ses concepts.

La **Figure 4.8** représente le *méta-modèle* que nous proposons pour représenter les *SPOs*. Pour rappel, ce sont les éléments à travers lesquels un *moteur d'exécution* gère les *produits-modèles* d'un *procédé* durant l'exécution. Les concepts définis par le *méta-modèle* ne représentent donc pas des descriptions abstraites de *produits-modèles* comme c'est le cas pour le *méta-modèle* de *WorkProducts*. Au contraire, ils correspondent à des descriptions concrètes de ces *produits-modèles*. Autrement dit, le *méta-modèle* décrit comment les *produits-modèles* sont manipulés par le *moteur d'exécution*. Chacun des concepts du *méta-modèle* est décrit dans la suite de la section.

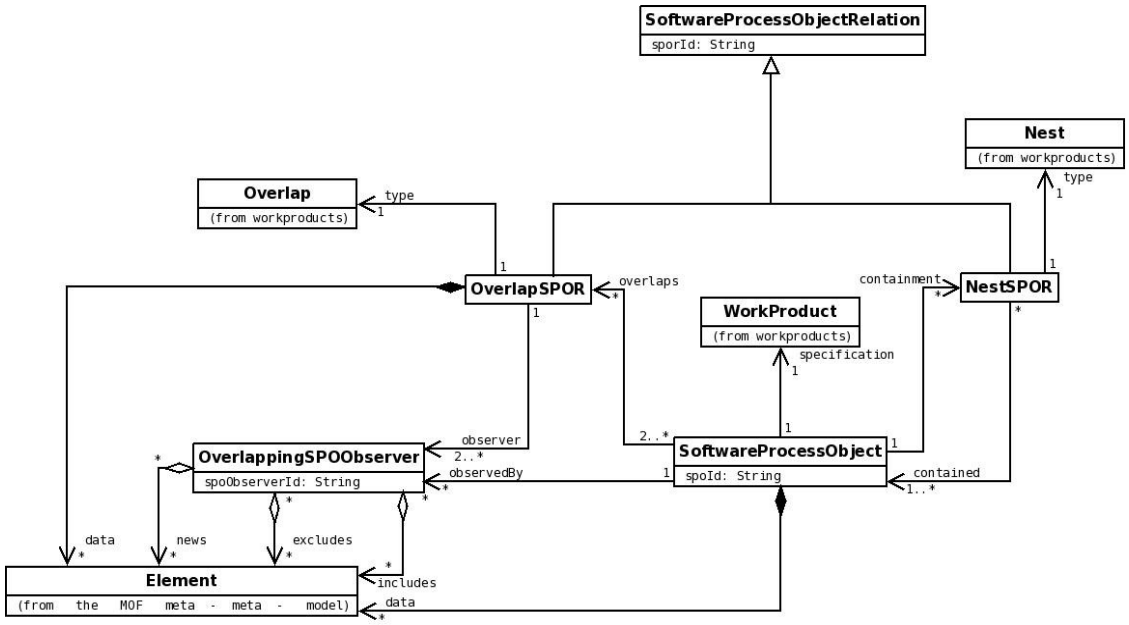


Figure 4.8: Le *Méta-Modèle* de SPOs

4.5.1 SoftwareProcessObject

4.5.1.1 Description et sémantique

Le concept de **SoftwareProcessObject** (**SPO**) est utilisé dans le *méta-modèle* pour représenter les *SPOs* d'un *procédé* durant l'exécution. Les **SPOs** sont donc gérés par le *moteur d'exécution* et réifient des *produits-modèles* ou parties de *produits-modèles*. Ils incluent également des informations sur les relations qui existent entre ces *derniers*. Pour une meilleure structuration du *méta-modèle*, deux concepts différents sont proposés pour représenter les *produits-modèles* d'une part, et leurs relations d'autre part. Ainsi, le concept de **SPO** est spécifique aux *produits-modèles* eux-mêmes, alors que celui de **SoftwareProcessObjectRelation (SPOR)** est relatif aux relations entre *produits-modèles*. Un **SPO** est caractérisé par son identifiant permettant au moteur chargé d'exécuter le procédé de l'identifier mais aussi par sa spécification qui est une référence vers un *WorkProduct* du *modèle de procédé* en cours d'exécution. Cette spécification permet au moteur de vérifier l'intégrité du **SPO**, notamment relativement au type du *produit-modèle* représenté. L'attribut

data permet de stocker les éléments de modèle du *produit-modèle* auquel correspond le **SPO**. Il est également possible pour chaque **SPO** de déterminer les relations dans lesquelles il est impliqué à travers les références ***overlaps*** et ***containments*** qui pointent respectivement vers des **OverlapSPOR** et **NestSPOR**. On peut, enfin, atteindre les **SPOObservers** qui concernent le **SPO** à travers la référence ***observers***. En effet, chaque fois qu'un **SPO** est associé à un **OverlapSPOR**, on lui associe également un **SPOObserver** qui est un « observateur » à travers lequel on peut stocker les évènements qui se produisent sur les éléments de modèles enregistrés dans l'**OverlapSPOR** (donc communs à plusieurs **SPOs).** C'est ainsi que de tels événements pourront être notifiés au développeur responsable du *produit-modèle* réifié par le **SPO**. Les concepts de **OverlapSPOR** et de **NestSPOR** décrivent les réifications des relations d'**Overlap** et de **Nest** (respectivement). Ces différents concepts sont décrits dans les paragraphes suivants.

4.5.1.2 Généralisations

Aucune généralisation définie pour le moment.

4.5.1.3 Attributs

spoId: String

Un identifiant unique du **SPO** durant l'exécution du *procédé*. Cet identifiant est utilisé par le *moteur d'exécution* du *procédé*.

4.5.1.4 Associations

spécification: WorkProduct[1]

Permet d'obtenir la spécification du **SPO** qui est donnée sous la forme d'un *WorkProduct* défini dans le *modèle de procédé* en exécution. Cette spécification est utilisée par le *moteur d'exécution* du procédé pour déterminer les caractéristiques intrinsèques du **SPO** (son type, ses relations, etc.).

data: Element[*]

Une structure de donnée permettant de stocker les éléments de modèles du *produit-modèle* correspondant au **SPO**. C'est elle qui stocke tous les éléments de modèles si le **SPO** n'est associé à aucun **OverlapSPOR**. Autrement, seuls les

éléments de modèles instances de méta-classes qui ne font partie d'aucune spécification d'aucun **Overlap** impliquant le **SPO** y sont stockés. Il est important de noter que le format (le type) des éléments de *produit-modèles* dépend de la représentation utilisée au sein des outils du *procédé*. Il peut ainsi s'agir d'éléments XML, ou XMI [OMG-XMI 2005], de séquences d'opérations, etc. Il existe, en effet, plusieurs approches qui préconisent la représentation d'un *modèle* par la liste des opérations de création, modification, suppression des éléments qui le composent. Ces approches connaissent une poussée fulgurante et ont donné naissance à plusieurs résultats présentés notamment dans [Herrmannsdoerfer & Koegel 2010]. Nous allons reparler de ces approches puisque le prototype que nous avons développé et présenté au chapitre suivant est basé sur une d'entre elles.

overlaps: OverlapSPOR [*]

Permet d'obtenir les **OverlapSPORs** qui concernent le **SPO**.

containment: NestSPOR [*]

Permet d'obtenir les **NestSPORs** qui concernent le **SPO**.

observedBy: SPOObserver [*]

Permet d'obtenir les **SPOObservers** qui concernent le **SPO**. Comme précisé plus haut, le concept de **SPOObserver** est décrit dans la suite.

4.5.1.5 Contraintes

- Si un **SPO** est associé à un **OverlapSPOR** donné, alors il est également associé à un **SPOObserver** qui à son tour est associé au même **OverlapSPOR**.
- La collection ***data*** d'un **SPO** ne peut contenir aucun élément de modèle dont la méta-classe n'est pas représentée dans le ***type*** du **WorkProduct** associé au **SPO**.

Ci-après, les descriptions en OCL de ces contraintes (respectivement).

```
O context SoftwareProcessObject inv :
  if self.overlaps->notEmpty() then
    self.overlaps->forAll(ov|ov.observer->forAll(obs|
    obs.overlapSPOR=ov implies
    obs.softwareProcessObject=self))
  endif
```

```
O context SoftwareProcessObject inv :
  self.data->forAll(me: Element |
  self.specification.type.usedModelTypeElement->
  includes(me.class))
```

4.5.1.6 Notation

Aucune notation définie pour le moment.

4.5.2 SoftwareProcessObjectRelation

4.5.2.1 Description et sémantique

Les relations entre *produits-modèles* sont modélisées à travers les concepts **Nest** et **Overlap** (cf section 4.3.1). Cette modélisation, de niveau conceptuel, est une description abstraite de ces relations. Lors de l'exécution du *procédé* par contre, des informations (éléments de modèle, etc.) relatives à ces relations sont créées et/ou modifiées au fur et mesure et ont ainsi besoin d'être stockées et gérées par le *moteur d'exécution* qui va aussi les utiliser dans sa gestion des *produits-modèles*. Par exemple, lors de la modélisation d'un *procédé*, on peut définir un **Overlap** entre deux ou plusieurs **WorkProducts** en indiquant sa nature (totale ou partielle) et sa spécification (sous la forme d'un ensemble de méta-classes). Nous avons vu précédemment à la section 4.4 que nous avons fait le choix de stocker de façon unique dans une entité distincte des SPOs associés à ces *produits-modèles* les éléments de modèles communs.

Une telle structure correspond alors à une réification de l'**Overlap** modélisé.

Nous définissons le concept de **SoftwareProcessObjectRelation** (**SPOR**) pour représenter les entités réifiant les relations. Il est caractérisé par son identifiant qui est une chaine de caractère. Ce concept est défini dans le seul but d'ensuite être spécialisé sous forme de **OverlapSPOR** et de **NestSPOR**.

4.5.2.2 Généralisations

Aucune généralisation définie pour le moment.

4.5.2.3 Attributs

sporId: String

Un identifiant unique du **SPOR** durant l'exécution du *procédé*. Cet identifiant est utilisé par le *moteur d'exécution* du procédé pour pouvoir accéder, au besoin, aux données (des éléments de modèles par exemple) qu'il contient.

4.5.2.4 Associations

Aucune association n'est définie pour le moment.

4.5.2.5 Contraintes

Aucune contrainte n'est définie pour le moment.

4.5.2.6 Notation

Aucune notation définie pour le moment.

4.5.3 OverlapSPOR

4.5.3.1 Description et sémantique

Le concept de **OverlapSPOR** est utilisé pour représenter les relations entre **SPO**. Un **OverlapSPOR** a pour caractéristiques son identifiant, son type et son champ ***data***. L'identifiant est hérité de la méta-classe plus générique **SoftwareProcessObjectRelation**. Le type correspond à une référence vers l'**Overlap** spécifiant l'**OverlapSPOR** dans le *modèle de procédé*. Le champ ***data*** d'un **OverlapSPOR** contient les objets correspondant aux éléments de modèles communs aux *produits-modèles* en relation. En effet, un **OverlapSPOR** est créé par le *moteur d'exécution* pour chaque relation d'**Overlap** définie entre deux ou plusieurs *WorkProducts* d'un *modèle de procédé.* C'est ensuite le champ ***data*** de cet **OverlapSPOR** qui est utilisé pour stocker l'ensemble des éléments de modèles communs aux *produits-modèles* liés. Ces *produits-modèles* sont ceux associés aux **SPOs** impliqués dans la relation. Les éléments de modèles communs sont des instances de méta-classes (**ModelTypeElements**) contenus dans le **ModelType** spécifiant l'overlap. C'est ainsi qu'un seul exemplaire de chacun de ces éléments de modèles communs sera stocké dans la base de modèles (le repository). Des

références sont alors utilisées dans les **SPOs** liés pour les remplacer. Pour rappel, nous avons indiqué plus haut (section 4.5.1.3) que le format des données stockées au niveau du champ ***data*** est fonction de la représentation utilisée par les outils de développement employés au cours de l'exécution du *procédé*. C'est cette représentation qui également détermine la syntaxe des références utilisées dans les **SPOs** pour désigner des éléments partagés. En plus de permettre un stockage unique des éléments de modèles communs à des *produits-modèles*, les mécanismes mis en œuvre autour du concept d'**OverlapSPOR** offrent la possibilité d'une évolution automatique des *produits-modèles*, mais aussi d'un maintien de leur cohérence relationnelle liée au partage d'éléments de modèles. En effet, comme cela sera présenté dans la suite avec le concept de **SPOObserver** que nous avons associé à celui d'**OverlapSPOR**, un mécanisme d'alerte-réponse permettra à un développeur d'automatiquement « importer » des éléments de modèles créés dans le cadre de la construction d'un autre *produit-modèle* avec lequel une relation d'*overlap* a été définie. Également, le fait de disposer d'un élément de modèle donné en un seul exemplaire offre naturellement la garantie d'une cohérence relationnelle entre les *produits-modèles* qui le partagent.

4.5.3.2 Généralisations

La méta-classe **OverlapSPOR** est une spécialisation de la méta-classe **SoftwareProcessObjectRelation** du *méta-modèle*.

4.5.3.3 Attributs

sporId: String

Un identifiant unique de l'**OverlapSPOR** durant l'exécution du *procédé*. Cet identifiant est utilisé par le *moteur d'exécution* du procédé pour pouvoir accéder, au besoin, aux éléments de modèles qu'il contient et aux autres données caractéristiques de l'**OverlapSPOR**.

4.5.3.4 Associations

observers: SPOObserver [2..*]

Permet de désigner les **SPOObservers** qui sont associés aux **SPOs** spécifiés par les *WorkProducts* liés.

type: Overlap [1]

Correspond à l'**Overlap** du *modèle de procédé* décrivant le l'**OverlapSPOR**.

data: Element[*]

Une structure de donnée qui permet de stocker des éléments de *produits-modèles* partagés. De tels *produits-modèles* sont spécifiés par des *WorkProducts* impliqués dans un **Overlap**. De plus les éléments de modèles ne sont stockés dans ce champs que s'ils sont des instances de méta-classes appartenant au **ModelType** correspondant à la spécification de l'**Overlap**.

4.5.3.5 Contraintes

Aucune contrainte définie pour le moment.

4.5.3.6 Notation

Aucune notation définie pour le moment.

4.5.4 NestSPOR

4.5.4.1 Description et sémantique

Le concept de **NestSPOR** est utilisé pour structurer les informations relatives aux relations de composition entre *produits-modèles* d'un *procédé* en exécution. Ces informations sont tirées du *modèle de procédé* et permettent au *moteur d'exécution* de gérer et/ou d'exploiter les *produits-modèles* d'une façon modulaire. En effet, elles offrent la possibilité d'un accès à la fois à leurs différentes parties mais aussi à des entités plus globales dont ils peuvent, à leur tour, être des parties. Les informations structurées par ce concept permettent également d'assurer une intégrité sémantique des *produits-modèles* en assurant une *cohérence relationnelle* de niveau « modèle » entre eux comme nous l'avons montré dans la section xxx de ce chapitre. Un **NestSPOR** est généré pour chaque relation **Nest** du *modèle de procédé*. Il est caractérisé par son identifiant et une référence vers le *Nest* du *modèle de procédé* auquel il correspond (son ***type***). Il contient également une référence (***contained***) vers le **SPO** composant de la relation. C'est donc à travers le concept de **NestSPOR**, dont le rôle est la mise en œuvre concrète du concept de **Nest** proposé dans le premier *méta-modèle*, qu'une

réponse est donnée au problème de la non flexibilité relative à la *granularité dèles* soulevé dans le chapitre 2.

4.5.4.2 Généralisations

La méta-classe **NestSPOR** est une spécialisation de la méta-classe **SoftwareProcessObjectRelation** du *méta-modèle*.

4.5.4.3 Attributs

sporId: String

Un identifiant unique du **NestSPOR** durant l'exécution du *procédé*. Cet identifiant est utilisé par le *moteur d'exécution* du procédé pour pouvoir accéder, au besoin, aux autres caractéristiques du **NestSPOR** ainsi qu'aux **SPOs** impliqués dans la relation à laquelle il correspond.

4.5.4.4 Associations

container: SoftwareProcessObject [1..*]

Permet de désigner le ou les **SPO(s)** composé(s) de l'association.

contained: SoftwareProcessObject [1..*]

Permet de désigner le ou les **SPO**(s) composant(s) de l'association.

type: Nest [1]

Correspond au **Nest** du *modèle de procédé* décrivant le **NestSPOR**.

4.5.4.5 Contraintes

Aucune contrainte définie pour le moment.

4.5.4.6 Notation

Aucune notation définie pour le moment.

4.5.5 SPOObserver

4.5.5.1 Description et sémantique

On a vu dans les précédentes sections que des entités de type **OverlapSPOR** sont utilisées par le *moteur d'exécution* pour stocker les données correspondant aux éléments communs entre plusieurs **SPOs**. Ces données stockées de façon unique font ainsi partie intégrante des **SPOs** alors en relation. Nous avons également parlé au cours de l'exemple (section 4.4) de l'enregistrement des actions réalisées sur les données communes au cours de l'exécution du *procédé*. Nous avions également précisé que ces actions interviennent au cours de la manipulation des *produits-modèles* par les développeurs, qu'elles correspondent à des ajouts, modifications et suppressions d'éléments de *modèles*, et qu'elles étaient enregistrées avec l'aide de structures appelées « observateurs ». Les entités de type **SPOObserver** sont utilisées pour représenter ces observateurs. Elles sont caractérisées par un nom (ID) utilisé pour les identifier, mais aussi par trois attributs: ***news***, ***includes***, et ***excludes***. Le champ ***news*** d'un **SPOObserver** donné est utilisé pour enregistrer les actions des développeurs sur les données communes si ces actions sont réalisées dans le cadre de la manipulation d'un autre **SPO** que celui associé au **SPOObserver**. Un développeur souhaitant alors travailler sur un **SPO** est informé (à l'aide du même champ ***news*** du **SPOObserver** associé au **SPO**) des modifications sur les données communes. Il a donc la possibilité d'accepter ou de refuser ces mises à jour selon les éventuelles contraintes du *procédé*. Dans le cas d'une acceptation, les actions correspondantes sont copiées dans le champ ***includes*** du même **SPOObserver**, elles sont copiées dans son champ ***excludes*** sinon. Dans tous les cas le champ ***news*** du **SPOObserver** est vidé de telles actions. Lorsqu'une nouvelle action est effectuée sur les données communes à des **SPO** donnés, elle est stockée directement dans le champ ***includes*** du **SPOObserver** associé, puis dans les champs ***news*** des **SPOObservers** associés à tous les autres **SPOs** liés.

4.5.5.2 Généralisations

Aucune généralisation n'est définie pour le moment.

4.5.5.3 Attributs

spoObserverId: String

Un identifiant unique du **SPOObserver** durant l'exécution du *procédé*. Cet identifiant est utilisé par le *moteur d'exécution* du procédé.

4.5.5.4 Associations

news: Element[*]

Permet de stocker de façon temporaire des actions nouvellement exécutées sur des éléments de modèles stockés dans un **SPORelation**. Les éléments de modèles en question sont donc partagés entre deux ou plusieurs *produits-modèles* dont celui qui correspond au **SPO** associé au **SPOObserver**. Les actions peuvent être des actions de création, de modification ou de suppression.

Includes: Element[*]

Permet de stocker des actions exécutées sur des éléments de modèles stockés dans un **SPORelation**. Les éléments de modèles en question sont partagés entre deux ou plusieurs *produits-modèles* dont celui qui correspond au **SPO** associé au **SPOObserver**. Les actions peuvent être des actions de création, de modification ou de suppression et sont acceptées par le développeur responsable du *produit-modèle*.

excludes: Element[*]

Permet de stocker des actions exécutées sur des éléments de modèles stockés dans un **SPORelation**. Les éléments de modèles en question sont partagés entre deux ou plusieurs *produits-modèles* dont celui qui correspond au **SPO** associé au **SPOObserver**. Les actions peuvent être des actions de création, de modification ou de suppression et sont rejetées par le développeur responsable du *produit-modèle*.

4.5.5.5 Contraintes

Aucune contrainte n'est définie pour le moment.

4.5.5.6 Notation

Aucune notation définie pour le moment.

Cette section était consacrée au *méta-modèle* des *SPOs* effectivement manipulés par un *moteur d'exécution*. Dans ce *méta-modèle*, le concept de **SoftwareProcessObject** est utilisé pour décrire les *produit-modèles* (ou parties de *produits-modèles*) d'un *procédé* en exécution.

Il contient aussi les méta-classes **OverlapSPOR** et **NestSPOR** qui correspondent aux projections respectives dans l'espace d'exécution des deux principales relations (**Overlap** et **Nest**) qui peuvent être modélisées entre les *WorkProducts*. Enfin, le *méta-modèle* définit le concept de **SPOObserver** qui est associé à celui de **OverlapSPOR** pour permettre au *moteur d'exécution* de gérer les actions effectuées sur les éléments communs à plusieurs *produits-modèles* du *procédé* au cours de son exécution.

A ce stade, nous avons présenté deux *méta-modèles*. Le premier (section 4.2) structure les *WorkProducts*, et le deuxième (section 4.5) les *SPOs* d'un *procédé*. La section suivante est ainsi consacrée à la définition des règles de correspondance sur la base desquelles les entités correspondant aux données utilisées à l'exécution d'un procédé sont générées à partir de celle définies à la modélisation.

4.6 La Création des SPOs

Le *moteur d'exécution* prend en entrée un *modèle de procédé* au sein duquel les *produits-modèles* sont décrits avec l'aide des concepts du *méta-modèle* de *WorkProducts* présenté dans la section 4.3. Pour tout *modèle de procédé*, le *moteur d'exécution* commence par la création, d'un ensemble d'instances des concepts du *méta-modèle* de SPOs. Ces instances correspondent aux SPOs à travers lesquels les *produits-modèles* seront gérés au fur et à mesure de l'exécution du procédé. Dans cette section, nous présentons la façon dont les **SPOs** sont générés (à partir des éléments du modèle de procédé) à travers une définition des règles de correspondance définies à cet effet.

L'exécution d'un *modèle de procédé* par un *moteur d'exécution* peut être découpée en différentes phases. Nous utilisons les termes d' « **initialisation** » et de « **déroulement** » pour désigner les deux phases que nous considérons comme les plus importantes dans notre cas présent. Ainsi, dans la suite de cette section, nous appelons **phase d'initialisation** la phase au cours de laquelle le *moteur d'exécution* lit un modèle de procédé pour créer un contexte d'exécution correspondant aux spécifications qu'il contient. La **phase de déroulement** désigne alors l'étape au cours de laquelle les activités du procédé sont réalisées.

Lors de la phase d'initialisation, tous les **SPOs** du procédé sont créés sous la forme d'instances des méta-classes **SoftwareProcessObject**, **SoftwareProcessObjectRelation** et **SPOObserver**. A cette étape, tous les **SPOs** sont vides; en effet, les champs ***data*** (des **SPOs** et **OverlapSPORs)**, ***news***, ***includes*** et ***excludes*** (des **SPOObservers**) contiennent tous des valeurs nulles. Le **Tableau 4.1** définit les correspondances entre les concepts de modélisation et ceux d'exécution. Ce sont ces correspondances sur la base desquelles les **SPOs** sont créés, tel que, par exemple, cela est illustré à travers l'exemple de la section 4.4 de ce chapitre. Elles sont explicitées par les règles listées ci-dessous.

Règle R1. Une entité de type **SoftwareProcessObject** est créée pour chaque **WorkProduct** du *modèle de procédé*. Le ***spoId*** du **SPO** est généré par le *moteur d'exécution* et son champ ***specification*** initialisé à la référence vers le **WorkProduct**.

Règle R2. Un **OverlapSPOR** est créé pour chaque **Overlap** du *modèle de procédé*:

- son ***sporId*** est généré par le *moteur d'exécution*, et son ***type*** initialisé à la référence vers l'**Overlap**,
- pour chaque **SPO** dont le **WorkProduct** correspondant fait partie de la collection ***overlapingWorkProducts*** de l'**Overlap**:
  - l'**OverlapSPOR** est ajouté à la collection ***overlaps*** du **SPO**,
  - un **SPOObserver** est créé et est ajouté aux collections ***observedBy*** et ***observer*** du **SPO** et du **OverlapSPOR**, respectivement.

Règle R3. Un **NestSPOR** est créé pour chaque **Nest** du *modèle de procédé*:

- **s**on ***sporId*** est généré par le *moteur d'exécution* et son ***type*** initialisé à la référence vers le **Nest**,
- sa référence ***container*** est initialisée au **SPO** correspondant au ***nester*** **WorkProduct** du **Nest** dont le ***containement*** est initialisé au Nest**SPOR**,

- sa référence ***contained*** est initialisée au **SPO** correspondant au ***nested*** **WorkProduct** du **Nest**.

Règles	Concepts de modélisation	Concepts d'exécution
R1	WorkProduct	SoftwareProcessObject
		-spoId
	-workProductId	-specification
		-data
R2	Overlap	OverlapSPOR
		-sporId
		-data
	-overlapId	-type
	-overlapingWorkProduct	SPO.overlaps
		SPOObserver
		OverlapSPOR.observer
		SPO.observedBy
R3	Nest	NestSPOR
		-sporId
	-nestId	-type
	-nested	-contained
	-nester	container
	WorkProduct.nest	SPO.containment

Tableau 4.1: Correspondances entre Concepts de Modélisation et Concepts d'Exécution

4.7 Conclusion

Dans ce chapitre nous avons présenté deux *méta-modèles*. Le premier correspond à celui des *WorkProducts* de *procédé* et introduit principalement le concept de **WorkProduct** pour représenter la spécification des *produits-modèles* d'un procédé dans MDE. Dans ce *méta-modèle*, un **WorkProduct** est associée un **ModelType** qui représente son type; nous utilisons en effet un typage de modèle basé essentiellement sur les travaux présentés dans [Jim Steel & Jézéquel 2007] et qui définit un type comme un ensemble de méta-classes considérées à travers l'introduction du concept de **ModelTypeElement**. Enfin, le *méta-modèle* définit des concepts permettant de spécifier les relations entre *produits-modèles*. Les concepts d'**Overlap** et de **Nest** sont ceux supportés pour le moment. Ils représentent, respectivement, les relations de partage d'éléments entre *modèles* et de composition de *modèles*. Le *méta-modèle* de

WorkProducts est ensuite utilisé, à des fin d'illustration, pour modéliser le procédé-exemple introduit depuis le chapitre 2.

Le deuxième *méta-modèle* est celui des produits tels que manipulés par le *moteur d'exécution* en charge de l'exécution du procédé. Il définit le concept de **SoftwareProcessObject** qui correspond à un *produit-modèle* du procédé et contient aussi les méta-classes **OverlapSPOR** et **NestSPOR** qui correspondent aux projections respectives des relations **Overlap** et **Nest** précédemment vues et actuellement supportées dans nos travaux. Enfin, ce deuxième *méta-modèle* introduit le concept de **SPOObserver** qui complète celui de **OverlapSPOR** en offrant la possibilité de la gestion des actions effectuées sur les données partagées entre **SPOs**.

Chacun des concepts définis dans ces deux *méta-modèles* est décrit dans cette section selon un style employé pour la description des concepts de UML2.

Le chapitre présente également des règles de correspondance entre les concepts de modélisation et ceux d'exécution.

Ces éléments correspondent à l'épine dorsale de la contribution de cette thèse. Dans le chapitre qui va suivre, nous présentons un prototype que nous avons développé pour expérimenter et outiller notre proposition.

Chapitre 5

Outillage de l'Approche

5.1 Introduction

Dans le chapitre précédent, nous avons présenté notre contribution. Du point de vue de la modélisation de *procédés*, elle consiste en la proposition d'un *méta-modèle* pour les *WorkProducts*. Ce dernier introduit deux types de relations qui sont le *Nest* et l'*Overlap* et qui permettent de spécifier, respectivement, l'inclusion entre *produit-modèles* ainsi que le partage d'éléments de *produit-modèle*. Du point de vue de l'exécution de *procédés*, la contribution correspond à la proposition d'un *méta-modèle* pour les **SPOs** à travers lesquels les *produits-modèles* et les données relatives aux relations entre eux sont gérés par un *moteur d'exécution*. La proposition est complétée par la définition de règles de correspondance entre les concepts des deux *méta-modèles* et ainsi de génération des entités d'exécution (i.e., **SPOs**) à partir de celles de modélisation (i.e., **WorkProducts**). Nous avons également montré que notre proposition offrait le support d'une gestion modulaire des *produits-modèles*. Finalement à travers un stockage centralisé des éléments de modèles partagés entre plusieurs *produits-modèles*, notre solution assure une forme de *cohérence relationnelle* de façon systématique ainsi qu'une évolution automatique et dynamique des *produits-modèles*. Dans le but d'illustrer la faisabilité de la solution que nous avons proposée, nous présentons dans ce chapitre le prototype que nous avons développé. Il « simule » un environnement de modélisation et d'exécution de *modèles de procédés* tenant compte de notre proposition. Il est utilisé pour la définition et l'exécution d'un *modèle de procédé* représentant le procédé-exemple que nous avons introduit au chapitre 2.

Nous présenterons d'abord succinctement l'objectif visé dans ce chapitre de la thèse. Puis nous détaillerons l'architecture du prototype développé. Nous présenterons ensuite les étapes de modélisation et d'exécution d'un *procédé* en nous appuyant sur le procédé-exemple pour illustrer le fonctionnement du prototype. Il s'ensuivra une présentation des principaux outils (EMF et Praxis) qui ont aidé à sa réalisation. Certaines étapes de la réalisation ainsi que leur résultat seront ensuite présentés avant la conclusion de ce chapitre.

5.2 Les objectifs du chapitre

Dans ce chapitre, nous cherchons à atteindre deux objectifs principaux. Le premier est de présenter un outillage de notre approche. Le second objectif est de décrire l'utilisation de cette approche dans un exemple.

En ce qui concerne l'outillage, nous avons développé un prototype de *PSEE*. Ledit *PSEE* comprend un environnement de modélisation et un autre d'exécution de *procédés*. L'environnement de modélisation supporte les concepts de modélisation définis dans le *méta-modèle* de *WorkProducts*. Il permet ainsi de modéliser les relations d'inclusion (i.e., *nest*) et de partage (i.e., *overlap*) d'éléments de modèles entre les futurs *produits-modèles* d'un *procédé*. Cet environnement supporte également la modélisation des autres éléments d'un *procédé* comme les activités, rôles, etc. A cet effet, nous avons choisi d'étendre le *méta-modèle* de *WorkProduct* auquel nous avons ajouté les concepts nécessaires (cf section 5.3.1.1). L'environnement d'exécution définit quant à lui un *moteur d'exécution* de *procédé*. Il supporte la possibilité de lire des spécifications de *produits-modèles* sous la forme (d'instances) de **WorkProducts** ainsi que des relations qui les lient. Le *moteur d'exécution* prend ensuite en compte ces relations afin d'assurer une modularité des *produits-modèles* spécifiés, leur évolution semi-automatique, voire automatique, ainsi que le maintien de leur *cohérence relationnelle*.

En ce qui concerne le second objectif, tout au long de la présentation du fonctionnement du *PSEE*, nous présenterons les différentes étapes de la modélisation et de l'exécution du procédé-exemple introduit depuis chapitre 2.

Dans la section suivante, nous présentons le prototype développé. Au fur et à mesure de cette présentation, nous montrons également l'utilisation de ses composants pour modéliser et

exécuter le procédé-exemple.

5.3 Présentation du *PSEE*

Pour l'outillage de notre approche, nous avons conçu le *PSEE* présenté dans ce chapitre. Il permet donc la définition ainsi que l'exécution de *modèles de procédés*. Au sein de ces *modèles de procédés*, les *produits-modèles* sont décrits en utilisant les concepts introduits par le *méta-modèle* de *WorkProducts* présenté au chapitre 4. Lors de l'exécution d'un *modèle de procédé* à l'aide du *PSEE*, les relations modélisées entre les *produits-modèles* sont prises en compte dans la gestion de ces derniers. Nous présentons dans cette section l'architecture du *PSEE* ainsi que son fonctionnement illustré au fur et à mesure.

5.3.1 Architecture détaillée du PSEE

Le *PSEE* prototypé comprend trois principaux composants: le composant de modélisation, celui d'exécution, et celui de stockage (voir **Figure 5.1**). Nous présentons ci-après chacun d'eux.

5.3.1.1 Le composant de modélisation (*ProcessModeling*)

Le composant *ProcessModeling* correspond à l'environnement de modélisation de *procédé*. Il permet d'utiliser les concepts définis par le *méta-modèle* de *WorkProducts* pour construire des *modèles de* procédés prenant en compte les relations entre *produits-modèles* actuellement supportées par notre proposition. Dans le but de pouvoir représenter les autres éléments d'un *procédé* (activités, rôles, etc.) dans cet environnement, nous avons ajouté au *méta-modèle* de *WorkProduct* les concepts nécessaires à leur modélisation. Plus précisément, nous avons fait ressortir les éléments suivants.

- Un *procédé* est un ensemble d'activités.
- Une activité peut être être simple ou composée d'autres activités.
- Une activité prend en entrée des *produits-modèles* et en fournit d'autres en sortie.
- Une activité est sous la responsabilité d'un ou de plusieurs rôles.

C'est donc sur la base du *méta-modèle* élargi que le composant *ProcessModeling* est construit.

Il faut rappeler que notre proposition est générique ; en effet, nous considérons l'approche de modélisation et de gestion de *produits-modèles* qu'elle définie comme une extension adaptée à chacune des approches de gestion *procédés* existantes et que nous avons jugées limitées du point de vue de la gestion de *produits-modèles*. Notre étude ne s'est ainsi pas focalisée sur les éléments de *modèles de procédés* autres que les *produits*. C'est ainsi que nous avons choisi de baser l'extension du *méta-modèle* de *WorkProduct* pour lui permettre le support de ces autres éléments sur l'étude que nous avons faite de la littérature sur la gestion de *procédé* en général et sur la modélisation de ces *procédés* en particulier et que nous avons présentée au chapitre 3. En effet, comme précisé dans [Scacchi 2001], il faut noter que, de façon générale, les approches considérées dans cette étude modélisent un *procédé* comme un ensemble d'activités qui consomment, produisent et modifient des *produits* et impliquent des agents (humains ou non) qui peuvent se servir d'outils pour jouer leurs rôles respectifs dans le *procédé*. 5.3.1.2 Le composant d'exécution (*ProcessExecution*)

Le composant *ProcessExecution* correspond au *moteur d'exécution*. Il s'occupe de l'exécution des différentes activités d'un *procédé* en prenant en compte les éléments relatifs aux relations entre *produits-modèles*. Il se compose de deux sous-composants: le composant d'initialisation et celui d'exécution d'activité.

5.3.1.2.1 Le composant d'initialisation (*ProcessInitialisation*)

Ce composant s'occupe de l'initialisation de l'exécution d'un *procédé*. Celle-ci a lieu lors de la première phase de l'exécution appelée phase d'initialisation au cours de laquelle le *moteur d'exécution* lit le *modèle de procédé* et crée les SPOs qui correspondent aux *WorkProducts* qu'il contient. Nous décrirons cette phase dans la suite (section 5.3.3.1) avec plus de détails.

5.3.1.2.2 Le composant d'exécution d'activité

C'est le composant responsable de l'exécution des activités d'un *procédé*. Il comprend à son tour les sous-composants de pré-exécution, de post-exécution et de travail décrits ci-après.

5.3.1.2.2.1 Le composant de pré-exécution (PreExecution)

C'est le composant qui s'occupe du traitement initial au cours de l'exécution d'une activité. Il identifie les *produits-modèles* nécessaires à l'activité, génère les **SPOs** correspondants dans

l'espace de stockage de **SPOs** selon les règles de correspondances définies et les met à la disposition du workspace du développeur.

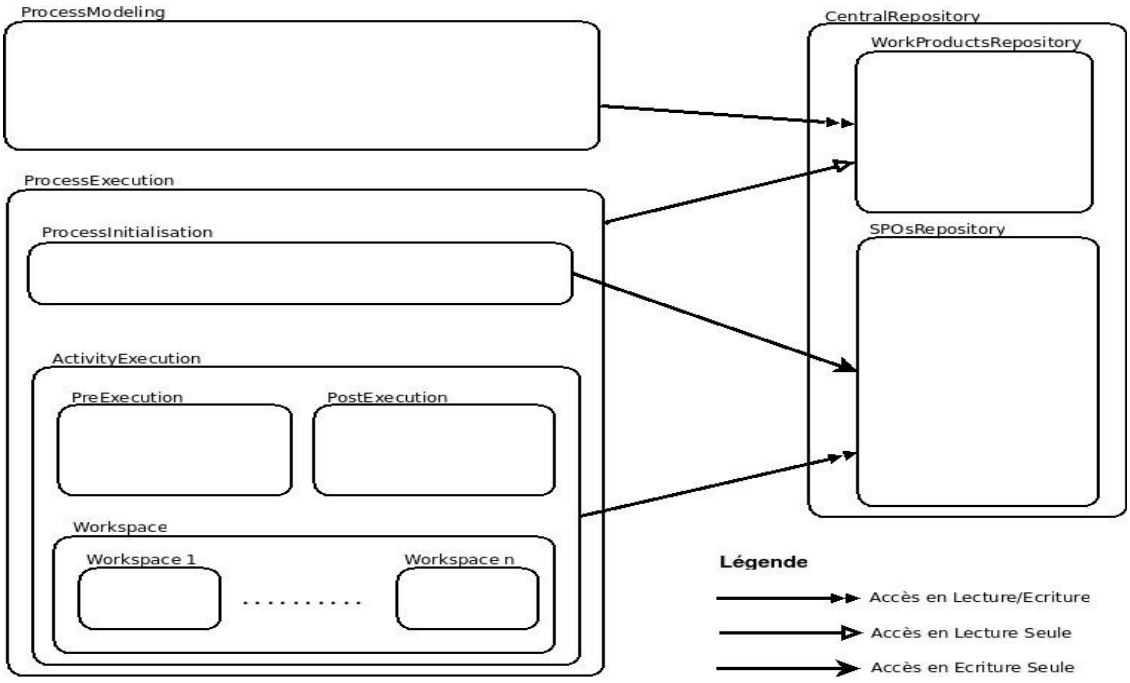


Figure 5.1: Architecture du PSEE Prototypé

5.3.1.2.2.2 Le composant de post-exécution (PostExecution)

C'est le composant responsable du traitement réalisé à la fin d'une activité. Il est chargé d'envoyer au composant de stockage les éléments de modèles des *produits-modèles* crées ou modifiés par l'activité.

5.3.1.2.2.3 Le composant travail (Workspace)

Il gère les différents espaces de travail des développeurs. Un espace de travail correspond à l'environnement permettant à un développeur de travailler en local en utilisant ses outils favoris.

5.3.1.3 Le composant de stockage (*CentralRepository*)

Le composant *CentralRepository* correspond au repository de stockage. Il comprend à son tour deux sous-composants spécialisés respectivement dans le stockage des éléments de

modèle de procédé et dans celui des **SPOs**. Nous présentons chacun des ces deux sous-composants ci-après.

5.3.1.3.1 Le composant de stockage de *WorkProducts* (*WorkProductsRepository*)

Il fournit les fonctionnalités de stockage et de restitution de *modèles de procédés*. C'est dans le souci d'avoir une meilleure modularité que ce module est séparé de celui du stockage des **SPOs**.

5.3.1.3.2 Le composant de stockage de SPOs (*SPOsRepository*)

Le composant de stockage des **SPOs** est chargé de gérer le stockage et l'accès des différents développeurs aux *produits-modèles* sur lesquels ils travaillent. Ces *produits-modèles* sont stockés sous forme de **SPOs**. Avec les sous-composants *Workspace*, *PreExecution* et *PostExecution* du composant *ActivityExecution*, il implémente une architecture de repository dite centralisée ou encore appelée architecture workspace-repository [Altmanninger et al. 2009]. Cette architecture distingue un référentiel central qui gère le stockage des *produits* manipulés par les développeurs dans leurs espaces de travail respectifs et répartis dans le réseau.

5.3.1.4 Les accès au composant de stockage

Les composants de modélisation et d'exécution travaillent sur des données (*modèle de procédé* et/ou *produits-modèles*) dont les éléments constitutifs sont stockées dans les composants de stockage. Ceux-ci fournissent donc des interfaces à travers lesquelles les **WorkProducts** ainsi que **SPOs** qu'ils décrivent sont accessibles en lecture et/ou écriture. Sur la **Figure 5.1** nous avons utilisé des flèches pour représenter les différents accès en lecture et/ou écriture d'éléments de modèle possibles entre les différents composants.

Pour conclure cette section, nous dirons que le *PSEE* prototypé se compose essentiellement d'un éditeur de *modèle de procédé* et d'un *moteur d'exécution*. Ces deux environnements exploitent les services d'un repository de stockage tel que présenté dans l'architecture de la section précédente. Dans la suite, nous présentons le fonctionnement de l'éditeur de modèle ainsi que celui du *moteur d'exécution*. Par souci de clarté, nous utilisons le procédé-exemple

pour illustrer les étapes qui le nécessitent.

5.3.2 La modélisation de procédé

Le composant de modélisation comporte principalement un éditeur de *modèle de procédé*. C'est cet éditeur qui permet de modéliser un *procédé* tout en offrant la possibilité de spécifier les relations qui existent entre ses *produits-modèles*. Les éléments de *modèle de procédé* sont stockés dans le repository de *WorkProducts*. Nous reviendrons sur le fonctionnement de ce composant en présentant sa réalisation en section 5.5.1.
Dans la sous-section suivante, nous présentons le fonctionnement du *moteur d'exécution* qui correspond, du point de vue de notre prototypage à la partie la plus importante du *PSEE*.

5.3.3 L'exécution de procédé

L'exécution d'un *modèle de procédé* par le *moteur d'exécution* se découpe en différentes phases. Nous nous sommes intéressés, comme précisé dans le chapitre précédent, à celles présentées ci-après.

5.3.3.1 La phase d'initialisation

Sa gestion est placée sous la responsabilité du composant d'initialisation. C'est la phase au cours de laquelle le *moteur d'exécution* lit un *modèle de procédé* pour créer un contexte d'exécution correspondant aux spécifications qu'il contient. C'est au cours de cette phase que les **SPOs** (instances des concepts du *méta-modèle* de **SoftwareProcessObjects**) sont créés. En effet, le moteur applique les règles définies dans le chapitre précédent (section 4.6), dans l'ordre **Règle 1, Règle 2**, puis **Règle 3**. Pour rappel, à cette étape, tous les **SPOs** (**SPOs** et **SPORs**) créés sont vides, c'est à dire qu'ils ne contiennent aucun élément de *produit-modèle*. Ils correspondent donc à des réceptacles pour les résultats des futures activités des développeurs. Une fois tous les **SPOs** vides créés, le déroulement des activités du *procédé* peut commencer.

En guise d'illustration de cette étape, nous présentons ci-après son déroulement sur le procédé-exemple.
Lors de la phase d'initialisation de l'exécution du procédé-exemple, le *modèle de procédé*

présenté à la **Figure 4.2** (voir chapitre 2) est lu et les **SPOs** qui lui correspondent sont générés. C'est ainsi que, comme décrit au chapitre 4 (section 4.4.2), et représenté par la **Figure 4.3** (voir chapitre 4), sont créés les **SPOs** suivants:

- *AnalysisModelSPO*, *DesignModelSPO*, et *UseCaseModelSPO,*
- *NestXSPOR* et *OverlapXSPOR,*
- *Observer_AnalysisModelSPO_OverlapXSPOR* et *Observer_DesignModelSPO_OverlapXSPOR*.

Tous ces **SPOs** sont vides à cette étape; leurs champs ***data***, ***includes***, ***excludes***, et ***news*** (selon le cas) ne contiennent aucun élément de modèle. Ils vont, au cours de l'étape de déroulement qui va suivre, servir à stoker les *produits-modèles* du *procédé* ainsi que les informations relatives aux relations entre-eux. Ils sont tous stockés dans le repository de **SPOs.**.

5.3.3.2 La phase de déroulement

Elle désigne l'étape au cours de laquelle les activités d'un *procédé* se déroulent. Elle suit la phase d'initialisation précédemment présentée. Elle est gérée par le composant d'exécution d'activité (*ActivityExecution*) qui lit les activités les unes après les autres et les « exécute » par le biais de ses sous-composants (*PreExecution, Workspace,* et *PostExecution*). Lorsqu'un développeur souhaite travailler dans le cadre de l'exécution d'une activité, les **SPOs** correspondant aux *produits-modèles* sur lesquels il doit travailler sont déterminés par le composant *PreExecution* qui s'appuie sur le *modèle de procédé*. Ces **SPOs**, s'ils ne sont pas vides, sont utilisés par le même composant pour créer le ou les *produit(s)-modèle(s)* qui est (sont) envoyé(s) dans l'espace de travail (workspace) local développeur. C'est à partir de cet espace de travail et de stockage local que les outils utilisés par le développeur accèdent au(x) *produit(s)-modèle(s)*.

Lorsqu'un développeur finit une activité, le (les) *produit(s)-modèle(s)* créé(s) ou modifié(s) est (sont) envoyé(s) au repository central par le biais du composant *PostExecution*. Chaque *produit-modèle* reçu au niveau du repository est traité de la façon suivante, soit *wP* le (l'instance de) **WorkProduct** du *modèle de procédé* qui modélise le *produit-modèle* considéré:

- Si *wP* n'est impliqué dans aucune relation d'**Overlap**, alors tous les éléments de modèles du *produit-modèle* sont directement enregistrés dans le **SPO** correspondant du repository.

- Sinon, le traitement suivant est réalisé pour chaque élément de modèle du *produit-modèle*:

  - Si l'élément de modèle est une instance d'une méta-classe qui fait partie de la spécification d'un **Overlap** *o1* du *modèle de procédé*, alors il est directement enregistré dans l'**OverlapSPOR** correspondant à *o1*. Dans ce cas, si l'élément de modèle est nouvellement créé, alors l'évènement de création correspondant est enregistré dans le champ ***news*** de chaque **SPOObserver** associé à l'**OverlapSPOR** (pour rappel, un **SPOObserver** est associé à chaque **SPO** dont le **WorkProduct** correspondant est impliqué dans la relation définie par *o1*, ce même **SPOObserver** est associé l'**OverlapSPOR** modélisé par *o1*). Les autres évènements (suppression et modifications d'éléments de modèles) sont également enregistrés dans ces mêmes champs ***news***. Pour rappel, nous précisions que la détermination de la nature des actions du développeur sur les éléments de modèles du *produit-modèle* envoyé peut se réaliser suivant différentes approches comme le calcul de la différence entre l'ancienne et la nouvelle version du *produit-modèle* [Xing & Stroulia 2005]. Dans le cas de ce prototype, nous considérons les opérations effectuées sur le *produit-modèle* pour déterminer ces actions.

  - L'élément de modèle est enregistré dans le **SPO** correspondant à *wP* s'il est une instance d'une méta-classe qui ne fait partie d'aucune spécification d'un **Overlap** impliquant le *produit-modèle*.

En résumé, la phase de déroulement est composée d'un ensemble de séquences correspondant chacune à l'exécution d'une activité du *procédé*. Chacune de ces séquences comporte les trois

étapes suivantes:

- Le *pré-traitement*: c'est l'étape au cours de laquelle le *produit-modèle* sur lequel le développeur doit travailler est construit à partir des **SPOs** idoines du repository puis envoyé au workspace du développeur. Cette étape est assurée par le composant de pré exécution (*PreExecution*) comme indiqué plus haut.

- Le *traitement*: cette étape correspondant à l'activité proprement dite. A l'aide de ses outils, le développeur travaille sur le *produit-modèle* reçu sur son workspace. Cette étape est réalisée par le composant *Workspace*.

- Le *post-traitement*: c'est l'étape au cours de laquelle les *produits-modèles* sur lesquels le développeur a travaillé sont envoyés du workspace au repository dans lequel ils vont être enregistrés sous forme de **SPOs** suivant le mécanisme décrit précédemment. Cette étape est assurée par le composant de post exécution (*PostExecution*).

La phase de déroulement de l'exécution du procédé-exemple a déjà été présentée au chapitre 4, section 4.4.2.

Nous venons, dans cette section de présenter l'architecture détaillée ainsi que le fonctionnement du *PSEE* proposé pour outiller notre approche. Dans la section suivante nous allons présenter les principaux outils utilisés pour sa mise en œuvre.

5.4 Outils utilisés: EMF et Praxis

Cette section se consacre à la présentation des principaux outils utilisés pour réaliser un prototype du *PSEE* conçu et présenté dans la section précédente. C'est ainsi qu'après une brève présentation de EMF, nous allons présenter Praxis, un cadre définissant un langage de représentation de *modèles* MOF, et permettant une vérification et un maintien de règles de cohérence dans ces *modèles*.

5.4.1 EMF

EMF est l'acronyme de Eclipse Modeling Framework [EMF-Website 2011; Budinsky 2003].

C'est un cadre de modélisation pour la plate-forme de développement intégrée Eclipse [Eclipse-Website 2011] dont il est le noyau d'un projet appelé *Modeling Project*.

EMF permet la méta-modélisation à travers un langage appelé Ecore et dispose d'une fonctionnalité de génération de code à partir d'un *modèle* en plus d'un éditeur de *modèles*. C'est ainsi qu'il s'appuie sur les trois éléments principaux suivants : le langage Ecore, la bibliothèque EMF.EDIT et le modèle de génération. Nous présentons brièvement chacune de ces briques dans les sous-sections suivantes.

5.4.1.1 Le langage Ecore

Le langage Ecore est équivalent au standard MOF pour la plate-forme de modélisation et de développement Eclipse. Ainsi, il fournit un certain nombre de concepts de base pour définir les *méta-modèles*. Il représente son propre *méta-modèle* et propose aussi des interfaces propres à un *méta-modèle* donné et des interfaces réflexives permettant ainsi la manipulation des *modèles*. La **Figure 5.5** représente un sous-ensemble simplifié du langage Ecore utilisé pour décrire les *méta-modèles*.

Ecore permet de décrire des *méta-modèles* en terme de **EPackage**, **EClass**, **EAttribute**, **EDataType**, **EOperation** et **EReference**. La **Figure 5.2** représente un sous-ensemble simplifié de son *méta-modèle*.

Les *méta-modèles* construits avec l'aide de EMF sont appelés modèles Ecore ou modèles EMF. EMF offre un support de persistance pour sauvegarder ces *modèles* en XML, une API réflexive pour la manipulation des objets EMF décrits dans les *modèles* ainsi qu'un mécanisme de notification de changement.

5.4.1.2 La bibliothèque EMF.EDIT

La bibliothèque EMF.EDIT contient un ensemble de classes génériques réutilisables pour la construction des éditeurs des *modèles* décrits en Ecore. Elle fournit un ensemble de classes permettant d'afficher et d'éditer (copie, collage, etc.) les *modèles*. La bibliothèque EMF.Edit contient aussi des classes qui permettent de fournir le contenu des modèles EMF. Ce contenu est fourni sous la forme d'objets observables; ils envoient, en effet, des notifications indiquant, par exemple, la modification d'un attribut ou d'une référence. Ces notifications peuvent être

interceptées et il est possible de leur attribuer des comportements particuliers.

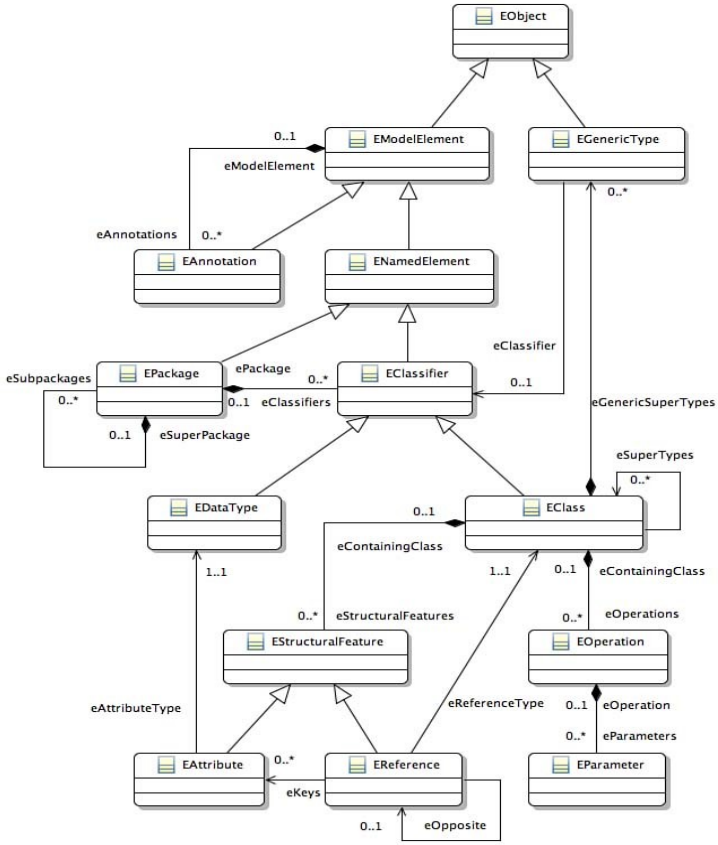


Figure 5.2: Un Sous-ensemble Simplifié du *Méta-Modèle* Ecore

5.4.1.3 Le modèle de génération

Lorsqu'un *méta-modèle* est décrit en Ecore, il est possible de générer un éditeur (un modeleur) complet qui permet de manipuler des *modèles*. La génération se fait suivant le principe suivant: Un *modèle* générateur (un fichier .genmodel) est créé à partir du modèle EMF. Il permet de configurer ce qui doit être généré, c'est à dire les packages à utiliser, la présentation des *modèles* instances du langage Ecore, l'organisation du code, le mécanisme de persistance à utiliser (XML, XMI), etc. Après la configuration du *modèle* générateur, EMF génère trois types de plug-ins qui s'appuient les uns sur les autres : le plug-in éditeur s'appuie sur le plug-in adaptateur qui, à son tour, s'appuie sur le plug-in modèle.

Notre intérêt pour la plate-forme EMF est lié au fait qu'elle nous permet de produire un ensemble de classes Java pour nos différents *méta-modèles*. EMF se base ensuite sur ces

classes pour nous offrir la possibilité de visionner et d'éditer ces *méta-modèles* ainsi que de générer des éditeurs de *modèles* représentant des instances de ces *méta-modèles*. C'est ainsi que le composant de modélisation du prototype est généré à l'aide de cette plate-forme. EMF nous a également permis de générer les APIs qui nous permettent de manipuler les *modèles de procédés* ainsi que les différents SPOs qui leur correspondent.

5.4.2 Praxis

5.4.2.1 Présentation

Praxis [Mougenot et al. 2010] qui a fait l'objet d'une thèse [Mougenot 2010] au sein de l'équipe MoVe au LIP6* est un cadre définissant un langage de représentation de *modèles* MOF. Il permet également de vérifier et de maintenir des règles de cohérence sur ces *modèles*. Praxis propose une représentation des *modèles* inspirée de l'API réflexive du MOF [OMG-MOF 2006]. A cet effet, Praxis propose l'utilisation de six actions unitaires de construction pour représenter un *modèle*. Les six actions unitaires Praxis sont listées et décrites ci-après.

- ***create(me,mc)***

Crée l'élément de modèle *me* instance de la méta-classe *mc*. Un élément de modèle peut être créé si et seulement s'il n'existe pas déjà dans le *modèle*.

- ***delete(me)***

Supprime un élément de modèle *me*. Un élément de modèle peut être supprimé si et seulement s'il existe dans le *modèle* et qu'il n'est pas référencé par un autre élément de modèle. Lorsqu'un élément de modèle est supprimé, toutes les valeurs de ses propriétés sont automatiquement supprimées.

- ***addProperty(me,p,v)***

Assigne la valeur *v* à la propriété *p* de l'élément de modèle *me.*

- ***remProperty(me,p)***

Supprime la valeur, si elle existe, de la propriété *p* de l'élément de modèle *me.*

- ***addReference(me,p,met)***

Assigne un élément de modèle cible *met* à la référence *r* de l'élément de modèle *me.*

* LIP6 – Laboratoire d'Informatique de Paris 6 - www.lip6.fr

- ***remReference(me,r)***

Supprime l'élément de modèle cible, s'il existe, de la référence *r* de l'élément de modèle *me*.

La **Figure 5.5** représente une séquence d'actions Praxis correspondant à un extrait (**Figure 5.3**) de *modèle* d'un système nommé Azureus, ce *modèle* étant une instance du *méta-modèle* UML dont un fragment simplifié est représenté par la **Figure 5.4**.

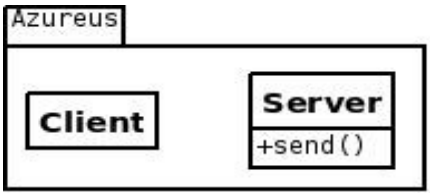


Figure 5.3: Le *Modèle* UML Azureus

Praxis fait donc partie de la famille d'approches de représentation de *modèles* MOF orientées opérations qui sont de plus en plus utilisées dans la gestion de l'évolution et de la cohérence de ces *modèles* [Herrmannsdoerfer & Koegel 2010].

De plus amples éléments peuvent être obtenus sur Praxis dans [Mougenot et al. 2010; Mougenot 2010; Xavier Blanc et al. 2008].

5.4.2.2 Motivation de l'utilisation de Praxis

Au cours de l'exécution d'un *procédé*, comme nous l'avons déjà décrit au chapitre 4 en présentant le *méta-modèle* de **SPOs**, lorsqu'un développeur modifie un *produit-modèle pm1* à partir de son workspace avant de le transmettre au repository central, si *pm1* partage des éléments de modèle avec un autre *produit-modèle pm2*, il doit y avoir une propagation automatique des modifications opérées sur *pm1* vers *pm2*. Celle-ci pouvant aussi être semi-automatique, i.e., nécessiter l'accord du développeur en charge de *pm2*. La satisfaction de cet impératif implique celle des deux exigences suivantes.

1. Il faut trouver un moyen permettant d'identifier les éléments nouvellement créés, ou modifiés, ou supprimés, du *produit-modèle pm1* et qui sont impliqués dans la relation de partage d'éléments.

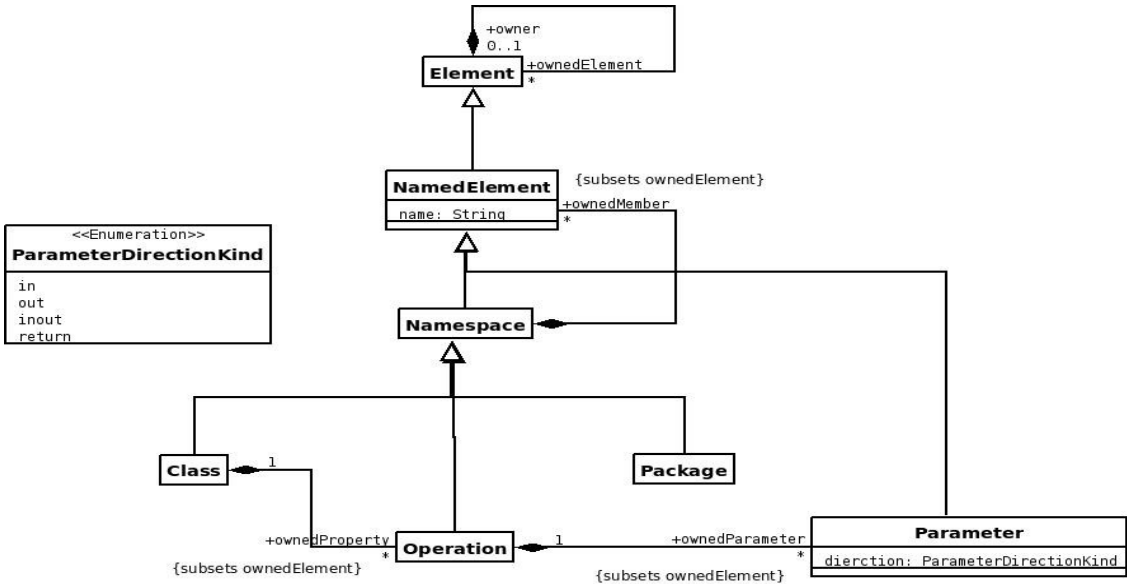


Figure 5.4: Un Fragment Simplifié du *Méta-Modèle* UML 2.1

```
create(p1,Package)
addProperty(p1,name, 'Azureus')
create(c1,Class)
addProperty(c1,name, 'Client')
create(c2,Class)
addProperty(c2,name,'Server')
addReference(p1,ownedMember,c1)
addReference(p1,ownedMember, c2)
addReference(p1,ownedElement,c1)
addReference(p1,ownedElement,c2)
create(o1,Operation)
addProperty(o1,name, 'send')
addReference(c2, ownedProperty, o1)
addReference(c2, ownedElement, o1)
```

Figure 5.5: Séquence Praxis Représentant le *Modèle* Azureus

2. Dans le cas où la propagation est semi-automatique, et dans le but d'assurer une traçabilité de ses choix, il faut trouver le moyen de mémoriser les refus et acceptations opérés par le développeur en charge de *pm2*.

Les champs ***news***, ***includes*** et ***excludes*** (voir chapitre 4, section 4.4.7.3) de l'observateur associe à *pm2* permettent de répondre à la deuxième exigence. En effet, quelque soit le format utilisé pour représenter les éléments de modèles (créés, modifiés, ou supprimés), leurs références sont temporairement enregistrées via le champs ***news***. Celles-ci sont ensuite déplacés vers le champs ***includes*** si elles sont acceptées, et vers le champs ***excludes*** sinon. En revanche, pour la deuxième exigence, il nous a fallu trouver un moyen adapté d'identification du différentiel entre deux versions successives d'un *produit-modèle*. Des solutions existent dans la littérature [Xing 2009; Xing & Stroulia 2005] et permettent de trouver le résultat voulu, mais après des calculs fastidieux de calcul de cette différence. Elles s'avèrent donc inadaptées comparées aux possibilités engendrées par des approches dites orientées opérations de représentation de *modèles* comme Praxis. En effet, selon ces approches, en lieu et place d'envoyer un *produit-modèle* en entier (après modifications) vers le repository, seules sont envoyées les opérations effectuées par le développeur, permettant ainsi d'obtenir directement le différentiel voulu. L'approche orientée opérations de représentation de *modèles* est donc plus adaptée à la représentation des *produit-modèles* par le *PSEE* en cours de conception. Praxis en est une des implémentations les plus récentes.

Toutefois, nous avons jugé nécessaire, pour son utilisation judicieuse, de faire face à une limite de taille de l'outil dans notre contexte. En effet, Praxis est très verbeux; la création, par exemple, d'une simple classe UML est représentée par une séquence de plus d'une dizaine d'actions élémentaires Praxis. Nous sommes alors confrontés à la manipulation d'un très grand nombre d'actions pour un *modèle*, même de petite taille. En plus de cette limite, une manipulation directe des actions Praxis par le code de notre prototype rendrait celui-ci fortement dépendant de l'implémentation de Praxis utilisée.

Pour ces raisons, nous avons choisi d'introduire la notion d'*opération de haut niveau* pour d'une part éviter un couplage intrinsèque du code de notre prototype et de celui de Praxis, et, d'autre part, à la place d'actions unitaires, manipuler des actions agrégées réduisant fortement la verbosité du langage de base utilisé (Praxis). La section suivante est consacrée à la définition des *opérations de haut niveau*.

5.4.3 Les opérations de haut niveau

Pour contrer le verbiage de Praxis, nous définissons une opération de haut niveau comme un

regroupement d'actions élémentaires Praxis afin d'obtenir des actions plus proches du langage humain dans lequel on retrouve des opérations telles que *« créer un élément de modèle »*, *« supprimer un élément »*, *« modifier un élément »*. Nous identifions alors un ensemble d'opérations de plus haut niveau que Praxis (cf. le *méta-modèle* ci-après, **Figure 5.6**) et proposons une génération automatique d'*opérations de haut niveau* à partir d'actions Praxis correspondant à un *modèle* donné. Ainsi notre prototype manipule uniquement ces *opérations de haut niveau* et n'a donc pas connaissance des actions Praxis. Utiliser une autre approche que Praxis ne nécessite alors que le changement du générateur de ces opérations à partir des actions élémentaires.

La **Figure 5.6** représente le *méta-modèle* que nous avons utilisé pour définir les opérations de haut niveau du prototype. Suivant le même style qu'au chapitre précédent ses concepts sont présentés dans les sections qui suivent.

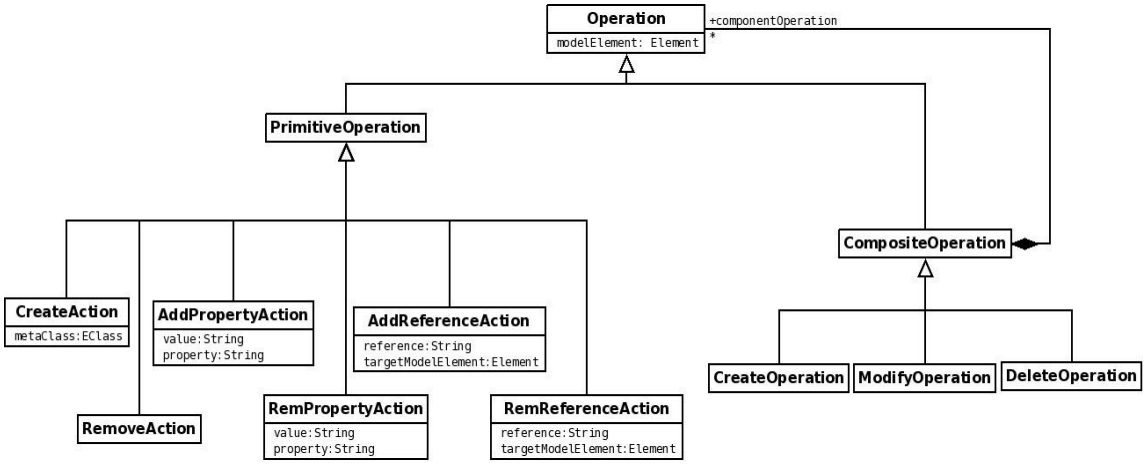


Figure 5.6: *Méta-modèle* des Opérations de Haut Niveau

5.4.4 Operation

5.4.4.1 Description et sémantique

Le concept d'**Opération** est utilisé pour représenter une opération de haut niveau sur un *produit-modèle*. Ce concept est défini pour être spécialisé dans la suite en opérations primitives et composées.

5.4.4.2 Généralisations

Aucune généralisation définie pour le moment.

5.4.4.3 Attributs

modelElement: MOF::Element

Une référence vers l'élément de modèle sur lequel s'applique l'opération. Il est définit sous la forme d'un MOF::Element pour permettre qu'il puisse correspondre à n'importe quel élément d'un *modèle* MOF.

5.4.4.4 Associations

Aucune association n'est définie pour le moment.

5.4.4.5 Contraintes

Aucune contrainte n'est définie pour le moment.

5.4.4.6 Notation

Aucune notation définie pour le moment.

5.4.5 PrimitiveOperation

5.4.5.1 Description et sémantique

Un **PrimitiveOperation** est une opération simple sur un *produit-modèle*. Il n'est donc pas décomposable en d'autres opérations. Ce concept est défini pour correspondre à n'importe quelle action unitaire de Praxis. Il est, à cet effet, défini pour être spécialisé dans la suite en opérations primitives correspondant à ces actions unitaires.

5.4.5.2 Généralisations

- Un **PrimitiveOpération** est un **Operation**.

5.4.5.3 Attributs

Aucun attribut additionnel n'est défini pour le concept pour le moment.

5.4.5.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.5.5 Contraintes

- Une opération primitive ne peut pas s'appliquer sur un élément de modèle qui peut avoir des ***ownedElement***.

5.4.5.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.6 CompositeOperation

5.4.6.1 Description et sémantique

Un **CompositeOperation** est une opération composée sur un *produit-modèle*. Il est composé d'autres opérations. Ce concept est défini pour correspondre à un regroupement d'opérations qui peuvent être primitives ou composées. Il est défini pour être spécialisé dans la suite en concepts correspondant aux opérations de haut niveau que nous définissons.

Nous avons choisi de réaliser le regroupement des opérations primitives (que nous définirons dans la suite) suivant les éléments sur lesquels elles s'appliquent. L'élément sur lequel s'applique une opération correspond, dans cette définition, à celui désigné par le champ ***modelElement*** de l'opération. C'est donc la philosophie définie par cette approche qui sera affinée et employée dans les définitions qui vont suivre de la sémantique donnée aux méta-classes correspondant aux opérations composées (**CreateOperation**, **DeleteOperation** et **ModifyOperation**).

Cette approche de regroupement a l'avantage de sa simplicité et de la précision de sa définition. Elle permet également un regroupement avec un niveau de granularité acceptable par rapport à la deuxième approche que nous avons pu identifier. En effet, cette dernière consiste à mettre ensemble chaque opération primitive qui s'applique à un élément donné avec toutes celles qui s'appliquent aux membres de celui-ci (appelés ***ownedMembers*** selon le MOF). Cette approche pose le problème du risque d'une grosse granularité pour les opérations qui n'est pas adéquate dans notre contexte car pouvant dans certains cas regrouper toutes les

opérations primitives correspondant à un *produit-modèle* sous la forme d'une seule opération composée.

5.4.6.2 Généralisations

- Un **CompositeOpération** est un **Operation**.

5.4.6.3 Attributs

Aucun attribut additionnel n'est défini pour le concept pour le moment.

5.4.6.4 Associations

componentOpération: Operation

Les opérations qui composent le **CompositeOperation**. Elles peuvent être primitives ou composées.

5.4.6.5 Contraintes

- A l'inverse d'une opération primitive, une opération composée s'applique toujours sur un élément de modèle qui peut posséder des ***ownedElement*** (du point de vue du *méta-modèle* MOF).

5.4.6.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.7 CreateOperation

5.4.7.1 Description et sémantique

Un **CreateOperation** sur un ***modelElement*** *me* est l'opération de création de l'élément *me* sur un *produit-modèle*. C'est une opération de haut niveau qui est composée d'autres opérations définies de la façon suivante:

- une parmi elles est une primitive **CreateAction** qui s'applique à l'élément *me*,
- les autres, si elles existent, sont d'autres opérations primitives d'ajout de propriétés (**AddPropertyAction**) dont la cible (***targetModelElement)*** est commune et correspond à l'élément *me*.

Un **CreateOperation** regroupe donc les opérations primitives relatives à la création d'un élément de modèle ainsi qu'à la définitions des propriétés de cet élément.

5.4.7.2 Généralisations

- Un **CreateOperation** est un **CompositeOperation**.

5.4.7.3 Attributs

Aucun attribut additionnel n'est défini pour le concept pour le moment.

5.4.7.4 Associations

Aucune association additionnelle n'est définie pour le moment pour ce concept.

5.4.7.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.7.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.8 DeleteOperation

5.4.8.1 Description et sémantique

Un **DeleteOperation** sur un ***modelElement*** *me* est l'opération de suppression de l'élément *me* d'un *produit-modèle*. Les opérations qui la composent sont définies de la manière suivante:

- une parmi elles est un **RemoveAction** qui s'applique à l'élément *me*,
- les autres sont des opérations primitives de suppression de propriétés (**RemPropertyAction**) dont la cible est commune et correspond à l'élément *me*.

5.4.8.2 Généralisations

- Un **DeleteOperation** est un **CompositeOperation**.

5.4.8.3 Attributs

Aucun attribut additionnel n'est défini pour le concept pour le moment.

5.4.8.4 Associations

Aucune association additionnelle n'est définie pour le moment pour ce concept.

5.4.8.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.8.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.9 ModifyOperation

5.4.9.1 Description et sémantique

Un **ModifyOperation** sur un ***modelElement*** *me* est l'opération de modification de l'élément *me* d'un *produit-modèle*. Les opérations qui la composent sont des opérations primitives d'ajout et de suppression de références (**AddReferenceAction** et **RemReferenceAction**) et de propriétés (**AddPropertyAction** et **RemPropertyAction**) dont la cible est commune et correspond à l'élément *me*.

5.4.9.2 Généralisations

- Un **ModifyOperation** est un **CompositeOperation**.

5.4.9.3 Attributs

Aucun attribut additionnel n'est défini pour le concept pour le moment.

5.4.9.4 Associations

Aucune association additionnelle n'est définie pour le moment pour ce concept.

5.4.9.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.9.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.10 CreateAction

5.4.10.1 Description et sémantique

Un **CreateAction** est une opération primitive de création d'un élément sur un *produit-modèle*. Il correspond à l'action unitaire **Create** de Praxis. Son attribut ***metaClass*** désigne la méta-classe de l'élément créé.

5.4.10.2 Généralisations

- Un **CreateAction** est un **PrimitiveOperation**.

5.4.10.3 Attributs

metaClass: MOF::Class

La méta-classe de l'élément créé.

5.4.10.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.10.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.10.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.11 RemoveAction

5.4.11.1 Description et sémantique

Un **RemoveAction** est une opération primitive de suppression d'un élément sur un *produit-modèle*. Il correspond à l'action unitaire **Delete** de Praxis.

5.4.11.2 Généralisations

- Un **RemoveAction** est un **PrimitiveOperation**.

5.4.11.3 Attributs

Aucun attribut n'est défini pour le moment pour ce concept.

5.4.11.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.11.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.11.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.12 AddReferenceAction

5.4.12.1 Description et sémantique

Un **AddReferenceAction** est une opération primitive d'ajout d'une référence à un élément d'un *produit-modèle*. Il correspond à l'action unitaire **AddReference** de Praxis. L'attribut ***reference*** désigne le nom de la référence à ajouter et ***targetModelElement*** à l'élément vers lequel la référence est définie.

5.4.12.2 Généralisations

- Un **AddReferenceAction** est un **PrimitiveOperation**.

5.4.12.3 Attributs

reference: String

Le nom de la référence à ajouter à l'élément.

targetModelElement: Element

L'élément vers lequel la référence est créée.

5.4.12.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.12.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.12.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.13 RemReferenceAction

5.4.13.1 Description et sémantique

Un **RemReferenceAction** est une opération primitive de suppression d'une référence à un élément d'un *produit-modèle*. Il correspond à l'action unitaire **RemReference** de Praxis. L'attribut ***reference*** désigne le nom de la référence à supprimer et ***targetModelElement*** à l'élément vers lequel la référence est définie.

5.4.13.2 Généralisations

- Un **RemReferenceAction** est un **PrimitiveOperation**.

5.4.13.3 Attributs

reference: String

Le nom de la référence à ajouter à l'élément.

targetModelElement: Element

L'élément vers lequel la référence est créée.

5.4.13.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.13.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.13.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.14 AddPropertyAction

5.4.14.1 Description et sémantique

Un **AddPropertyAction** est une opération primitive d'ajout d'une propriété à un élément d'un *produit-modèle*. Il correspond à l'action unitaire **AddProperty** de Praxis. Son attribut ***property*** désigne le nom de la propriété à ajouter et ***value*** à la valeur de cette propriété.

5.4.14.2 Généralisations

- Un **AddPropertyAction** est un **PrimitiveOperation**.

5.4.14.3 Attributs

property: String

Le nom de la propriété à ajouter à l'élément.

value: String

La valeur de la propriété à ajouter à l'élément.

5.4.14.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.14.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.14.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.4.15 RemPropertyAction

5.4.15.1 Description et sémantique

Un **RemPropertyAction** est une opération primitive de suppression d'une propriété pour un élément d'un *produit-modèle*. Il correspond à l'action unitaire **RemProperty** de Praxis. Son attribut ***property*** désigne le nom de la propriété à supprimer et ***value*** à la valeur de cette propriété.

5.4.15.2 Généralisations

- Un **RemPropertyAction** est un **PrimitiveOperation**.

5.4.15.3 Attributs

property: String

Le nom de la propriété à ajouter à l'élément.

value: String

La valeur de la propriété à ajouter à l'élément.

5.4.15.4 Associations

Aucune association n'est définie pour le moment pour ce concept.

5.4.15.5 Contraintes

Aucune contrainte additionnelle n'est définie pour le moment pour ce concept.

5.4.15.6 Notation

Aucune notation n'est définie pour le moment pour ce concept.

5.5 Réalisation des principaux éléments constitutifs du noyau du *PSEE*

Cette section a pour objectif de présenter comment les éléments constitutifs du prototype développé ont été réalisés en utilisant les outils précédemment présentés. Ainsi nous y présentons brièvement l'éditeur de *modèle de procédé* dans un premier temps. S'ensuivra la présentation de l'environnement qui correspond au *moteur d'exécution*. Enfin, dans le but d'aider à mieux comprendre des éléments déjà décrits du code métier du prototype (comme la construction et l'exécution des opérations de haut niveau, et le stockage des **SPOs**), nous présenterons les algorithmes utilisés pour les implémenter.

5.5.1 L'éditeur de modèle de procédé

Nous avons vu plus haut que EMF permet de générer de façon automatique un éditeur graphique à partir d'un *méta-modèle* et qu'il est ensuite possible avec un tel éditeur de construire des modèles instances du même *méta-modèle*. Nous avons utilisé cette

fonctionnalité pour générer le composant de modélisation du prototype à partir du *méta-modèle* étendu* de *WorkProducts*. Le composant est donc réalisé sous la forme d'une fenêtre Eclipse qui permet, de façon arborescente, de construire un *modèle de procédé* avec la possibilité de spécifier des relations de types *nest* et *overlap* entre *WorkProducts*.

5.5.2 Le moteur d'exécution

L'exécution d'un *procédé* est simulée à travers une fenêtre Eclipse dont une capture est présentée à la **Figure 5.7**.

Cette fenêtre se subdivise en trois parties décrites ci-après.

- La partie centrale (1) permet de visionner les contenus des différents workspaces ainsi que celui du repository de **SPOs** au fur et à mesure que l'exécution du *procédé* progresse. C'est donc cette partie qui simule l'outil de travail du développeur qui peut, en effet, directement modifier les *produits-modèles* du workspace via l'interface définie dans cette partie. Les actions correspondant au travail du développeur sont alors mises à la disposition du module de post exécution.
- La partie de droite (2) contient les différentes étapes du *procédé*. Ici sont considérées les deux premières itérations du procédé-exemple dont l'une sera entièrement exécutée alors que pour la deuxième seule l'activité d'*analyse* sera exécutée. En effet, continuer l'exécution serait une répétition de cette même séquence. Cette partie représente ainsi une vue du *modèle de procédé* au sein duquel les *produits-modèles* sont spécifiés en utilisant les concepts du *méta-modèle* de *WorkProducts*. Enfin, elle présente au développeur des commandes lui permettant de démarrer ou de finir une activité.
- La partie d'en bas (3) est une console permettant au développeur, avant de commencer une activité, de lire des notifications venant du moteur et de prendre des décisions (accepter ou ignorer de nouveaux éléments de modèles dans un *produit-modèle*).

Dans les sous-section qui suivent nous présentons les algorithmes utilisés réaliser la construction des *opérations de haut niveau*, leur stockage dans le repository, et à leur exécution. Ils ont, respectivement, été utilisés dans le codage des composants *Workspace, PostExecution,* et *PreExecution*.

* Le méta-modèle de la Figure 4.1 a été modifié en rajoutant les concepts d'Activity, Role, etc. afin de permettre la modélisation des activités, rôles, etc. du procédé.

5.5.2.1 La construction des opérations de haut niveau à partir des actions unitaires Praxis

Les actions Praxis correspondent à celles du développeur sur un *produit-modèle*. Elles sont livrées par le composant de travail au composant de pré-exécution après un regroupement sous formes d'*opérations de haut niveau* structurées à l'aide du *méta-modèle* de la **Figure 5.6**. Nous présentons ci-après les différentes étapes de la création des *opérations de haut niveau*. En guise d'illustration, à chaque étape, nous présentons celles qui sont créées à partir de la séquence d'actions Praxis de la **Figure 5.5** prise donc comme exemple.

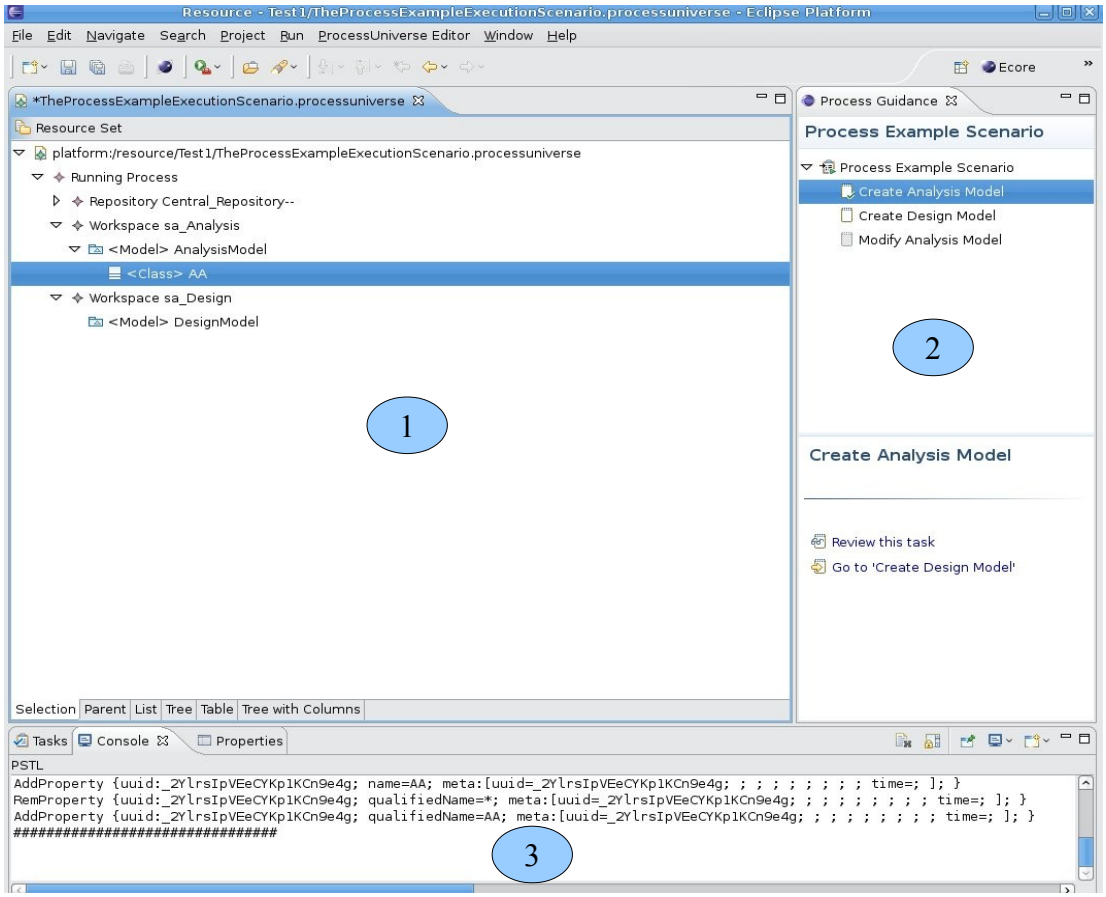


Figure 5.7: Capture de la Fenêtre Eclipse Correspondant au *Moteur d'Exécution*

1. La séquence d'actions unitaires Praxis est considérée dans sa globalité et les correspondances suivantes sont satisfaites:

- un **CreateAction** est créé pour chaque action Praxis **Create**,
- un **RemoveAction** est créé pour chaque action Praxis **Delete**,
- un **AddPropertyAction** est créé pour chaque action Praxis **AddProperty**,
- un **RemPropertyAction** est créé pour chaque action Praxis **RemProperty**,
- un **AddReferenceAction** est créé pour chaque action Praxis **AddReference**,
- un **RemReferenceAction** est créé pour chaque action Praxis **RemReference**.

Dans chacun des cas précédents, l'EObject d'UUID correspondant au champ ID de l'action Praxis est recherché et obtenu à l'aide de la méthode `getEObject()` de la ressource EMF **ProcessUniverse** (la racine du *méta-modèle* de **SPOs**). C'est une référence à cet objet qui va être affectée à l'attribut ***modelElement*** de l'opération primitive. Les autres attributs des opérations nouvellement créées sont renseignés en utilisant les valeurs des champs des actions Praxis correspondants. La correspondance utilisée entre les actions Praxis et les opérations primitives est présentée à l'aide de la **Table 5.1**.

Considérant l'exemple, à cette étape, chaque action unitaire Praxis lue donne lieu à la création de l'opération primitive correspondante.

Remarque importante: Dans l'étape 2 de l'algorithme, nous avons tenu compte du fait que la séquence d'actions unitaires Praxis reçue est ordonnée et qu'elle ne contient pas d'action **Delete**. En effet, l'implémentation de Praxis qui nous a été fournie ne prend pas en compte les actions de suppression

Actions Praxis	Attributs	Attributs	**Opérations de haut niveau**
Create	*Id*	*modelElement*	CreateAction
	mc	*metaClass*	
Delete	*Id*	*modelElement*	RemoveAction
AddProperty	*Id*	*modelElement*	AddPropertyAction
	property	*property*	
	value	*value*	
RemProperty	*Id*	*modelElement*	RemPropertyAction
	property	*property*	
	value	*value*	
AddReference	*Id*	*modelElement*	AddReferenceAction
	reference	*reference*	
	me	*targetModelElement*	
RemReference	*Id*	*modelElement*	RemReferenceAction
	reference	*reference*	
	me	*targetModelElement*	

Tableau 5.1: Correspondances Actions Praxis-Opérations Primitives

2. Après les opérations primitives, les **CompositeOperations** sont créées. Les opérations composées sont créées en regroupant les opérations primitives (de haut niveau) suivant un principe déjà annoncé en section 5.4.6.1 et dont nous donnons puis illustrons les détails suivants:

- La séquence entière d'opérations primitives est considérée,

  pour chaque **CreateAction** de la séquence, un **CreateOperation** est créée à laquelle on ajoute le **CreateAction**, et, puisque la séquence est ordonnée, tous les **AddPropertyAction** du reste de la séquence sont ajoutées à la collection ***operations*** du **CreateOperation**. Les opérations ajoutées au **CreateOperation** nouvellement créé sont alors supprimées de la séquence d'origine.

C'est ainsi que les quatre opérations composées suivantes seront créées, considérant la séquence définie à la **Figure 5.5**. Conformément au *méta-modèle* des *opérations de haut niveau*, pour chaque opération sont fournies (dans cet ordre) la référence vers l'élément de modèle sur lequel elle s'applique (exemple *p1*), le nom de la méta-classe de cet élément (exemple *Package*), et la liste des opérations (primitives) qui la composent.

- **CreateOperation**

```
(
p1,
Package,
{
CreateAction(p1,Package),
AddPropertyAction(p1,name, « Azureus »)
}
)
```

- **CreateOperation**

```
(
c1,
Class,
{
CreateAction(c1, Class),
AddPropertyAction(c1, name, «Client»)
}
)
```

- **CreateOperation**

```
(
c2,
Class,
{
CreateAction(c2, Class),
AddPropertyAction(c2, name, «Server»)
}
)
```

- **CreateOperation**

```
(
o1,
Operation,
{
CreateAction(o1, Operation),
AddPropertyAction(o1, name, «send»)
}
)
```

Comme la séquence d'origine est réduite de toutes les opérations **CreateAction** et **AddPropertyAction**, elle ne contient plus alors que des **AddReferenceAction**.

- Le reste de la séquence est considérée,

pour chaque action rencontrée, un **ModifyOperation** est créée avec son ***modelElement*** égal à celui de l'action en cours et son ensemble ***operations*** rempli avec l'ensemble de toutes les autres opérations de la séquence qui portent sur le même ***modelElement***. Les opérations classées dans le **ModifyOperation** nouvellement créé sont alors supprimées de la séquence

d'origine.

En reprenant le même exemple, les deux opérations suivantes seront créées, dans cet ordre, au cours de cette étape. Conformément au *méta-modèle* des *opérations de haut niveau*, pour chaque opération sont fournies (dans cet ordre) la référence vers l'élément de modèle sur lequel elle s'applique (exemple *p1*), et la liste des opérations (primitives) qui la composent.

- **ModifyOperation**

  (

  p1,

  {

  AddReferenceAction(p1,ownedMember,c1),

  AddReferenceAction(p1, ownedMember, c2),

  AddReferenceAction(p1, ownedElement, c1),

  AddReferenceAction(p1, ownedElement, c2)

  }

)

- **ModifyOperation**

  (

  c2,

  {

  AddReferenceAction(c2, ownedProperty, o1),

  AddReferenceAction(c2, ownedElement, o1)

  }

)

5.5.2.2 Le stockage des éléments de modèles dans le repository

En lieu et place d'un stockage direct des *produits-modèles* dans le repository central, sont

enregistrées les opérations de haut niveau qui correspondent aux actions opérées sur eux par les développeurs depuis les workspaces. Ces opérations sont enregistrées dans les champs ***data*** des **SPOs** par le composant de post-exécution sur la base d'un principe déjà explicité section 5.3.2.1.2.2 et que nous n'avons pas jugé nécessaire de reprendre.

5.5.2.3 L'exécution des *opérations de haut niveau*

Elle est effectuée avant le début de chaque activité d'un développeur (par le composant de pré-exécution). Son objectif est en effet de construire les *produits-modèles* dont un développeur a besoin et de les fournir au workspace de celui-ci. Elle se fait suivant le principe suivant.

Les opérations de haut niveau qui correspondent au *produit-modèle* à reconstruire sont d'abord collectées. Elles correspondent à celles contenues dans le ou les **SPO(s)** qui représentent le *produit-modèle* au niveau du repository central, en sus d'éventuelles opérations stockées par les **SPOObservers** impliqués s'ils existent. Il est en effet nécessaire, pour obtenir le *produit-modèle* en cours de reconstruction, de supprimer de l'ensemble des opérations partagées (et donc stockées dans un **OverlapSPOR**), toutes celles qui ont été rejetées par le développeur au cours du *procédé* et qui sont, par conséquent, marquées à l'aide de champs ***excludes*** d'observateurs. Une fois toutes les opérations collectées, la séquence correspondante est exécutée de la façon suivante.

1. Les **CreateOpération** sont exécutées dans un premier temps. Tous les éléments de *produit-modèle* sont ainsi créés et leurs différentes propriétés (*name*, *visibility*, etc.) éventuellement renseignées.

2. Les **ModifyOperation** sont ensuite exécutées. Les références nécessaires à l'interconnexion des éléments de modèles créés à l'étape 1 sont créées. Cette étape peut également modifier des propriétés de ces éléments de modèles.

L'exécution d'une opération composée correspond à une exécution, dans l'ordre, des opérations primitives qui la composent. Chaque opération primitive est exécutée en effectuant l'action correspondante au sein du *produit-modèle* en cours de création.

5.6 Discussions

Les préoccupations majeures soulevées en définissant la problématique de la thèse (cf chapitre 2) étaient liées à l'expression et à l'exploitation des relations entre *produits-modèles* de *procédés*. C'est ainsi que nous nous sommes, plus spécifiquement, intéressés à la *granularité* des *produits-modèles*, de même qu'à la présence d'éléments communs entre ces *produits-modèles*. Ces deux problèmes ont respectivement été illustrés à travers, d'une part, le *modèle d'analyse* et, d'autre part, le *modèle d'analyse* et le *modèle de conception* du procédé-exemple. Ils ont ensuite été pris en charge à travers notre approche qui propose de spécifier ces relations lors de la modélisation de *procédés*, et ensuite d'utiliser les spécifications résultantes à l'exécution afin d'assurer une gestion plus adaptées des *produits-modèles*. C'est dans le but d'outiller cette approche que le *PSEE* présenté dans ce chapitre a été conçu et son prototype réalisé. Le *PSEE* définit un cadre de modélisation de *procédé* introduisant les concepts de **Nest** et de l'**Overlap**, et d'exécution de ces *procédés* avec la prise en compte des spécifications qui correspondent à ces concepts et qui sont relatives aux deux relations entre *produits-modèles* sus-nommées.

Dans le but de mettre en exergue l'apport de cette approche, nous comparons ses principales fonctionnalités avec celles des approches que nous avons identifiées et étudiées dans l'état de l'art sur la gestion de *procédés*. C'est ainsi que le **Tableau 5.2** montre que notre approche se distingue de celles rencotrées dans la littérature par un support entier de la modélisation des relations de types *nest* et *overlap* et à leur prise en compte à l'exécution. Elle permet ainsi d'assurer une cohérence relation, une évolution automatique et dynamique et une modularité des *produits-modèles* à l'exécution des *procédés*.

Par ailleurs, le *moteur d'exécution* que nous avons prototypé comporte des avantages considérables par rapport aux *VCS* de type traditionnel comme SVN [Collins-Sussman, Fitzpatrick & Pilato 2008a], mais aussi à des environnements plus récents et spécifiques à la gestion de *produits-modèles* comme ModelBus [P Sriplakich, X Blanc, et al. 2006; SRIPLAKICH 2007]. De façon générale, il faut dire, comme on l'avait souligné au chapitre 3 traitant de l'état de l'art, que les environnements de type SVN ne peuvent être utilisés pour gérer les *produits-modèles*. Par rapport aux environnements de gestion de *modèles* comme ModelBus, notre solution se démarque principalement par le support d'une prise en compte de

spécifications relatives à des relations de type *overlap* et *nest*. C'est ainsi que notre proposition constitue une solution pour certaines faiblesses de ces environnements en permettant une gestion modulaire des *produits-modèles* de *procédés*, leur évolution automatique et dynamique, ainsi qu'un maintien systématique de leur *cohérence relationnelle*.

APPROCHES	MODELISATION	EXECUTION			
	Relations	Cohérence Relationnelle	Évolution Automatique	Évolution Dynamique	Modularité
Notre approche	*Nest* *Overlap*	Oui	Oui	Oui	Oui
APPL/A	*Dérivation* *Nest*	**Non**	Oui	**Non**	Oui *
MSL/MARVEL	*Dérivation* *Nest* *Overlap*	**Non**	Oui	**Non**	Oui *
SLANG/SPADE	*Nest*	**Non**	Oui	**Non**	Oui *
TEMPO/ADELE	*Dérivations* *Nest* *Overlap* (**seulement entre 2 *produits***) ...	**Non**	Oui	**Non**	Oui *
Di Nito et al.	**Aucune**	**Non**	**Non**	**Non**	**Non**
Chou	*Nest*	**Non**	**Non**	**Non**	**Non**
UML4SPM	**Aucune**	**Non**	**Non**	**Non**	**Non**
SPEM2	*Nest* *Dérivation* *Overlap* etc.	**Non**	**Non**	**Non**	**Non**

Tableau 5.2 : Comparaison de notre Approche de Gestion de *Procédé* et de celles de l'Etat de l'Art.

Malgré ses avantages, notre outil présente un certains nombre de limites par rapport à certaines approches et outils que nous avons étudiés. Par exemple, notre approche ne supporte pour le moment que les relations de type *nest* et *overlap* là où certaines approches (voir **Tableau 3.1**) supportent d'autres relations comme la *dérivation*. Quant à elles, les limites actuelles du *moteur d'exécution* conçu et prototypé sont essentiellement relatives à

l'impossibilité de gérer plusieurs versions successives d'un *produit-modèle*, et au non support de la collaboration.
Nous reviendrons sur ces faiblesses de l'approche en dégageant les perspectives ouvertes par ce travail dans le dernier chapitre de cette thèse.

5.7 Conclusion

Dans ce chapitre nous avons présenté un outillage et une application de notre proposition sur un exemple. Nous l'avons fait à travers la construction d'un *PSEE* comprenant un environnement de modélisation et d'exécution de *procédés*.
L'environnement de modélisation permet de spécifier des *procédés* prenant en compte les concepts proposés dans notre approche. L'environnement d'exécution quant à lui permet de lire et d'exécuter les *modèles* qui résultent de cette spécification. Il est construit suivant une architecture de type workspace-repository qui détermine d'une part comment les *produits-modèles* sont stockés dans une base centrale et d'autre part comment ils sont utilisés par les développeurs. C'est ainsi qu'à travers la présentation de son mode de fonctionnement, nous avons montré les différentes formes prises par ces *produits-modèles*. En effet, les développeurs utilisent des outils de leurs workspaces respectifs qui sont capables de livrer leurs actions sous la forme d'opérations à travers lesquelles les *produits-modèles* sont gérés au niveau du repository.
Le *PSEE* prototypé est réalisé sous Eclipse EMF qui est présenté dans le chapitre de même que l'approche Praxis de représentation orientée opérations de *modèles*. Les plus importantes étapes de sa réalisation sont présentées de même que les principaux algorithmes implémentés pour réaliser le métier du *moteur d'exécution*. Le chapitre présente également l'utilisation du prototype développé pour la modélisation et l'exécution d'un procédé-exemple. Il finit par un ensemble de discussions et de commentaires dont le but est de mettre en exergue les apports de notre approche par rapport à d'autres existantes.

Chapitre 6

Conclusion Générale et Perspectives

6.1 Conclusion

Dans cette thèse, nous nous sommes intéressés à la problématique de la spécification et de l'exploitation des relations entre *produits-modèles*, respectivement à la modélisation et à l'exécution de *procédés* considérés dans le contexte IDM. Nous avons défini et illustré le contexte et les motivations de cet intérêt. Nous avons ensuite présenté la solution que nous avons proposée. La contribution de cette thèse s'articule autour des trois points qui suivent.

Du point de vue de la modélisation des *procédés*.

Nous avons proposé un *méta-modèle* permettant de structurer les éléments utilisés lors de la modélisation d'un *procédé* pour décrire les *produits-modèles*. Ce *méta-modèle* contient les concepts nécessaires à la définition des relations entre ces différents éléments et d'associer à ces relations les caractéristiques nécessaires à leur exploitation par un *moteur d'exécution* pour une meilleure gestion des *produits-modèles* impliqués. Notre approche supporte, pour le moment, les relations qui sont relatives à l'inclusion et au partage d'éléments entre deux ou plusieurs *produits-modèles* d'un *procédé* en exécution.

Du point de vue de l'exécution des *procédés*.

Nous avons proposé un autre *méta-modèle* dont le but est de structurer les entités logiques à travers lesquelles un *moteur d'exécution* gère les *produits-modèles* d'un *procédé*. Ce deuxième *méta-modèle* contient les concepts nécessaires à la représentation des éléments de procédé correspondants aux *produits-modèles* ainsi qu'aux relations prises en compte dans le premier *méta-modèle*.

Notre proposition comprend également des règles de transformation qui expriment une correspondance entre les concepts des deux *méta-modèles* proposés. Comme nous l'avons indiqué plus haut, les apports de notre approche sont relatifs à la gestion des *produits-modèles* de *procédés* en exécution. Nous les avons résumés à travers les points suivants.

Apports relatifs à l'overlap.

- Une *cohérence relationnelle* systématique des *produits-modèles* qui se partagent des éléments communs.
- Une synchronisation, également systématique, de ces mêmes *produits-modèles*.

Ces deux apports sont la conséquence du stockage unique des éléments partagés que nous avons proposé. Notre approche supporte également une synchronisation interactive, de *produits-modèles* en relation.

- Une construction assistée des *produits-modèles*. L'approche supporte la création de certains *produits-modèles* par un mécanisme d'import automatique d'éléments d'autres *produits-modèles* construits lors des étapes antérieures du *procédé* et avec lesquels ils ont des éléments communs. Ce mécanisme peut également être rendu interactif.

Apports relatifs au nest.

- Une flexibilité du point de vue de la granularité des *produits-modèles*. Notre approche permet aux outils d'un *procédé* d'avoir un accès aussi bien à des parties précises d'un *produit-modèle* sous la forme d'entités séparées qu'au même *produit-modèle* considéré globalement.
- Une intégrité ou cohérence sémantique des *produits-modèles*. A travers les caractéristiques associées à une relation de type *nest*, l'approche introduit des liens de dépendance d'un nouveau type entre les *produits_modèles* ou partie de *produits-modèles*. Ces dépendances permettent, par exemple, de justifier l'existence ou l'inexistance d'un *produit-modèle* donné en fonction de celle d'un ou de plusieurs autres. Elles permettent également d'indiquer l'appartenance unique ou partagée d'une partie à un *produit-modèle* et contribuent ainsi à rendre possible un contrôle basé sur les *produits-modèles* de l'exécution d'un *procédé*.

L'expérimentation.

Nous avons conçu et réalisé un prototype qui simule un environnement de modélisation et d'exécution de *procédés* suivant notre approche. Cette environnement s'appuie sur une représentation orientée-opération des *produits-modèles*. Il a été réalisé sous la forme d'une application Eclipse et avec l'aide du cadre EMF et de Praxis, une approche de représentation orientée-opération et de gestion de la cohérence de *modèles*. Afin de mieux adapter l'approche Praxis à notre contexte, nous avons proposé un *méta-modèle* permettant de regrouper les actions élémentaires qu'elle définit sous la forme d'opérations de plus haut niveau. Aussi, notre approche de regroupement est adaptée à toute autre approche permettant, comme Praxis, de représenter un *produit-modèle* sous la forme d'une séquence formée des actions élémentaires qui ont lieu lors de sa construction.

6.2 Perspectives

Les relations entre les relations entre *produits-modèles*.

Au chapitre 2 nous avons identifié plusieurs types de relations possibles entre *produits-modèles*. Notre approche ne supporte que deux de ces relations, le *nest* et l'*overlap*. Notre attention a surtout été attirée par le fait que pour la plupart des autres relations, il est fréquent que les *produits-modèles* impliqués aient des éléments communs. Cette situation renvoie presque toujours à l'*overlap*. C'est ainsi que nous avons commencé à réfléchir sur la possibilité d'exprimer certaines relations en fonction d'autres. L'objectif est d'arriver, à terme, à trouver une base canonique formée au moins des relations *nest* et *overlap* et à partir de laquelle toute autre relation entre *produits-modèles* pourra être exprimée.

Les services associés aux relations entre *produits-modèles*.

Une pleine exploitation d'une relation entre des *produits-modèles* par *un moteur d'exécution* nécessite que celui-ci dispose de mécanismes spécifiques à la relation considérée. Ces mécanismes se définissent en terme d'opérations, de services applicables aux *produits-modèles* et certains d'entre eux n'existent pas dans la littérature. Par exemple, la relation *nest* s'associe, de ce point de vue, aux opérations de fusion et de fragmentation de *modèles*. En effet, il est nécessaire de fusionner des modèles composants pour obtenir le modèle composé. De même, il est nécessaire de fragmenter un modèle composé pour retrouver ses composants. Notre approche supporte en ce moment une fragmentation

systématique et statique des *produits-modèles* basée sur le *modèle de procédé* et cela dès l'initialisation du procédé. Dans le but d'assurer un plein support de la relation *nest*, une fragmentation dynamique des *produits-modèles* (qui s'appuie toujours sur le *modèle de procédé*) est nécessaire.

Vers des *CMSs*/*VCSs* intégrant le *procédé*.

Dans son état actuel, notre approche supporte un déroulement séquentiel des activités de *procédés*. Elle ne prend pas en compte, en effet, les accès simultanés aux *produits-modèles* en relation. Il faut noter que très souvent, l'exécution d'un *procédé* fait intervenir plusieurs développeurs qui dans certains cas travaillent de façon simultanée. Une autre étude est alors nécessaire pour permettre à l'approche de supporter cette réalité. Il s'agit de travailler à rendre possible le stockage des différentes versions des *produits-modèles* et la gestion des accès simultanés.

Références Bibliographiques

Altmanninger, K., Seidl, M. & Wimmer, M., 2009. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3), pp.271-304.

Ambriola, V., Conradi, R & Fuggetta, A, 1997. Assessing process-centered software engineering environments. *ACM Transactions on Software Engineering and Methodology*, 6(3), pp.283-328.

Anderson, B., 1988. Object-oriented programming. *Microprocessors and Microsystems*, 12(8), pp.433-442.

Atkinson, M. et al., 1992. The object-oriented database system manifesto. In San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 1-20.

Bandinelli, S. et al., 1994. SPADE : An Environment for Software Process Analysis, Design and Enactment. *Software Process Modeling and Technology*, (6115), pp.223-247.

Barghouti, N.S., 1992. Supporting cooperation in the Marvel process-centered SDE. *SIGSOFT Softw. Eng. Notes*, 17(5), pp.21-31.

Baudry, Benoit et al., 2005. Exploring the Relationship between Model Composition and Model Transformation. In O. Aldawud et al., eds. *Aspect Oriented Modeling AOM Workshop*. Citeseer, pp. 2-6.

Belkhatir, N, Estublier, J & Melo, W L, 1993. Software Process Model and Workspace Control in the Adele/Tempo System. In *2nd International Conference on the Software Process*. IEEE Computer Society Press, pp. 2-12.

Belkhatir, Noureddine, Melo, Walcélio L, et al., 1992. Supporting software maintenance evolution processes in the Adele system. *Proceedings of the 30th annual Southeast*

regional conference on ACMSE 30, p.165.

Belkhatir, Noureddine, Estublier, Jacky & Melo, W., 2007. THE ADELE-TEMPO experience: an environment to support process modeling and enaction. *Software Process Modelling and Technology Research Studies Press*, pp.1-37.

Belkhatir, Noureddine, Melo, Walcelio L & Adam, J.-michel, 1992. TEMPO: a software process model based on object context behavior. In *In Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*. pp. 733-742.

Bendraou, R et al., 2007. Software Process Modeling and Execution: The UML4SPM to WS-BPEL Approach. *33rd EUROMICRO Conference on Software Engineering and Advanced Applications EUROMICRO 2007*, (Seaa), pp.314-321.

Bendraou, Reda, 2007. *UML4SPM: Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle*. Université Pierre & Marie Curie – Paris VI.

Bendraou, Reda, Blanc, Xavier & Gervais, M.-P., 2010. A Comparison of Six UML-Based Languages for Software Process Modeling. *IEEE Transactions on Software Engineering*, 36(5), pp.662-675.

Bendraou, Reda, Gervais, M.-P. & Blanc, Xavier, 2006. UML4SPM: An Executable Software Process Modeling Language Providing High-Level Abstractions. *Enterprise Distributed Object Computing Conference, IEEE International*, 0, pp.297-306.

Bendraou, Reda, Gervais, M.-P. & Blanc, Xavier, 2005. UML4SPM : A UML2 . 0-Based Metamodel for Software L. Briand & C. Williams, eds. *MoDELS'05*, 6(511731), pp.17-38.

Bendraou, Reda et al., 2008. Vers l'Exécutabilité des Modèles de Procédés Logiciels. In *14ieme colloque international sur les Langages et Modèles à Objets (LMO'08)*. Montreal, Canada: Revue des Nouvelles Technologies de l'Information (RNTI), pp. 155-170.

Blanc, Xavier et al., 2008. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th international conference on Software*

engineering. New York, NY, USA: ACM, pp. 511-520.

Boehm, B. et al., 1998. Using the WinWin spiral model: a case study. *Case Study, Computer*, 31(7), pp.33-44.

Bottoni, P. et al., 2008. Maintaining Coherence Between Models With Distributed Rules: From Theory to Eclipse. *Electronic Notes in Theoretical Computer Science*, 211, pp.87-98.

Budinsky, F., 2003. *Eclipse Modeling Framework*, Addison-Wesley Professional.

Bézivin, J. et al., 2006. A Canonical Scheme for Model Composition. In A. Rensink & J. Warmer, eds. *Model Driven Architecture–Foundations and Applications*. Springer-Verlag, pp. 346-360.

Cass, A.G. et al., 2000. Little-JIL/Juliette: a process definition language and interpreter. In *Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA: ACM, pp. 754-757.

Cederqvist, P. & Pesch, R., 1993. Version management with CVS. *Signum Support AB*.

Chou, S.-C., 2002. A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language. *Journal of Object Technology*, 1(4), pp.137-163.

Chou, S.C. & Jason Chen, J.Y., 2000. Process program development based on UML and action cases: Part 1: The model. *JOOP Journal of ObjectOriented Programming*, 13(Compendex), pp.21-27.

Collins-Sussman, B., Fitzpatrick, B.W. & Pilato, C.M., 2008a. *Version Control with Subversion* M. E. Treseler, ed., O'Reilly Media.

Collins-Sussman, B., Fitzpatrick, B.W. & Pilato, C.M., 2008b. *Version Control with Subversion, For Subversion 1.5*, Red-bean.com.

Conradi R., Fernström C., Fuggeta A., S.B., 1992. Towards a Reference Framework for Process Concepts. *Proc. Second European Workshop on Software Process Technology*, Lecture No(Springer-Verlag Ed), p.635.

Conradi, Reidar & Westfechtel, B., 1998. Version models for software configuration management. *ACM Computing Surveys*, 30(2), pp.232-282.

Cugola, G, Di Nitto, E. & Fuggetta, A, 2001. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), pp.827-850.

Cugola, G, Nitto, E.D. & Fuggetta, A, 1998. Exploiting an event-based infrastructure to develop complex distributed systems. *Proceedings of the 20th International Conference on Software Engineering*, pp.261-270.

Cugola, Gianpaolo & Ghezzi, C., 1998. Software Processes: a Retrospective and a Path to the Future. *Software Process: Improvement and Practice*, 4(3), pp.101-123.

Curtis, B., Kellner, Marc I & Over, J., 1992. Process modeling D. L. Levin & F. C. Morriss, eds. *Communications of the ACM*, 35(9), pp.75-90.

Dart, S., 1991. Concepts in Configuration Management Systems. *3rd international workshop on Software configuration management*, pp.1-18.

Diaw, S., 2011. *SPEM4MDE: un métamodèle et un environnement pour la modélisation et la mise en oeuvre assistée de processus IDM*. Université de Toulouse II - Université de Ziguinchor.

Diaw, S., Lbath, R. & Coulette, B., 2011. Specification and Implementation of SPEM4MDE, a metamodel for MDE software processes (regular paper). In *International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Miami - USA: Knowledge Systems Institute, pp. 646-653.

Diaw, S. et al., 2010. SPEM4MDE: a Metamodel for MDE Software Processes Modeling and Enactment. (regular paper). In *Third Workshop on Model-Driven Tool & Process Integration. Associated to ECMFA (European Conference on Modelling Foundations and Applications), Paris, 15/06/2010-18/06/2010*.

EMF-Website, 2011. Eclipse Modeling Framework. Available at: http://www.eclipse.org/emf/ [Accessed 2011].

Eclipse-Website, 2011. Eclipse Platform. Available at: http://www.eclipse.org [Accessed October 19, 2011].

Edwards, J., Jackson, D. & Torlak, E., 2004. A type system for object models. *ACM SIGSOFT Software Engineering Notes*, 29(6), p.189.

Egyed, Alexander, 2007. Fixing Inconsistencies in UML Design Models. *29th International Conference on Software Engineering ICSE07*, pp(May), pp.292-301.

Estublier, J & Casallas, R., 1994. The Adele Configuration Manager. *Configuration Management*, 2(C), pp.99-134.

Estublier, Jacky et al., 2005. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4), pp.383-430.

Farah, M., Molter, G. & Schumann, T., 2009. A Model Repository for Collaborative Modeling with the Jazz Development Platform. In *System Sciences 2009 HICSS 09 42nd Hawaii International Conference on*. IEEE, pp. 1-10.

Favre, J.-M., 2004. Foundations of Meta-Pyramids : Languages vs . Metamodels. *Proceedings of the International Seminar on Language*, pp.1-28.

Feiler, P.H. & Kaiser, G.E., 1987. Granularity issues in a knowledge-based programming environment. *Information and Software Technology*, 29(10), pp.531-539.

Fowler, G., Korn, D. & Rao, H., 1995. n-DFS: the Multiple Dimensional File System. In New York, NY, USA: John Wiley & Sons, Inc., pp. 135-154.

Fuggetta, Alfonso, 2000. Software process: a roadmap. In A. F. ED, ed. *Conference on The Future of Software Engineering*. Limerick, Ireland: ACM, pp. 25-34.

Gerber, A. et al., 2002. Transformation : The Missing Link of MDA A. Corradini et al., eds. *Graph Transformation*, 2505, pp.90-105.

Gruhn, V., 2002. Process-Centered Software Engineering Environments, A Brief History and Future Challenges. *Annals of Software Engineering*, 14(1-4), pp.363–382.

Hailpern, B. & Tarr, P., 2006. Model-driven development: The good, the bad, and the ugly.

IBM Systems Journal, 45(3), pp.451-461.

Herrmannsdoerfer, M. & Koegel, M., 2010. Towards a generic operation recorder for model evolution. *Proceedings of the 1st International Workshop on Model Comparison in Practice IWMCP 10*, p.76.

Hosier, W.A., 1987. Pitfalls and safeguards in real-time digital systems with emphasis on programming. In *ICSE 87 Proceedings of the 9th international conference on Software Engineering*. {IEEE} Computer Society Press, pp. 311-327.

Humphrey, W.S. & Kellner, M I, 1989. Software Process Modeling: Principles Of Entity Process Models. *11th International Conference on Software Engineering*, (February), pp.331-342.

Hürsch, W.L. & Lopes, C.V., 1995. Separation of Concerns.

ISO-SEMDM, 2007. Software Engineering Metamodel for Development Methodologies. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38854 [Accessed October 17, 2011].

Johann, S. & Egyed, A, 2004. Instant and incremental transformation of models. *Proceedings 19th International Conference on Automated Software Engineering 2004*, (September), pp.362-365.

Kaiser, G., Barghuti, N. & Sokolsky, M., 1990. Preliminary Experience with Process Modeling in the Marvel SDE Kernel. In *Proceedings IEEE 23th Hawaii ICSS Software Track*.

Kim, W., 1989. Composite Objects Revisited. *SIGMOD Annual Conference on Management of Data*, 21, pp.589-632.

Kim, W. et al., 1987. Composite Object Support in an Object-Oriented Database System. In *Proceedings OOPSLA 87 ACM SIGPLAN Notices*. pp. 118-125.

Lampson, B.W. & Schmidt, E.E., 1983. Practical use of a polymorphic applicative language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, pp. 237-255.

Linternaute-Website, 2011. Le Portail Linternaute. Available at: http://www.linternaute.com/ [Accessed April 20, 2011].

Lonchamp, J., 1993. A structured conceptual and terminological framework for software process engineering. *1993 Proceedings of the Second International Conference on the Software ProcessContinuous Software Process Improvement*, pp.41-53.

Mens, T & Gorp, P.V., 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152(GraMoT), pp.125-142.

Mens, Tom, Van Der Straeten, R. & D'Hondt, M., 2006. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *9th International Conference on Model-Driven Engineering Languages and Systems*. Genoa, Italy.

Mili, H., Elkharraz, A. & Mcheick, H., 2004. Understanding Separation of Concerns B. Tekinerdougan et al., eds. *Aspect-Oriented Requirements Engineering and Architecture Design*, pp.75-84.

Mougenot, A., 2010. *Praxis: Détection des incohérences dans les modèles répartis*. Université Pierre et Marie Curie - Paris 6.

Mougenot, A., Blanc, Xavier & Gervais, M.-P., 2010. Inconsistency Detection in Distributed Model Driven Software Engineering Environments. In *Proceedings of the third Workshop on Living with Inconsistencies in Software Development LWI10*. pp. 6-11.

Muller, P.-A. et al., 2010. Modeling modeling modeling. *Software Systems Modeling*, (i), pp.1-13.

Murta, L. et al., 2007. Odyssey-SCM: An integrated software configuration management infrastructure for UML models. *Science of Computer Programming*, 65(3), pp.249-274.

Méndez Fernández, D. et al., 2010. A Meta Model for Artefact-Orientation: Fundamentals and Lessons Learned in Requirements Engineering. *Model Driven Engineering Languages and Systems*, pp.183–197.

Nentwich, C. et al., 2002. Xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2), pp.151-185.

Di Nitto, E. et al., 2002. Deriving executable process descriptions from UML. In *Proceedings of the 24th International Conference on Software Engineering ICSE 2002*. Institute of Electrical and Electronics Engineers Computer Society, pp. 155-165.

O2 Technology, 1992. *The O2 manual*,

OASIS-WSBPEL, 2007. Web Services Business Process Execution Language Version 2.0. Working Draft.

OMG-BPMN2, 2011. Business Process Model and Notation (BPMN), Specification, Version 2.0. *OMG Document Number: dtc/2010-06-05*, (January). Available at: http://www.omg.org/spec/BPMN/2.0/ [Accessed October 19, 2011].

OMG-MOF2, 2006. Meta Object Facility (MOF) Core Specification Version 2.0. Available at: http://www.omg.org/spec/MOF/2.0/ [Accessed October 17, 2011].

OMG-OCL, 2010. Object Constraint Language. *International Business*, 03(May), pp.852-852.

OMG-QVT, 2008. Meta Object Facility (MOF) 2 . 0 Query / View / Transformation Specification. *OMG Document*, (January), pp.1-230.

OMG-SPEM1, 2002. Software Process Engineering Metamodel Specification. *OMG Document*, (January).

OMG-SPEM2, 2008. Software & Systems Process Engineering Metamodel (SPEM) Version 2.0. Available at: http://www.omg.org/spec/SPEM/2.0/ [Accessed October 17, 2011].

OMG-SPEM2-RFP, 2005. SPEM2.0 RFP, "Software Process Engineering Metamodel." Available at: http://www.omg.org/docs/ad/04-11-04.pdf [Accessed April 4, 2005].

OMG-UML-Exe, 2008. Semantics of a Foundational Subset for Executable UML Models. *OMG Document*, (March). Available at: http://www.omg.org/docs/ad/08-03-09.pdf [Accessed October 17, 2011].

OMG-UML2, 2007. OMG UML Superstructure v2.1.2. , (November). Available at: http://www.omg.org/cgi-bin/doc?formal/05-07-04 [Accessed October 19, 2011].

OMG-UML2, 2009. OMG Unified Modeling Language TM (OMG UML), Infrastructure.

OMG Document, 21(February).

OMG-XMI, 2005. XML Metadata Interchange (XMI) Specification. *An Adopted Specification*, 01(May).

Oliveira, H., Murta, L. & Werner, C., 2005. Odyssey-VCS: a flexible version control system for UML model elements. In *12th international workshop on Software configuration management*. ACM, pp. 1-16.

Osterweil, Leon, 1987. Software processes are software too. In *Proceedings of the 9th international conference on Software Engineering*. IEEE Computer Society Press, pp. 2-13.

Pons, C., Giandini, R. & Baum, G., 2000. Dependency relations between models in the Unified Process. *Tenth International Workshop on Software Specification and Design IWSSD10 2000*, pp.149-157.

Prieto-Díaz Rubén, N.J., 1986. Module interconnexion language. *Journal of Systems Software*, 6(4), pp.307-334.

Qvt-Merge-Group, 2005. Revised submission for Query / View / Transformation RFP (ad / 2002-04-10).

Reddy, R. et al., 2005. Model Composition - A Signature-Based Approach. In O. Aldawud et al., eds. *Aspect Oriented Modeling AOM Workshop*.

Rothenberg, J., 1989. The nature of modeling. In L. E. Widman, K. A. Loparo, & N. R. Nielsen, eds. *Artificial intelligence simulation modeling*. John Wiley & Sons, Inc., pp. 75-92.

Royce, W.W., 1970. Managing the development of large software systems. In *IEEE WESCON*. Los Angeles, pp. 1-9.

Ráth, I., Varró, G. & Varró, D., 2009. Change-driven model transformations. *Software Systems Modeling*, pp.342-356.

SRIPLAKICH, P., 2007. *ModelBus : un environnement réparti et ouvert pour l'ingénierie de modèles*. Université Pierre et Marie Curie - Paris 6.

Scacchi, W., 2001. Process Models in Software Engineering S. Ghosh, ed. *Encyclopedia of Software Engineering 2nd Edition*, 6002(May), pp.1-24.

Sendall, S., 2003. Combining Generative and Graph Transformation Techniques for Model Transformation : An Effective Alliance ? *2nd OOPLSLA WorkShop on Generative Technoqies in the Contexte of MDA*.

Shen, H. & Sun, C., 2002. Flexible Merging for Asynchronous Collaborative Systems. In R. Meersman & Z. Tari, eds. *Confederated International Conferences DOA, CoopIS and ODBASE*. Springer, pp. 304-321.

Silva, M. & Oliveira, T., 2011. Towards Detailed Software Artifact Specification with SPEMArti. *International Conference on on Software and Systems Process*, pp.213-217.

Sommerville, I., 2006. *Software engineering* 8th ed. Addison-Wesley, ed.,

Spanoudakis, G. & Finkelstein, Anthony, 1998. A Semi-automatic process of Identifying Overlaps and Inconsistencies between Requirement Specifications. In *In Proceedings of the 5th International Conference on ObjectOriented Information Systems OOIS 98*. OOIS, pp. 405–424.

Spanoudakis, G., Finkelstein, Anthony & Till, D., 1999. Overlaps in Requirements Engineering. *Automated Software Engineering*, 6(2), pp.171-198.

Sriplakich, P, Blanc, X & Gervais, M., 2006. Supporting transparent model update in distributed CASE tool integration. In H. Haddad, ed. *SAC*. ACM, pp. 1759-1766.

Sriplakich, Prawee, Blanc, Xavier & Gervais, M.-P., 2008. Collaborative Software Engineering on Large-scale models : Requirements and Experience in ModelBus. In *ACM Symposium on Applied Computing*. ACM, pp. 674-681.

Sriplakich, Prawee, Blanc, Xavier & Gervais, M.-P., 2006. Supporting Collaborative Development in an Open MDA Environment. In *ICSM*. Ieee, pp. 244-253.

Steel, J & Lawley, M, 2004. Model-Based Test Driven Development of the Tefkat Model-Transformation Engine. *Reliability Engineering*, pp.151-160.

Steel, Jim & Jézéquel, J.-M., 2005. Model Typing for Improving Reuse in Model-Driven

Engineering. In L Briand & C Williams, eds. *Proceedings of the 8th IEEEACM International Conference on Model Driven Engineering Languages and Systems MoDELS*. Springer, pp. 84-96.

Steel, Jim & Jézéquel, J.-M., 2007. On model typing. *Software Systems Modeling*, 6(4), pp.401-413.

Sutton, S., Heimbigner, D. & Osterweil, L, 1990. Language Constructs for Managing Change in Process-Centered Environments R. Taylor, ed. *Proc 4th ACM Symp on Software Development Environments*, 15:6, pp.206-217.

Tichy, W.F., 1985. Rcs — a system for version control. *Software Practice and Experience*, 15(7), pp.637-654.

WFMC, 2008. Standard Process Definition Interface - XML Process Definition Language. Available at: http://www.wfmc.org/xpdl.html [Accessed October 19, 2011].

Wikipedia-Website, 2011. Wikipedia. Available at: http://www.wikipedia.org/ [Accessed April 20, 2011].

Xing, Z., 2009. GenericDiff: A general framework for model comparison, *Technical Report, NUS.*

Xing, Z. & Stroulia, E., 2005. UMLDiff: an algorithm for object-oriented design differencing. In *20th IEEE/ACM international Conference on Automated software engineering*. ACM, pp. 54-65.

Xiong, Y. et al., 2007. Towards automatic model synchronization from model transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, pp. 164-173.

Zamli, K.Z. & Lee, P.A., 2001. Taxonomy of Process Modeling Languages. *Proceedings ACSIEEE International Conference on Computer Systems and Applications*, pp.435-437.